|  |
| --- |
| Code Warrior |
|  | OWASP top 10 Security Vulnerabilities     1. SQL Injection |
|  | Ans: SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. |
|  | To avoid SQL Injection. |
|  | 1. Stop writing dynamic queries |
|  | 2. Prevent user supplied input which contains malicious SQL that affecting the logic of executing queries. |
|  | 3. Below are the technique which can used to prevent SQL injection of any database |
|  | There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well |
|  |  |
|  | Primary Defenses: |
|  | • Option 1: Use of Prepared Statements (with Parameterized Queries) |
|  | • Option 2: Use of Stored Procedures |
|  | • Option 3: Whitelist Input Validation |
|  | • Option 4: Escaping All User Supplied Input |
|  | Additional Defenses: |
|  | • Also: Enforcing Least Privilege |
|  | • Also: Performing Whitelist Input Validation as a Secondary Defense |
|  |  |
|  | Exa: SQL injection flaws typically look like this: |
|  | The following (Java) example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database. The unvalidated "customerName" parameter that is simply appended to the query allows an attacker to inject any SQL code they want. Unfortunately, this method for accessing databases is all too common |
|  | String query = "SELECT account\_balance FROM user\_data WHERE user\_name = " |
|  | + request.getParameter("customerName"); |
|  | try { |
|  | Statement statement = connection.createStatement( ... ); |
|  | ResultSet results = statement.executeQuery( query ); |
|  | } |
|  |  |
|  |  |
|  | Primary Defenses |
|  | Defense Option 1: Prepared Statements (with Parameterized Queries) |
|  | 1. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. |
|  | 2. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied. |
|  | 3. Prepared statements ensure that an attacker is not able to change the intent of a query |
|  | 4. if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1 |
|  | 5. Java EE – use PreparedStatement() with bind variables |
|  |  |
|  | 1. PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)"); |
|  | 2. stmt.setInt(1,101);//1 specifies the first parameter in the query |
|  | 3. stmt.setString(2,"Ratan"); |
|  | 4. PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?"); |
|  | 5. stmt.setString(1,"Sonoo");//1 specifies the first parameter in the query i.e. name |
|  | 6. stmt.setInt(2,101); |
|  |  |
|  | Safe Java Prepared Statement Example |
|  | The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query. |
|  | // This should REALLY be validated too |
|  | String custname = request.getParameter("customerName"); |
|  | // Perform input validation to detect attacks |
|  | String query = "SELECT account\_balance FROM user\_data WHERE user\_name = ? "; |
|  | PreparedStatement pstmt = connection.prepareStatement( query ); |
|  | pstmt.setString( 1, custname); |
|  | ResultSet results = pstmt.executeQuery( ); |
|  |  |
|  |  |
|  | Hibernate Query Language (HQL) have the same type of injection problems (which we call HQL Injection). HQL supports parameterized queries as well, so we can avoid this problem: |
|  | //First is an unsafe HQL Statement |
|  |  |
|  | Hibernate Query Language (HQL) Prepared Statement (Named Parameters) Examples |
|  | Query unsafeHQLQuery |
|  | = session.createQuery("from Inventory where productID='"+userSuppliedParameter+"'"); |
|  | //Here is a safe version of the same query using named parameters |
|  | Query safeHQLQuery = session.createQuery("from Inventory where productID=:productid"); |
|  | safeHQLQuery.setParameter("productid", userSuppliedParameter); |
|  |  |
|  |  |
|  | Defense Option 2: Stored Procedures |
|  | 1. Stored procedures are not always safe from SQL injection. |
|  | 2. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself. |
|  | 3. and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you |
|  | Note: 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. However, it can be done, but should be avoided. |
|  | If it can't be avoided, the stored procedure must use input validation. |
|  |  |
|  | Safe Java Stored Procedure Example |
|  | The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query |
|  | // This should REALLY be validated |
|  | String custname = request.getParameter("customerName"); |
|  | try { |
|  | CallableStatement cs = connection.prepareCall("{call sp\_getAccountBalance(?)}"); |
|  | cs.setString(1, custname); |
|  | ResultSet results = cs.executeQuery(); |
|  | // … result set handling |
|  | } catch (SQLException se) { |
|  | // … logging and error handling |
|  | } |
|  |  |
|  |  |
|  | Defense Option 3: Whitelist Input Validation |
|  | Various parts of SQL queries aren't legal locations for the use of bind variables, such as the names of tables or columns, and the sort order indicator (ASC or DESC). In such situations, input validation or query redesign is the most appropriate defense. For the names of tables or columns, ideally those values come from the code, and not from user parameters. |
|  | But if user parameter values are used for targeting different table names and column names, then the parameter values should be mapped to the legal/expected table or column names to make sure unvalidated user input doesn't end up in the query. |
|  |  |
|  | String tableName; |
|  | switch(PARAM): |
|  | case "Value1": tableName = "fooTable"; |
|  | break; |
|  | case "Value2": tableName = "barTable"; |
|  | break; |
|  | ... |
|  | default : throw new InputValidationException("unexpected value provided" |
|  | + " for table name"); |
|  |  |
|  | 4. For something simple like a sort order, it would be best if the user supplied input is converted to a boolean, and then that boolean is used to select the safe value to append to the query. This is a very standard need in dynamic query creation. |
|  | 5. public String someMethod(boolean sortOrder) { |
|  | 6. String SQLquery = "some SQL ... order by Salary " + (sortOrder ? "ASC" : "DESC"); |
|  |  |
|  |  |
|  | Defense Option 4: Escaping All User-Supplied Input |
|  | Database specific |
|  |  |
|  |  |
|  | Additional Defenses |
|  |  |
|  | Least Privilege |
|  | 1.To minimize the potential damage of a successful SQL injection attack, you should minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts. |
|  | 2. If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts. |
|  |  |
|  |  |
|  | Whitelist Input Validation |
|  | input validation can also be a secondary defense used to detect unauthorized input before it is passed to the SQL query |
|  |  |
|  |  |
|  | 1. Now the app used parameterized queries. Parametrized queries force first to write the complete SQL code, and then pass each parameter as per requirement. It helps in distinguishing between code and data regardless of what the user has supplied and prevent SQL injection. |
|  | 2. The application uses safe eq() operator. This operator ensures that the user inputs are bound securely to variables in the generated queries. It is recommended to avoid the where operator and use the safer alternatives provided by the application programming interface. |
|  | 3. The application does not validate the path and the filename format. So the attacker can for a malicious construct to view confidential user files. |
|  | 4. The application is not relying on any input to decide for a filename and storage location in the filesystem. The file ID parameter value checked for its presence in the database and to determine if the user has permission to view the file. |
|  |  |
|  | 1. Injection Flaws-OS Command Injection |
|  | 1. Command injection (or OS Command Injection) is a type of injection where the software, that constructs a system command using externally influenced input, does not correctly neutralizes the input from special elements that can modify the initially intended command. |
|  | 2. For example, if the supplied value is: |
|  | 3. calc |
|  | 4. when typed in a Windows command prompt, the application Calculator is displayed. |
|  | 5. However, if the supplied value has been tempered with, and now it is: |
|  | 6. calc & echo "test" |
|  | 7. When execute, it changes the meaning of the initial intended value. |
|  | 8. Now, both the Calculator application and the value test are displayed |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | Primary Defenses |
|  | Defense Option 1: Avoid calling OS commands directly |
|  | The primary defense is to avoid calling OS commands directly. Built-in library functions are a very good alternative to OS Commands, and they cannot be manipulated to perform tasks other than those it is intended to do. |
|  | For example use mkdir() instead of system("mkdir /dir\_name"). |
|  | If there are available libraries or APIs for the language you used, this is the preferred method. |
|  |  |
|  | Defense option 2: Escape values added to OS commands specific to each OS |
|  |  |
|  | Defense option 3: Parameterization in conjunction with Input Validation |
|  |  |
|  | Layer 1 |
|  | Parameterization: If available, use structured mechanisms that automatically enforce the separation between data and command. These mechanisms can help to provide the relevant quoting, encoding |
|  |  |
|  | Layer 2 |
|  | Input validation: The values for commands and the relevant arguments should be both validated. There are different degrees of validation for the actual command and its arguments: |
|  | • When it comes to the commands used, these must be validated against a whitelist of allowed commands. |
|  | • In regards to the arguments used for these commands, they should be validated using the following options: |
|  | o Positive or whitelist input validation: Where are the arguments allowed explicitly defined. |
|  | o White list Regular Expression: Where is explicitly defined a whitelist of good characters allowed and the maximum length of the string. Ensure that metacharacters like ones specified in Note A and white-spaces are not part of the Regular Expression. For example, the following regular expression only allows lowercase letters and numbers, and does not contain metacharacters. The length is also being limited to 3-10 characters: ^[a-z0-9]{3,10}$ |
|  | • Note A: |
|  | • & | ; $ > < ` \ ! |
|  |  |
|  | Additional Defenses |
|  | On top of primary defences, parameterizations and input validation, we also recommend adopting all of these additional defenses in order to provide defense in depth. |
|  | These additional defenses are: |
|  | • Applications should run using the lowest privileges that are required to accomplish the necessary tasks. |
|  | • If possible, create isolated accounts with limited privileges that are only used for a single task. |
|  |  |
|  | Code examples |
|  | In Java, use ProcessBuilder and the command must be separated from its arguments. |
|  | In Java, use ProcessBuilder and the command must be separated from its arguments. |
|  | Note about the Java's Runtime.exec method behavior: |
|  | There are many sites that will tell you that Java's Runtime.exec is exactly the same as C's system function. This is not true. Both allow you to invoke a new program/process. |
|  | However, C's system function passes its arguments to the shell (/bin/sh) to be parsed, whereas Runtime.exec tries to split the string into an array of words, then executes the first word in the array with the rest of the words as parameters. |
|  | Runtime.exec does NOT try to invoke the shell at any point and do not support shell metacharacters |
|  |  |
|  | The key difference is that much of the functionality provided by the shell that could be used for mischief (chaining commands using &, &&, |, ||, etc, redirecting input and output) would simply end up as a parameter being passed to the first command, and likely causing a syntax error, or being thrown out as an invalid parameter. |
|  |  |
|  |  |
|  | Incorrect usage: |
|  | ProcessBuilder b = new ProcessBuilder("C:\DoStuff.exe -arg1 -arg2"); |
|  | In this example, the command together with the arguments are passed as a one string, making easy to manipulate that expression and inject malicious strings. |
|  |  |
|  | Correct Usage: |
|  | Here is an example that starts a process with a modified working directory. The command and each of the arguments are passed separately. This make it easy to validated each term and reduces the risk to insert malicious strings. |
|  | ProcessBuilder pb = new ProcessBuilder("TrustedCmd", "TrustedArg1", "TrustedArg2"); |
|  |  |
|  | Map<String, String> env = pb.environment(); |
|  |  |
|  | pb.directory(new File("TrustedDir")); |
|  |  |
|  | Process p = pb.start(); |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | 1. A dangerous OS invocation construct is used to perform the file removal from the file system. An attacker could tamper with the input parameters to modify the general structure of the command and execute arbitrary instructions in servers. |
|  | 2. The app uses the Java API to perform file operations against the server's filesystem. This way, this ensuring no commands outside of instructions can be executed by sending a malicious request to servers. |
|  | 3. The application mixes Hibernate Query Language (HQL) - queries and user data. It increases the likelihood of SQL-injection.. |
|  | 4. The prepared parameter is added to the query. The email variable is properly handled and SQL-injection prevented. Using prepared statements is one of the methods to write dynamic queries but a faster, safer, and easier way. Parametrized Queries force to first write the complete SQL code, and then pass each parameter as per requirement. It helps in distinguishing between code and data regardless of what user input has supplied to SQL code. |
|  | 5. String formatting to generate the query does not ensure that string values will be safe, because this does nothing except inserting the value in the string. So this is just the same as using the inputs directly in the SQL queries, and it increases the likelihood of SQL-injection |
|  | 6. Now a prepared parameter is added to query. In this way, the statement is escaped correctly and ensures that the information is used only as data. It helps prevent SQL injection. |
|  | 7. A dangerous OS invocation construct is used to perform the file removal from the file system. An attacker could tamper with the input parameters to modify the general structure of the command and execute arbitrary instructions in servers. |
|  |  |
|  | 1. Broken Authentication |
|  | Authentication General Guidelines |
|  | User IDs |
|  | Make sure your usernames/user IDs are case-insensitive. User 'smith' and user 'Smith' should be the same user. Usernames should also be unique. For high-security applications, usernames could be assigned and secret instead of user-defined public data. |
|  |  |
|  | Authentication and Error Messages |
|  |  |
|  | The application does not provide an attacker with any information used to run an email enumeration. It recommended that carefully consider error messages and response times from the server. |
|  | Using any of the authentication mechanisms (login, password reset or password recovery), an application must respond with a generic error message regardless of whether: |
|  | • The user ID or password was incorrect. |
|  | • The account does not exist. |
|  | • The account is locked or disabled. |
|  | The account registration feature should also be taken into consideration, and the same approach of generic error message can be applied regarding the case in which the user exists. |
|  | The objective is to prevent the creation of a discrepancy factor, allowing an attacker to mount a user enumeration action against the application. |
|  | It is interesting to note that the business logic itself can bring a discrepancy factor related to the processing time taken. Indeed, depending on the implementation, the processing time can be significantly different according to the case (success vs failure) allowing an attacker to mount a time-based attack (delta of some seconds for example). |
|  | Example using pseudo-code for a login feature: |
|  | • First implementation using the "quick exit" approach |
|  | • IF USER\_EXISTS(username) THEN |
|  | • password\_hash=HASH(password) |
|  | • IS\_VALID=LOOKUP\_CREDENTIALS\_IN\_STORE(username, password\_hash) |
|  | • IF NOT IS\_VALID THEN |
|  | • RETURN Error("Invalid Username or Password!") |
|  | • ENDIF |
|  | • ELSE |
|  | • RETURN Error("Invalid Username or Password!") |
|  | • ENDIF |
|  | It can be clearly seen that if the user doesn't exist, the application will directly throw an error. Otherwise, when the user exists and the password doesn't, it is apparent that there will be more processing before the application errors out. In return, the response time will be different for the same error, allowing the attacker to differentiate between a wrong username and a wrong password. |
|  | password\_hash=HASH(password) |
|  | IS\_VALID=LOOKUP\_CREDENTIALS\_IN\_STORE(username, password\_hash) |
|  | IF NOT IS\_VALID THEN |
|  | RETURN Error("Invalid Username or Password!") |
|  | ENDIF |
|  | This code will go through the same process no matter what the user or the password is, allowing the application to return in approximately the same response time. |
|  |  |
|  | Regarding the user enumeration itself, protection against brute-force attack are also effective because they prevent an attacker from applying the enumeration at scale. Usage of CAPTCHA can be applied on a feature for which a generic error message cannot be returned because the user experience must be preserved. |
|  |  |
|  |  |
|  | Incorrect and correct response examples |
|  | Login |
|  | Incorrect response examples: |
|  | • "Login for User foo: invalid password." |
|  | • "Login failed, invalid user ID." |
|  | • "Login failed; account disabled." |
|  | • "Login failed; this user is not active." |
|  | Correct response example: |
|  | • "Login failed; Invalid user ID or password." |
|  | Password recovery |
|  | Incorrect response examples: |
|  | • "We just sent you a password reset link." |
|  | • "This email address doesn't exist in our database." |
|  | Correct response example: |
|  | • "If that email address is in our database, we will send you an email to reset your password." |
|  |  |
|  | Account creation |
|  | Incorrect response examples: |
|  | • "This user ID is already in use." |
|  | • "Welcome! You have signed up successfully." |
|  | Correct response example: |
|  | • "A link to activate your account has been emailed to the address provided." |
|  | 1. An adversary can determine which accounts are registered when providing the user with specific feedback about the authentication process. |
|  | • It's recommended to provide vague feedback to the user about the authentication process. It's important to keep the response time consistent in all cases, this way an adversary won't be able to use a timing attack to determine existing accounts. |
|  | • The application is setting a remember-me token before confirming if the user's credentials are valid. In addition to this, the token provided has no expiry date. |
|  | • The application is limiting the time the token considered valid and will only create a token if the credentials have been correctly validated first. However, the token is never invalidated nor removed when the user logs out. That means if an attacker can hijack a user's token, it can be used to access the system even after the user has logged out |
|  |  |
|  | 1. Sensitive Data Exposure |
|  | Cryptographic\_Storage:Storage Management |
|  |  |
|  | Sensitive Information stored in plaintext. Thus, if an attacker gains access to the database, then it can easily read user data. |
|  |  |
|  | An Advanced Encryption Standard (AES) algorithm in Galois/Counter Mode (GCM) is used to perform the encryption of sensitive data. GCM mode has the benefit of providing authenticity (integrity) in addition to confidentiality. Encoding sensitive user data before storing will successfully hide the data from any attacker that can not access the encryption keys. |
|  |  |
|  | Username Enumeration |
|  | { |
|  | throw new UserAuthenticationException("Username is not registered."); |
|  | } |
|  |  |
|  | if (!isValidPassword) { |
|  | throw new UserAuthenticationException("Wrong password for Username "+username); |
|  | } |

1. Passwords are stored in the database in plain text form. It means that all passwords are easily available to the attacker in case of a database breach.
2. The application uses an Argon2 hashing algorithm to store passwords securely. Argon2 is particularly resistant to ranking tradeoff attacks, making it much more challenging to optimize its calculation on FPGAs, so the attacker is required to use more computational resources for attacks.
3. Storing a password in clear text in the database may result in a system compromise. It means that anyone with access to a database can easily see passwords for all the users.
4. The application uses the Argon2 algorithm to store user’s passwords more securely. Argon2 is particularly resistant to ranking tradeoff attacks, making it much more difficult to optimize on FPGAs cheaply, so the attacker must use more computational resources.
5. XML\_External\_Entity

XML eXternal Entity injection (XXE), which is now part of the [OWASP Top 10](https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A4-XML_External_Entities_%28XXE%29) via the point **A4**, is a type of attack against an application that parses XML input.

XXE issue is referenced under the ID [611](https://cwe.mitre.org/data/definitions/611.html) in the [Common Weakness Enumeration](https://cwe.mitre.org/index.html) referential.

This attack occurs when untrusted 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](https://owasp.org/www-community/attacks/Server_Side_Request_Forgery) (SSRF), port scanning from the perspective of the machine where the parser is located, and other system impacts. The following guide provides concise information to prevent this vulnerability.

**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 [DTD](https://www.w3schools.com/xml/xml_dtd.asp)s also makes the parser secure against denial of services (DOS) attacks such as [Billion Laughs](https://en.wikipedia.org/wiki/Billion_laughs_attack). 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.

Java applications using XML libraries are particularly vulnerable to XXE because the default settings for most Java XML parsers is to have XXE enabled. To use these parsers safely, you have to explicitly disable XXE in the parser you use. The following describes how to disable XXE in the most commonly used XML parsers for Java.

## JAXP DocumentBuilderFactory, SAXParserFactory and DOM4J

DocumentBuilderFactory, SAXParserFactory and DOM4J XML Parsers can be configured using the same techniques to protect them against XXE.

Only the DocumentBuilderFactory example is presented here. The JAXP DocumentBuilderFactory [setFeature](https://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/DocumentBuilderFactory.html" \l "setFeature(java.lang.String,%20boolean" \t "_blank)) method allows a developer to control which implementation-specific XML processor features are enabled or disabled.

The features can either be set on the factory or the underlying XMLReader [setFeature](https://docs.oracle.com/javase/7/docs/api/org/xml/sax/XMLReader.html" \l "setFeature%28java.lang.String,%20boolean%29" \t "_blank) method.

Each XML processor implementation has its own features that govern how DTDs and external entities are processed.

## JAXB Unmarshaller

Since a javax.xml.bind.Unmarshaller parses XML and does not support any flags for disabling XXE, it's imperative to parse the untrusted XML through a configurable secure parser first, generate a source object as a result, and pass the source object to the Unmarshaller

//Disable XXE

SAXParserFactory spf = SAXParserFactory.newInstance();

spf.setFeature("http://xml.org/sax/features/external-general-entities", false);

spf.setFeature("http://xml.org/sax/features/external-parameter-entities", false);

spf.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);

//Do unmarshall operation

Source xmlSource = new SAXSource(spf.newSAXParser().getXMLReader(),

new InputSource(new StringReader(xml)));

JAXBContext jc = JAXBContext.newInstance(Object.class);

Unmarshaller um = jc.createUnmarshaller();

um.unmarshal(xmlSource);

## SAXBuilder

To protect a Java org.jdom2.input.SAXBuilder from XXE, do this:

SAXBuilder builder = new SAXBuilder();

builder.setFeature("http://apache.org/xml/features/disallow-doctype-decl",true);

builder.setFeature("http://xml.org/sax/features/external-general-entities", false);

builder.setFeature("http://xml.org/sax/features/external-parameter-entities", false);

Document doc = builder.build(new File(fileName));

## SAXReader

To protect a Java org.dom4j.io.SAXReader from XXE, do this:

saxReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

saxReader.setFeature("http://xml.org/sax/features/external-general-entities", false);

saxReader.setFeature("http://xml.org/sax/features/external-parameter-entities", false);

## XMLReader

To protect a Java org.xml.sax.XMLReader from XXE, do this:

XMLReader reader = XMLReaderFactory.createXMLReader();

reader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);

// This may not be strictly required as DTDs shouldn't be allowed at all, per previous line.

reader.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);

reader.setFeature("http://xml.org/sax/features/external-general-entities", false);

reader.setFeature("http://xml.org/sax/features/external-parameter-entities", false);

**Note: Use of the following XMLConstants requires JAXP 1.5, which was added to Java in 7u40 and Java 8:**

* javax.xml.XMLConstants.ACCESS\_EXTERNAL\_DTD
* javax.xml.XMLConstants.ACCESS\_EXTERNAL\_SCHEMA
* javax.xml.XMLConstants.ACCESS\_EXTERNAL\_STYLESHEET

## Validator

To protect a javax.xml.validation.Validator from XXE, do this:

SchemaFactory factory = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");

Schema schema = factory.newSchema();

Validator validator = schema.newValidator();

validator.setProperty(XMLConstants.ACCESS\_EXTERNAL\_DTD, "");

validator.setProperty(XMLConstants.ACCESS\_EXTERNAL\_SCHEMA, "");

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);

// 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);

// And, per Timothy Morgan: "If for some reason support for inline DOCTYPEs are a requirement, then

// ensure the entity settings are disabled (as shown above) and beware that SSRF attacks

// (http://cwe.mitre.org/data/definitions/918.html) and denial

// of service attacks (such as billion laughs or decompression bombs via "jar:") are a risk."

// remaining parser logic

...

} catch (ParserConfigurationException e) {

// This should catch a failed setFeature feature

logger.info("ParserConfigurationException was thrown. The feature '" + FEATURE

+ "' is probably not supported by your XML processor.");

...

} catch (SAXException e) {

// On Apache, this should be thrown when disallowing DOCTYPE

logger.warning("A DOCTYPE was passed into the XML document");

...

} catch (IOException e) {

// XXE that points to a file that doesn't exist

logger.error("IOException occurred, XXE may still possible: " + e.getMessage());

...

}

// Load XML file or stream using a XXE agnostic configured parser...

DocumentBuilder safebuilder = dbf.newDocumentBuilder();

[Xerces 1](https://xerces.apache.org/xerces-j/) [Features](https://xerces.apache.org/xerces-j/features.html):

* Do not include external entities by setting [this feature](https://xerces.apache.org/xerces-j/features.html#external-general-entities) to false.
* Do not include parameter entities by setting [this feature](https://xerces.apache.org/xerces-j/features.html#external-parameter-entities) to false.
* Do not include external DTDs by setting [this feature](https://xerces.apache.org/xerces-j/features.html#load-external-dtd) to false.

[Xerces 2](https://xerces.apache.org/xerces2-j/) [Features](https://xerces.apache.org/xerces2-j/features.html):

* Disallow an inline DTD by setting [this feature](https://xerces.apache.org/xerces2-j/features.html#disallow-doctype-decl) to true.
* Do not include external entities by setting [this feature](https://xerces.apache.org/xerces2-j/features.html#external-general-entities) to false.
* Do not include parameter entities by setting [this feature](https://xerces.apache.org/xerces2-j/features.html#external-parameter-entities) to false.
* Do not include external DTDs by setting [this feature](https://xerces.apache.org/xerces-j/features.html#load-external-dtd) to false.

1. The application uses a SAXParser to parse Extensible Markup Language (XML) files. This parser allows Document Type Definitions (DTD) which makes the parser vulnerable to XML External Entity (XXE) attacks.
2. The external-general-entities, external-parameter-entities, setXIncludeAware and load-external-dtd features are set to false. It disables External Entities and Document Type Definitions (DTDs) and protects the application from Extensible Markup Language (XML) - related attacks.
3. The application is parsing Extensible Markup Language (XML) data without restricting the origin or Document Type, making it vulnerable to XML External Entity (XXE) attacks. These can lead to information exposure, denial-of-service (DoS) and other problems.
4. The external-general-entities, external-parameter-entities, setXIncludeAware and load-external-dtd features are set to false. That protects the application from various attacks which can be executed at the Extensible Markup Language (XML) parsing phase.
5. Extensible Markup Language (XML) files for updating the book shop catalog usual generated externally. If XML External Entity (XXE) is used in XML, then an attacker could abuse privileges using XML objects. For example, an attacker may request the execution of an individual file or even download the content of the file.
6. Almost all XML External Entity (XXE) vulnerabilities arise because the applications use an Extensible Markup Language (XML) parsing library that supports potentially unsafe functions. The easiest and most effective way to prevent XXE attacks is to disable these features.
7. Broken Access Control

Access control involves the use of several protection mechanisms such as:

* Authentication (proving the identity of an actor)
* Authorization (ensuring that a given actor can access a resource), and
* Accountability (tracking of activities that were performed)

## What is Access Control / Authorization?

1. Authorization is the process where requests to access a particular resource should be granted or denied. It should be noted that authorization is not equivalent to authentication.
2. Authentication is providing and validating identity. Authorization includes the execution rules that determine what functionality and data the user (or Principal) may access, ensuring the proper allocation of access rights after authentication is successful.
3. Web applications need access controls to allow users (with varying privileges) to use the application. They also need administrators to manage the applications access control rules and the granting of permissions or entitlements to users and other entities. Various access control design methodologies are available. To choose the most appropriate one, a risk assessment needs to be performed to identify threats and vulnerabilities specific to your application, so that the proper access control methodology is appropriate for your application

**Role-Based Access Control (RBAC)**

In Role-Based Access Control (RBAC), access decisions are based on an individual's roles and responsibilities within the organization or user base.

The process of defining roles is usually based on analyzing the fundamental goals and structure of an organization and is usually linked to the security policy. For instance, in a medical organization, the different roles of users may include those such as a doctor, nurse, attendant, patients, etc. Obviously, these members require different levels of access in order to perform their functions, but also the types of web transactions and their allowed context vary greatly depending on the security policy and any relevant regulations (HIPAA, Gramm-Leach-Bliley, etc.).

An RBAC access control framework should provide web application security administrators with the ability to determine who can perform what actions, when, from where, in what order, and in some cases under what relational circumstances.

The advantages of using this methodology are:

* Roles are assigned based on organizational structure with emphasis on the organizational security policy
* Easy to use
* Easy to administer
* Built into most frameworks
* Aligns with security principles like segregation of duties and least privileges

Problems that can be encountered while using this methodology:

* Documentation of the roles and accesses has to be maintained stringently.
* Multi-tenancy can not be implemented effectively unless there is a way to associate the roles with multi-tenancy capability requirements, e.g. OU in Active Directory
* There is a tendency for scope creep to happen, e.g. more accesses and privileges can be given than intended for. Or a user might be included in two roles if proper access reviews and subsequent revocation is not performed.
* Does not support data-based access control

The areas of caution while using RBAC are:

* Roles must be only be transferred or delegated using strict sign-offs and procedures.
* When a user changes his role to another one, the administrator must make sure that the earlier access is revoked such that at any given point of time, a user is assigned to only those roles on a need to know basis.
* Assurance for RBAC must be carried out using strict access control reviews.

**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.

* 1. The application provides direct access to objects based on user-supplied input. As a result of this vulnerability, attackers can bypass authorization and access resources (for example, database records) in the system directly.
  2. The profile link does not contain the user id parameter. The attacker could not access resources directly anymore. It is always recommended to conduct a secure verification of user rights and avoid direct links where possible.

Missing-function-level-access-control:

1. If the authentication check in sensitive request handlers is insufficient or non-existent the vulnerability can be categorised as Missing Function Level Access Control.
2. An example of this would be an unauthorised user being able to access a URL that contains any sensitive information or exposes functionality intended only for authorized users. Another example of a common type of this vulnerability would be to simply hide a feature from the user, but allowing the request if the user is able to figure out how to conduct it.

## **How to discover**

One way to discover Missing Function Level Access Control is to browse the website while logged in and log all pages visited. The next step is to log out and then revisit all pages. If you get the same result, it is likely that this vulnerability exists. Some proxies made for security testing support this type of analysis by default.

Another way is to simply bruteforce different paths. An example may be /admin, /settings or similar that only an admin should be allowed to visit. If any user can access these, it would be considered a vulnerability. This is also called forced browsing, which, simplified, is to enumerate and access resources that are not referenced by the application, but are still accessible.

Yet another way is to identify user IDs and similar data in requests and simply change them to someone else. Chances are that information about some other user can be received that way, or even the ability to execute actions in their name. That would be similar to the Twitter vulnerability mentioned above in Well-known events.

1. **I**nsecure **D**irect **O**bject **R**eference

**I**nsecure **D**irect **O**bject **R**eference (called **IDOR** from here) occurs when a application exposes a reference to an internal implementation object. Using this way, it reveals the real identifier and format/pattern used of the element in the storage backend side. The most common example of it (although is not limited to this one) is a record identifier in a storage system (database, filesystem and so on).

# Context

IDOR do not bring a direct security issue because, by itself, it reveals only the format/pattern used for the object identifier. IDOR bring, depending on the format/pattern in place, a capacity for the attacker to mount a enumeration attack in order to try to probe access to the associated objects

Enumeration attack can be described in the way in which the attacker build a collection of valid identifiers using the discovered format/pattern and test them against the application.

**For example:**

Imagine an HR application exposing a service accepting employee ID in order to return the employee information and for which the format/pattern of the employee ID is the following:

EMP-00000

EMP-00001

EMP-00002

Based on this, an attacker can build a collection of valid ID from EMP-00000 to EMP-99999.

To be exploited, an IDOR issue must be combined with an [Access Control](https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html) issue because it's the Access Control issue that "allow" the attacker to access to the object for which he have guessed the identifier through is enumeration attack.

# Proposition

The proposal use a hash to replace the direct identifier. This hash is salted with a value defined at application level in order support topology in which the application is deployed in multi-instances mode

Using a hash allow the following properties:

* Do not require to maintain a mapping table (real ID vs front end ID) in user session or application level cache.
* Makes creation of a collection a enumeration values more difficult to achieve because, even if attacker can guess the hash algorithm from the ID size, it cannot reproduce value due to the salt that is not tied to the hidden value.
* public class IDORUtil {
* /\*\*
* \* SALT used for the generation of the HASH of the real item identifier
* \* in order to prevent to forge it on front end side.
* \*/
* private static final String SALT = "[READ\_IT\_FROM\_APP\_CONFIGURATION]";
* /\*\*
* \* Compute a identifier that will be send to the front end and be used as item
* \* unique identifier on client side.
* \*
* \* @param realItemBackendIdentifier Identifier of the item on the backend storage
* \* (real identifier)
* \* @return A string representing the identifier to use
* \* @throws UnsupportedEncodingException If string's byte cannot be obtained
* \* @throws NoSuchAlgorithmException If the hashing algorithm used is not
* \* supported is not available
* \*/
* public static String computeFrontEndIdentifier(String realItemBackendIdentifier)
* throws NoSuchAlgorithmException, UnsupportedEncodingException {
* String frontEndId = null;
* if (realItemBackendIdentifier != null && !realItemBackendIdentifier.trim().isEmpty()) {
* //Prefix the value with the SALT
* String tmp = SALT + realItemBackendIdentifier;
* //Get and configure message digester
* //We use SHA1 here for the following reason even if SHA1 have now potential collision:
* //1. We do not store sensitive information, just technical ID
* //2. We want that the ID stay short but not guessable
* //3. We want that a maximum of backend storage support the algorithm used in order to compute it in selection query/request
* //If your backend storage supports SHA256 so use it instead of SHA1
* MessageDigest digester = MessageDigest.getInstance("sha1");
* //Compute the hash
* byte[] hash = digester.digest(tmp.getBytes("utf-8"));
* //Encode is in HEX
* frontEndId = DatatypeConverter.printHexBinary(hash);
* }
* return frontEndId;
* }
* }
* @RequestMapping(value = "/movies", method = GET, produces = {MediaType.APPLICATION\_JSON\_VALUE})
* public Map<String, String> listAllMovies() {
* Map<String, String> result = new HashMap<>();
* try {
* this.movies.forEach(m -> {
* try {
* //Compute the front end ID fof the current element
* String frontEndId = IDORUtil.computeFrontEndIdentifier(m.getBackendIdentifier());
* //Add the computed ID and the associated item name to the result map
* result.put(frontEndId, m.getName());
* } catch (Exception e) {
* LOGGER.error("Error during ID generation for real ID {}: {}", m.getBackendIdentifier(),
* e.getMessage());
* }
* });
* } catch (Exception e) {
* //Ensure that in case of error no item is returned
* result.clear();
* LOGGER.error("Error during processing", e);
* }
* return result;
* }

@RequestMapping(value = "/movies/{id}", method = GET, produces = {MediaType.APPLICATION\_JSON\_VALUE})

public Movie obtainMovieName(@PathVariable("id") String id) {

//Search for the wanted movie information using Front End Identifier

Optional<Movie> movie = this.movies.stream().filter(m -> {

boolean match;

try {

//Compute the front end ID for the current element

String frontEndId = IDORUtil.computeFrontEndIdentifier(m.getBackendIdentifier());

//Check if the computed ID match the one provided

match = frontEndId.equals(id);

} catch (Exception e) {

//Ensure that in case of error no item is returned

match = false;

LOGGER.error("Error during processing", e);

}

return match;

}).findFirst();

//We have marked the Backend Identifier class field as excluded

//from the serialization

//So we can send the object to front end through the serializer

return movie.get();

}

public class Movie {

/\*\*

\* We indicate to serializer that this field must never be serialized

\*

\* @see "https://fasterxml.github.io/jackson-annotations/javadoc/2.5/com/fasterxml/

\* jackson/annotation/JsonIgnore.html"

\*/

@JsonIgnore

private String backendIdentifier;

...

}

* 1. The upload functionality is accessible for any authenticated user. Thus, an adversary could easily abuse the upload functionality.
  2. The administrator rights are checked, but only for the GET method. The user can send a POST request without using the application interface and thus gain access to the application function.
  3. An authorization check added to the file upload function. But the web.xml file is not configured correctly. As a result, the flawed definition will take priority and will allow regular users to access that functionality.
  4. If a user tries to access manager functions, then the application verifies that the user has manager permissions. An application must first verify function level access rights before making that functionality visible in the UI.
  5. The application provides direct access to user information based on the user-supplied input. That means attackers can bypass authorization and access resources in the system directly.
  6. The application does not accept the userId parameter, and the profile link does not contain it. Therefore an attacker cannot pick up the userId parameter and access user's personal information

# 8. Security Misconfiguration

Improper server or web application configuration leading to various flaws:

* Debugging enabled.
* Incorrect folder permissions.
* Using default accounts or passwords.
* Setup/Configuration pages enabled.

# **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.

# 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.  
\* 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?

# 

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](https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.html)).  
\* Disable caching for response that contain sensitive data.  
\* Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as [Argon2](https://www.cryptolux.org/index.php/Argon2), [scrypt](https://wikipedia.org/wiki/Scrypt), [bcrypt](https://wikipedia.org/wiki/Bcrypt) or [PBKDF2](https://wikipedia.org/wiki/PBKDF2).  
\* Verify independently the effectiveness of configuration and settings.

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.

# Detailed error messages are displayed for debugging purposes. If the application encounters an error, it shows an error page with that specific message. Using this, an attacker can collect debug information and use it to access the system.

# A debug information is not provided anymore, so the attacker cannot use it to access the system. It is strongly recommended to remove the debug functionality before deploying the application to the production environment.

# The application has hardcoded configuration settings. If a source code leak occurs, sensitive information will be available to anyone who can get access to the code base.

# The settings are not stored in the codebase anymore. They are stored as environment variables, so if there is a source code leak, an attacker will not be able to get the parameters. Environment variables can be defined without changing any deployment artifacts. But the most secure to store confidential data like keys and passwords is to use special protected key storage.

# The application is using hardcoded values as default values for the database. As a result, any leaks of the source code would compromise the system fundamentally, requiring recompiling to update these values in live deployments.

# The configuration moved to a separate property file, but it does not solve the problem, and important data is still hardcoded within the program itself.

# The settings stored not in the code base but environment variables, so if there is a leak, an attacker will not be able to get the parameters. This way of storing settings is quite safe and convenient since environment variables supported by almost every operating system, and they can define without any changes in deployment artifacts.

# Cross\_Site\_Scripting

## Bonus Rule #2: Implement Content Security Policy

# There is another good complex solution to mitigate the impact of an XSS flaw called Content Security Policy. It's a browser side mechanism which allows you to create source whitelists for client side resources of your web application, e.g. JavaScript, CSS, images, etc. CSP via special HTTP header instructs the browser to only execute or render resources from those sources.

For example this CSP:

Content-Security-Policy: default-src: 'self'; script-src: 'self' static.domain.tld

Will instruct web browser to load all resources only from the page's origin and JavaScript source code files additionally from static.domain.tld

# An application can be exposed to DOM-Based Cross-Site Scripting if it is using html() to render data on the client-side. An adversary can use this vulnerability to malicious code such as scripts or links.

# Using jQuery the text() method instead of the html() method ensures that malicious code won't be executed. It will sanitize all data before rendering on the client thus succesfully preventing Dom-Based Cross-Site Scripting (XSS).

# Malicious code is stored on a web server (for example, in a database). An attack executes every time a user opens a page with that malicious data previously saved on the server.

# JavaServer Pages Standard Tag Library (JSTL) library tag is used for sanitization. It allows displaying unescaped text through its c:out core tag. However, the use of a escapeXml="false" in the c:out core tag is not secure as it does not properly escape Hypertext Markup Language/Extensible Markup Language. Consequently, this allows for Cross-Site Scripting (XSS) attacks.

# The application uses a c:out core tag. Thus, the JavaServer Pages Standard Tag Library (JSTL) will automatically take care of properly escaping text for Hypertext Markup Language (HTML) output. Also, a Content-Security-Policy Hypertext Transfer Protocol (HTTP) header allows creating a whitelist of sources of the trusted content. It instructs the browser to execute resources only from those sources.

# Stored cross-site scripting can arise when the malicious code or script saved on the webserver (for example, in the database) and executed whenever the users will call the appropriate functionality. This way, stored XSS attacks can affect many users. Also, as the script stored on the webserver, it will affect the website for a longer time.

# Now the app used the output filters c:out. The attribute of the escapeXml tag is responsible for displaying special characters on the page as HTML code or as simple characters. By default, escapeXml is true. Also, CSP defines the Content-Security-Policy HTTP header that allows creating a whitelist of sources of trusted content and instructs the browser only to execute resources from those sources.

# Insecure Deserialization

# What is Deserialization?

**Serialization** is the process of turning some object into a data format that can be restored later. People often serialize objects in order to save them to storage, or to send as part of communications.

**Deserialization** is the reverse of that process, taking data structured from some format, and rebuilding it into an object. Today, the most popular data format for serializing data is JSON. Before that, it was XML.

However, many programming languages offer a native capability for serializing objects. These native formats usually offer more features than JSON or XML, including customizability of the serialization process.

Unfortunately, the features of these native deserialization mechanisms can be repurposed for malicious effect when operating on untrusted data. Attacks against deserializers have been found to allow denial-of-service, access control, and remote code execution (RCE) attacks.

# Guidance on Deserializing Objects Safely

## Java

The following techniques are all good for preventing attacks against deserialization against [Java's Serializable format](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html).

Implementation advices:

* In your code, override the ObjectInputStream#resolveClass() method to prevent arbitrary classes from being deserialized. This safe behavior can be wrapped in a library like [SerialKiller](https://github.com/ikkisoft/SerialKiller" \t "_blank).
* Use a safe replacement for the generic readObject() method as seen here. Note that this addresses "[billion laughs](https://en.wikipedia.org/wiki/Billion_laughs_attack)" type attacks by checking input length and number of objects deserialized.

### WhiteBox Review

Be aware of the following Java API uses for potential serialization vulnerability.

1. XMLdecoder with external user defined parameters
2. XStream with fromXML method (xstream version <= v1.46 is vulnerable to the serialization issue)
3. ObjectInputStream with readObject
4. Uses of readObject, readObjectNodData, readResolve or readExternal
5. ObjectInputStream.readUnshared
6. Serializable

### BlackBox Review

If the captured traffic data include the following patterns may suggest that the data was sent in Java serialization streams

* AC ED 00 05 in Hex
* rO0 in Base64
* Content-type header of an HTTP response set to application/x-java-serialized-object

### Prevent Data Leakage and Trusted Field Clobbering

If there are data members of an object that should never be controlled by end users during deserialization or exposed to users during serialization, they should be declared as [the transient keyword](https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serial-arch.html#7231) (section Protecting Sensitive Information)

For a class that defined as Serializable, the sensitive information variable should be declared as private transient.

For example, the class myAccount, the variable 'profit' and 'margin' were declared as transient to avoid to be serialized:

public class myAccount implements Serializable

{

private transient double profit; // declared transient

private transient double margin; // declared transient

### Prevent Deserialization of Domain Objects

Some of your application objects may be forced to implement Serializable due to their hierarchy. To guarantee that your application objects can't be deserialized, a readObject() method should be declared (with a final modifier) which always throws an exception:

private final void readObject(ObjectInputStream in) throws java.io.IOException {

throw new java.io.IOException("Cannot be deserialized");

}

### Harden Your Own java.io.ObjectInputStream

The java.io.ObjectInputStream class is used to deserialize objects. It's possible to harden its behavior by subclassing it. This is the best solution if:

* You can change the code that does the deserialization
* You know what classes you expect to deserialize

The general idea is to override [ObjectInputStream.html#resolveClass()](https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html" \l "resolveClass(java.io.ObjectStreamClass" \t "_blank)) in order to restrict which classes are allowed to be deserialized.

Because this call happens before a readObject() is called, you can be sure that no deserialization activity will occur unless the type is one that you wish to allow

The general idea is to override [ObjectInputStream.html#resolveClass()](https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html" \l "resolveClass(java.io.ObjectStreamClass" \t "_blank)) in order to restrict which classes are allowed to be deserialized.

Because this call happens before a readObject() is called, you can be sure that no deserialization activity will occur unless the type is one that you wish to allow.

A simple example of this shown here, where the the LookAheadObjectInputStream class is guaranteed not to deserialize any other type besides the Bicycle class:

public class LookAheadObjectInputStream extends ObjectInputStream {

public LookAheadObjectInputStream(InputStream inputStream) throws IOException {

super(inputStream);

}

/\*\*

\* Only deserialize instances of our expected Bicycle class

\*/

@Override

protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException {

if (!desc.getName().equals(Bicycle.class.getName())) {

throw new InvalidClassException("Unauthorized deserialization attempt", desc.getName());

}

return super.resolveClass(desc);

}

}

More complete implementations of this approach have been proposed by various community members:

* [NibbleSec](https://github.com/ikkisoft/SerialKiller) - a library that allows whitelisting and blacklisting of classes that are allowed to be deserialized
* [IBM](https://www.ibm.com/developerworks/library/se-lookahead/) - the seminal protection, written years before the most devastating exploitation scenarios were envisioned.
* [Apache Commons IO classes](https://commons.apache.org/proper/commons-io/javadocs/api-2.5/org/apache/commons/io/serialization/ValidatingObjectInputStream.html)

### Harden All java.io.ObjectInputStream Usage with an Agent

To enable these agents, simply add a new JVM parameter:

-javaagent:name-of-agent.jar

# Language-Agnostic Methods for Deserializing Safely

## Using Alternative Data Formats

A great reduction of risk is achieved by avoiding native (de)serialization formats. By switching to a pure data format like JSON or XML

## Only Deserialize Signed Data

If the application knows before deserialization which messages will need to be processed, they could sign them as part of the serialization process. The application could then to choose not to deserialize any message which didn't have an authenticated signature

# The application performs deserialization of untrusted data without input validation. It means that malformed data can be used to abuse the application logic, deny service, or execute arbitrary code when deserialized.

# Data is encrypted and checked by HomeWorkDeserializer.java. However, using cipher mode Cipher.DECRYPT\_MODE in the saveDraft() method, causes the application to throw an exception.

# The application uses readUnshared() method for reads an object from ObjectInputStream. This method is identical to readObject, except it prevents subsequent calls to from returning additional references to the de-serialized instance obtained via this call. This method does not prevent deserialization attacks.

# The serialized object is encrypted before saving and is also decrypted before deserialization using the CipherSecurity.java class. The HomeWorkDeserializer.java class also confirms the same object.

# The application is deserializing untrusted data into arbitrary classes. An attacker can use this to specify a class of their choice, uploading malicious objects or code to the server.

# The application is using a class that extends ObjectInputStream to deserialize the data into the intended class or fail safely. This helps ensure that any data received will match the type expected.

# The application checks an object class before deserialization. However, the check occurs after the deserialization process ends. Therefore, it does not help to prevent any attacks which happen during deserialization.

# The application performs validation of untrusted data using the AttachmentDeserializer class. It ensures that input data will match the type expected.

# 

# 

# Using Components with known vulnerabilities

# A9-Using\_Components\_with\_Known\_Vulnerabilities

# The objective of thevulnerabilities is to provide a proposal of approach regarding the handling of vulnerable third-party dependencies when they are detected, and this, depending on different situation.

# **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:

# Context

Most of the projects use third-party dependencies to delegate handling of different kind of operations, e.g. generation of document in a specific format, HTTP communications, data parsing of a specific format, etc

# It's a good approach because it allows the development team to focus on the real application code supporting the expected business feature. The dependency brings forth an expected downside where the security posture of the real application is now resting on it.

# The application uses a JavaScript library from an untrusted source, which may allow the attacker to execute the malicious scripts.

# The application uses known and trusted JavaScript sources for reCaptcha, which protects the execution of the arbitrary code.

# Modern software is made up of dozens, if not hundreds, of third-party components. Even if the code we write is secure, the other components we are using may not be. The app might be using an out-of-date version of a library that has a known vulnerability.

# Now the application uses the version of jQuery from a trusted CDN, so the app secured against vulnerabilities from the previous versions. A project needs to ensure that all the third-party dependencies implemented are clean of any security issue. If they happen to contain any security issues, the development team needs to be aware of it and apply the required mitigation measures to secure the affected application.

# The application uses old versions of some libraries in which there are vulnerabilities. So one weakness allows us to provide an SSL Factory and not check the hostname. And other vulnerabilities allow an unauthenticated user to perform code execution by sending the maliciously crafted input to the readValue method of the ObjectMapper.

# The application uses a safe version Jackson-Databind library. However, the PostgreSQL library with such a version does not yet exist. It recommended that carefully select libraries.

# The application uses the latest and secured versions of libraries. As a general rule, for library-specific security issues, it is always recommended to use updated versions that have the security patches.

# 

# Insufficient Logging & Monitoring

# Building application logging mechanisms, especially related to security logging

# Many systems enable network device, operating system, web server, mail server and database server logging.

# Application logging should be consistent within the application, consistent across an organization's application portfolio and use industry standards where relevant, so the logged event data can be consumed, correlated, analyzed and managed by a wide variety of systems

**Purpose**

Application logging should be always be included for security events. Application logs are invaluable data for:

* Identifying security incidents
* Monitoring policy violations
* Establishing baselines
* Assisting non-repudiation controls
* Providing information about problems and unusual conditions
* Contributing additional application-specific data for incident investigation which is lacking in other log sources
* Helping defend against vulnerability identification and exploitation through attack detection

Application logging might also be used to record other types of events too such as:

* Security events
* Business process monitoring e.g. sales process abandonment, transactions, connections
* Anti-automation monitoring
* Audit trails e.g. data addition, modification and deletion, data exports
* Performance monitoring e.g. data load time, page timeouts
* Compliance monitoring
* Data for subsequent requests for information e.g. data subject access, freedom of information, litigation, police and other regulatory investigations
* Legally sanctioned interception of data e.g application-layer wire-tapping
* Other business-specific requirements

# Design, implementation and testing

## Event data sources

The application itself has access to a wide range of information events that should be used to generate log entries. Thus, the primary event data source is the application code itself.

The application has the most information about the user (e.g. identity, roles, permissions) and the context of the event (target, action, outcomes), and often this data is not available to either infrastructure devices, or even closely-related applications.

Other sources of information about application usage that could also be considered are:

* Client software e.g. actions on desktop software and mobile devices in local logs or using messaging technologies, JavaScript exception handler via Ajax, web browser such as using Content Security Policy (CSP) reporting mechanism
* Embedded instrumentation code
* Network firewalls
* Network and host intrusion detection systems (NIDS and HIDS)
* Closely-related applications e.g. filters built into web server software, web server URL redirects/rewrites to scripted custom error pages and handlers
* Application firewalls e.g. filters, guards, XML gateways, database firewalls, web application firewalls (WAFs)
* Database applications e.g. automatic audit trails, trigger-based actions
* Reputation monitoring services e.g. uptime or malware monitoring
* Other applications e.g. fraud monitoring, CRM
* Operating system e.g. mobile platform

The degree of confidence in the event information has to be considered when including event data from systems in a different trust zone. Data may be missing, modified, forged, replayed and could be malicious – it must always be treated as untrusted data.

## Where to record event data

Applications commonly write event log data to the file system or a database (SQL or NoSQL). Applications installed on desktops and on mobile devices may use local storage and local databases, as well as sending data to remote storage.

Your selected framework may limit the available choices. All types of applications may send event data to remote systems (instead of or as well as more local storage).

This could be a centralized log collection and management system (e.g. SIEM or SEM) or another application elsewhere. Consider whether the application can simply send its event stream, unbuffered, to stdout, for management by the execution environment.

* When using the file system, it is preferable to use a separate partition than those used by the operating system, other application files and user generated content
  + For file-based logs, apply strict permissions concerning which users can access the directories, and the permissions of files within the directories
  + In web applications, the logs should not be exposed in web-accessible locations, and if done so, should have restricted access and be configured with a plain text MIME type (not HTML)
* When using a database, it is preferable to utilize a separate database account that is only used for writing log data and which has very restrictive database , table, function and command permissions
* Use standard formats over secure protocols to record and send event data, or log files, to other systems e.g. Common Log File System (CLFS) or Common Event Format (CEF) over syslog; standard formats facilitate integration with centralised logging services

Where possible, always log:

* Input validation failures e.g. protocol violations, unacceptable encodings, invalid parameter names and values
* Output validation failures e.g. database record set mismatch, invalid data encoding
* Authentication successes and failures
* Authorization (access control) failures
* Session management failures e.g. cookie session identification value modification
* Application errors and system events e.g. syntax and runtime errors, connectivity problems, performance issues, third party service error messages, file system errors, file upload virus detection, configuration changes
* Application and related systems start-ups and shut-downs, and logging initialization (starting, stopping or pausing)
* Use of higher-risk functionality e.g. network connections, addition or deletion of users, changes to privileges, assigning users to tokens, adding or deleting tokens, use of systems administrative privileges, access by application administrators,all actions by users with administrative privileges, access to payment cardholder data, use of data encrypting keys, key changes, creation and deletion of system-level objects, data import and export including screen-based reports, submission of user-generated content - especially file uploads
* Legal and other opt-ins e.g. permissions for mobile phone capabilities, terms of use, terms & conditions, personal data usage consent, permission to receive marketing communications

Optionally consider if the following events can be logged and whether it is desirable information:

* Sequencing failure
* Excessive use
* Data changes
* Fraud and other criminal activities
* Suspicious, unacceptable or unexpected behavior
* Modifications to configuration
* Application code file and/or memory changes

## Event attributes

Each log entry needs to include sufficient information for the intended subsequent monitoring and analysis. It could be full content data, but is more likely to be an extract or just summary properties.

The application logs must record "when, where, who and what" for each event

The properties for these will be different depending on the architecture, class of application and host system/device, but often include the following:

* When
  + Log date and time (international format)
  + Event date and time - the event time stamp may be different to the time of logging e.g. server logging where the client application is hosted on remote device that is only periodically or intermittently online
  + Interaction identifier Note A
* Where
  + Application identifier e.g. name and version
  + Application address e.g. cluster/host name or server IPv4 or IPv6 address and port number, workstation identity, local device identifier
  + Service e.g. name and protocol
  + Geolocation
  + Window/form/page e.g. entry point URL and HTTP method for a web application, dialogue box name
  + Code location e.g. script name, module name
* Who (human or machine user)
  + Source address e.g. user's device/machine identifier, user's IP address, cell/RF tower ID, mobile telephone number
  + User identity (if authenticated or otherwise known) e.g. user database table primary key value, user name, license number
* What
  + Type of event Note B
  + Severity of event Note B e.g. {0=emergency, 1=alert, ..., 7=debug}, {fatal, error, warning, info, debug, trace}
  + Security relevant event flag (if the logs contain non-security event data too)
  + Description

## Data to exclude

Never log data unless it is legally sanctioned. For example intercepting some communications, monitoring employees, and collecting some data without consent may all be illegal.

Never exclude any events from "known" users such as other internal systems, "trusted" third parties, search engine robots, uptime/process and other remote monitoring systems, pen testers, auditors. However, you may want to include a classification flag for each of these in the recorded data.

The following should not usually be recorded directly in the logs, but instead should be removed, masked, sanitized, hashed or encrypted:

* Application source code
* Session identification values (consider replacing with a hashed value if needed to track session specific events)
* Access tokens
* Sensitive personal data and some forms of personally identifiable information (PII) e.g. health, government identifiers, vulnerable people
* Authentication passwords
* Database connection strings
* Encryption keys and other master secrets
* Bank account or payment card holder data
* Data of a higher security classification than the logging system is allowed to store
* Commercially-sensitive information
* Information it is illegal to collect in the relevant jurisdictions
* Information a user has opted out of collection, or not consented to e.g. use of do not track, or where consent to collect has expired

## Verification

Logging functionality and systems must be included in code review, application testing and security verification processes

* Ensure the logging is working correctly and as specified
* Check events are being classified consistently and the field names, types and lengths are correctly defined to an agreed standard
* Ensure logging is implemented and enabled during application security, fuzz, penetration and performance testing
* Test the mechanisms are not susceptible to injection attacks
* Ensure there are no unwanted side-effects when logging occurs
* Check the effect on the logging mechanisms when external network connectivity is lost (if this is usually required)
* Ensure logging cannot be used to deplete system resources, for example by filling up disk space or exceeding database transaction log space, leading to denial of service
* Test the effect on the application of logging failures such as simulated database connectivity loss, lack of file system space, missing write permissions to the file system, and runtime errors in the logging module itself
* Verify access controls on the event log data
* If log data is utilized in any action against users (e.g. blocking access, account lock-out), ensure this cannot be used to cause denial of service (DoS) of other users
* 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.
  1. Suspicious events are not being logged within the login functionality. This can allow a hacker to perform attacks without being detected
  2. The application monitors and records a range of suspicious activities using an appropriate logger. However, confidential information is also recorded, and this can lead to a sensitive data exposure.
  3. The application registers input data validation failures with a sufficient amount of information. This information will allow to identify suspicious activities and take appropriate actions
  4. Insufficient logging and monitoring allow attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract or destroy data without companies noticing.
  5. With proper monitoring of the password reset functionality, it becomes easier to identify and react to incidents when they occur.
  6. Using insufficient logging and monitoring is the bedrock of nearly every major incident. Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected
  7. Now the app logged password change errors. Server-side input validation failures must be registered with sufficient user context to identify suspicious or malicious accounts, and held for enough time to allow delayed analysis.