The Dos and Don'ts of Secure Coding!

Secure Code Practices of
Top Programming Languages

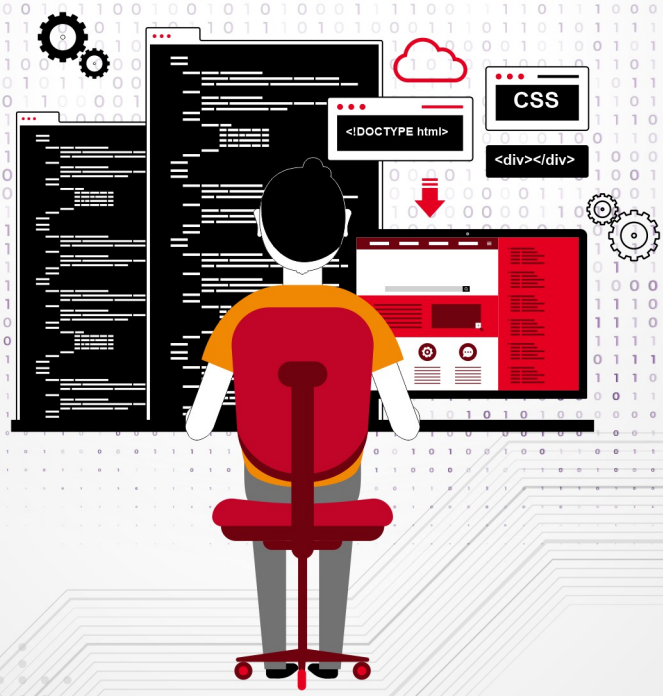


TABLE OF CONTENTS

PART - 1

PART - 2

ABOUT US

Payatu is a research-focused security testing service organization specializing in IoT and embedded products, web, mobile, cloud, infrastructure security assessments, and in-depth technical security training. Payatu's state-of-the-art research, methodologies, and tools ensure the safety of our client's assets. At Payatu, the core belief is in following one's passion. With that thought, Payatu has created a world-class team of researchers and executors who are bending the rules to provide the best security services. A passionate bunch of folks working on the latest and leading-edge security technology. Payatu's impact is delivered every day through its three verticals:

1. Consulting and Training
2. Conferences
3. Products

PART - 1

“Make your application unhackable by learning secure coding."

Murtuja Bharmal
CEO at Payatu - Cybersecurity

CHAPTER 1- INTRODUCTION

It makes, it breaks!

If coding is your great power, then securing it is your big responsibility. Factually, programming is the foundation that makes the application or software; Unfortunately, it can break it if it is weak. Secure code is a set of best practices that applies security consideration and defends the software from attacks.

A professional coder will not put the company's reputation at stake with vulnerable code. However, we know how overwhelming and complicated it can be to practice secure coding.

Which bring us to this ebook. The reason for curating this ebook is to promote secure coding practices. Extracted from our very own Secure Code Wiki.

What is Secure Code Wiki?

Different programming languages bring different challenges regarding secure coding, which drove us to create a Secure Code Wiki, a readily available portal comprising the best Secure Coding Practices of popular programming languages. Do not forget to bookmark it.

The Secure Code Wiki portal comprises the best secure coding practices collected from all around the Internet, plus the practices followed by Payatu Bandits. Bandits have contributed with their time, efforts, knowledge, and experience to make it easy to understand and readily available.

This ebook highlights the Best Secure Code Practices of Top Programming Languages derived from our Secure Code Wiki.

THE AUTHORS

"Individual commitment to a group effort - that is what makes a teamwork, a company work, a society work, a civilization work."

Vince Lombardi

Kumar Ashwin
Security Consultant

Inas Sheikh
Content Specialist

Manmeet Singh
Security Consultant

Dipti Dhandha
Associate Security Consultant

Nikhil Mittal
Associate Security Consultant

Doshan Jinde
Sr. Security Consultant

Suraj Kumar
Sr. Security Consultant

Abhilash Nigam
Security Consultant

Amit Kumar
Security Consultant

Farid Luhar
Security Consultant

Hrushikesh Kakade
Security Consultant

Mayank Arora
Associate Security Consultant

Appar Thusoo
Security Consultant

Nandhakumar
Security Consultant

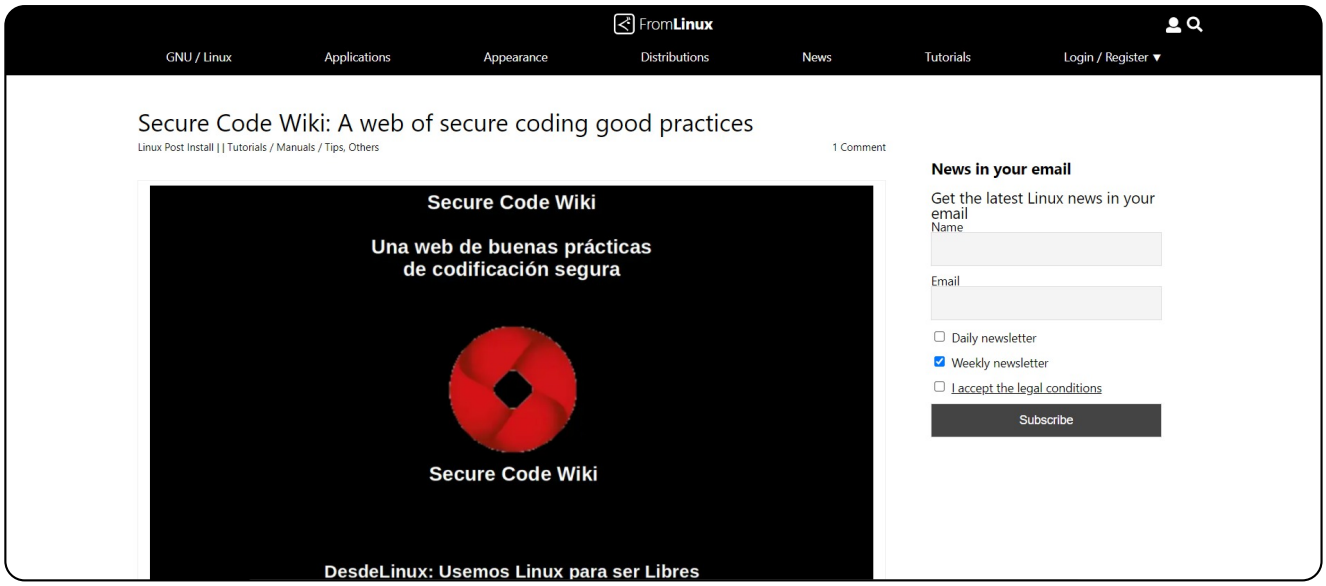
Sourov Ghosh
Security Consultant

Akansha Kesharwani
Senior Security Consultant

To make the Internet a secure place, thanks to those who dedicated their knowledge and skill. What makes it even better are people who share the gift of time and passion to mentor future leaders. This ebook is a result of the dedication and vision of Payatu Bandits. Special thanks to **Abhilash Nigam, Amit Kumar, Dipti Dhandha, Doshan Jinde, Farid Luhar, Hrushikesh Kakade, Kumar Ashwin, Manmeet Singh, Mayank Arora, Nandhakumar, Suraj Kumar,** who strive to grow and help others succeed.

Recognition

We are glad that our Secure Code Wiki is helping future leaders and many organizations to code securely.
Infrequent commentary on IT(sec) by Anant #6 | Revue (anantshri.info)



PART - 2

"Uncover the secrets of hacking applications and eliminate them by implementing secure coding."

Murtuja Bharmal
CEO at Payatu - Cybersecurity

TOP PROGRAMMING LANGUAGE

The ranks of the programming languages are defined by weighting and combining 11 metrics from sources like CareerBuilder, GitHub, Google, Hacker News, IEEE, Reddit, Stack Overflow, and Twitter.

The IEEE spectrum produces the set of ranking by default sets of weight. You can customize your order by adjusting weights and using the "Create Custom Ranking" option.

As per IEEE research, the top 5 programming languages are:

1. Python
2. Java
3. C
4. C++
5. JavaScript

With the help of **Secure Code Wiki**, let's jump into the best practices of Secure Coding in

- Python
- Java
- JavaScript

SECURE CODE PRACTICES

1. Python

A1- Injections

SQL Injection

- Escaping all user-supplied input.
- Performing whitelist input validation as a secondary defence.
- Use of prepared statements (with parameterized queries).
- Use of database ORMs is generally safe as most implementations rely on prepared statements.
- If using SQLAlchemy, upgrade sqlalchemy to version 1.2.18 or higher

When using SQLAlchemy module as ORM

Noncompliant Code

- Flask application

```
stmt = text("SELECT * FROM users where id=%s" % id) # Query is
constructed based on user inputs query =
SQLAlchemy().session.query(User).from_statement(stmt) # Noncompliant user =
query.one()
```

Compliant Code

- Flask application

```
stmt = text("SELECT * FROM users where id=:id")
query
SQLAlchemy().session.query(User).from_statement(stmt).params
(id=d) # Compliant user = query.one()Noncompliant user =
```

When using Django ORM

Noncompliant Code

- Django application

```
from django.db import connection
cursor = connection.cursor()
cursor.execute("SELECT username FROM auth_user WHERE id=%s" % id) #
Noncompliant; Query is constructed based on user inputs
row = cursor.fetchone()
```

Noncompliant Code

- Django application

```
from django.db import connection

cursor = connection.cursor()
cursor.execute("SELECT username FROM auth_user WHERE id=:id", {"id": id}) #
Compliant
row = cursor.fetchone()
return HttpResponse("Hello %s" % row[0])
```

When using native SQLite3 database driver (sqlite3) for python

Noncompliant Code

```
  cursor.execute("SELECT admin FROM users WHERE username = '" + username + "');
  cursor.execute("SELECT admin FROM users WHERE username = '%s' % username);
  cursor.execute("SELECT admin FROM users WHERE username =
'{}'".format(username));
  cursor.execute(f"SELECT admin FROM users WHERE username = '{username}'");
```

Compliant Code

```
  cursor.execute("SELECT admin FROM users WHERE username = ?", (username, ));
  cursor.execute("SELECT admin FROM users WHERE username = :username",
{'username': username});
```

When using native MySQL driver(mysql)/ Postgresql driver (psycopg2) for python

Noncompliant Code

```
  cursor.execute("SELECT admin FROM users WHERE username = '" + username + "');
  cursor.execute("SELECT admin FROM users WHERE username = '%s' % username);
  cursor.execute("SELECT admin FROM users WHERE username =
  '{}'".format(username));
  cursor.execute(f"SELECT admin FROM users WHERE username = '{username}'");
```

Compliant Code

```
  cursor.execute("SELECT admin FROM users WHERE username = %s", (username, ));
  cursor.execute("SELECT admin FROM users WHERE username = %(username)s",
{'username': username});
```

When using OracleDB native driver (cx_Oracle) for python

Noncompliant Code

```
  cursor.execute("SELECT admin FROM users WHERE username = '" + username + "');
  cursor.execute("SELECT admin FROM users WHERE username = '%s' % username);
  cursor.execute("SELECT admin FROM users WHERE username =
  '{}'".format(username));
  cursor.execute(f"SELECT admin FROM users WHERE username = '{username}'");
```

Compliant Code

```
  cursor.execute("SELECT admin FROM users WHERE username = :username",
  {'username': username});
  cursor.execute("SELECT admin FROM users WHERE username = :username",
  (username,));
```

When using MSSQL native driver (pymssql) for python

Noncompliant Code

```
cursor.execute("SELECT admin FROM users WHERE username = '" + username + "');
cursor.execute("SELECT admin FROM users WHERE username = '%s' % username);
cursor.execute("SELECT admin FROM users WHERE username =
'{}'".format(username));
cursor.execute(f"SELECT admin FROM users WHERE username = '{username}'");
```

Compliant Code

```
cursor.execute("SELECT admin FROM users WHERE username = %s", (username, ));
```

NOSQL Injection

When using MongoDB native module (pymongo) in pytho

- When dealing with MongoDB database with its native driver (pymongo) don't let user-supplied input to control $where operator or map_reduce option without sanitizing user input and converting or banning use of special characters like (& | ; $ > < ` \ ! ' “).
- Never parse the user-supplied string to a JSON object and use it do MongoDB operations.

Noncompliant Code

```
client = MongoClient()
db = client.test_database
collection = db.test
from flask import request
import json

@route.app("/getwhere")
def getWhere():
# Get data from fields
condition = json.load("{ name: "+request.form.getvalue('name'))
if condition:
output =collection.find(condition) # Non compliant inserting of user supplied
input
else:
output = ""
return output
```

Compliant Code

```
client = MongoClient()
db = client.test_database
collection = db.test

@route.app("/getwhere")
def getWhere():
# Get data from fields
condition = request.form.getvalue('name')
if condition:
output = collection.find({"name": condition } # compliant
else:
output = ""

return output
```

OS Command Injection

- Avoid calling OS commands directly
- Escape values added to OS commands specific to each OS
- Sanitized untrusted user input (& | ; $ > < ` \ !) properly
- Parameterization in conjunction with Input Validation
- Use structured mechanisms that automatically enforce the separation between data and command.

When using python os module for running system commands

Noncompliant Code

- os

```
import os

cmd = "ping -c 1 %s" % address
os.popen(cmd) # Noncompliant
```

Compliant Code

- os

```
import os
import shlex

address = shlex.quote("address") # address argument is shell-escaped
cmd = "ping -c 1 %s" % address
os.popen(cmd ) # Compliant
```

When using the python subprocess module for running system commands

Noncompliant Code

- subprocess

```
import subprocess

cmd = "ping -c 1 %s" % address
subprocess.Popen(cmd, shell=True) # Noncompliant; using shell=true is unsafe
```

Compliant Code

```
import subprocess

args = ["ping", "-c1", address]
subprocess.Popen(args) # Compliant
```

Server-Side Template Injection

When using Jinja2 or Mako with flask framework

- Never use render_template_string() to render a string containing template which was supplied with the user-supplied input without proper filtering.
- Whatever operations and tags to be used should be hardcoded in the separate template file and rendered by render_template().
- All the user-supplied inputs should be properly escaped using escape().

Noncompliant Code

```
@app.errorhandler(404)
def page_not_found(e):

template = '''{ extends "layout.html" }
{ block body }
<div class="center-content error">
<h1>Oops! That page doesn't exist.</h1>
<h3>%s</h3>
</div>
{ endblock }
''' % (request.url)

return render_template_string(template), 404
```

Compliant code

```
* templates/404.html

{ extends "layout.html" }
{ block body }
<div class="center-content error">
<h1>Oops! That page doesn't exist.</h1>
<h3>{{url}}</h3>
</div>
{ endblock }

* app.py

from flask import escape
@app.errorhandler(404)
def page_not_found(e):

return render_template('404.html', url=escape(request.url)), 404
```

- Enable auto-escaping by default and continue to review the use of inputs to be sure that the chosen auto-escaping strategy is the right one.

Noncompliant Code

```
from jinja2 import Environment

env = Environment() # Sensitive: New Jinja2 Environment has autoescape set to
false
env = Environment(autoescape=False) # Noncompliant
```

Compliant Code

```
from jinja2 import Environment
env = Environment(autoescape=True) # Compliant
```

When using Django Template

- String templates should not be passed with user-supplied input directly as a format string and rendered.
- Use Context() to handle user-supplied variables which need to be inserted into template strings.

Noncompliant Code

```
from django.template import Template

template = Template("My name is {{ %my_name }}."% userInput)

template.render(template)
```

Compliant code

```
from django.template import Context, Template

template = Template("My name is {{ %my_name }})
context = Context({"my_name": userInput})
template.render(context)
```

XPath Injection

- Treat all user input as untrusted, and perform appropriate sanitization.
- When sanitizing user input, verify the correctness of the data type, length, format, and content. For example, use a regular expression that checks for XML tags and special characters in user input. This practice corresponds to input sanitization.
- In a client-server application, perform validation at both the client and the server sides.
- Extensively test applications that supply, propagate, or accept user input.
- Use `lxml` module instead of `xml.etree.ElementTree`

Noncompliant Code

```
from flask import request
import xml.etree.ElementTree as ET

tree = ET.parse('users.xml')
root = tree.getroot()

@app.route('/user')
def user_location():
username = request.args['username']
query = ".//users/user/[@name='"+username+"']/location"
elmts = root.findall(query) # Noncompliant
return 'Location %s' % list(elmts)
```

Compliant Code

```
from flask import request
from lxml import etree

parser = etree.XMLParser(resolve_entities=False)
tree = etree.parse('users.xml', parser)
root = tree.getroot()

@app.route('/user')
def user_location():
username = request.args['username']
query = "/collection/users/user[@name = $paramname]/location/text()"
elmts = root.xpath(query, paramname = username)
return 'Location %s' % list(elmts)
```

LDAP Injection

- The special characters “"', ‘#’, “", ‘+’, ‘,’, ‘;’, ‘<’, ‘>’, ",'*’, ‘(’, ‘)’” and null must be escaped.
- Use Frameworks that Automatically Protect from LDAP Injection
- Minimize the privileges assigned to the LDAP binding account in your environment.
- Input validation can be used to detect unauthorized input before it is passed to the LDAP query.

Noncompliant Code

```
from flask import request
import ldap

@app.route("/user")
def user():
dn = request.args['dn']
username = request.args['username']

search_filter = "(&(objectClass=*)(uid="+username+"))"
ldap_connection = ldap.initialize("ldap://127.0.0.1:389")
user = ldap_connection.search_s(dn, ldap.SCOPE_SUBTREE, search_filter) #
Noncompliant
return user[0]
```

Compliant Code

```
from flask import request
import ldap
import ldap.filter
import ldap.dn

@app.route("/user")
def user():
dn = "dc=%s" % ldap.dn.escape_dn_chars(request.args['dc']) # Escape
distinguished names special characters
username = ldap.filter.escape_filter_chars(request.args['username']) # Escape
search filters special characters

search_filter = "(&(objectClass=*)(uid="+username+"))"
ldap_connection = ldap.initialize("ldap://127.0.0.1:389")
user = ldap_connection.search_s(dn, ldap.SCOPE_SUBTREE, search_filter) #
Compliant
return user[0]
```

Log Injection

- Filter the user input used to prevent injection of Carriage Return (CR) or Line Feed (LF) characters.
- Limit the size of the user input value used to create the log message.
- Encode to proper context, or sanitize user input
- Make sure all XSS defences are applied when viewing log files in a web browser.

Noncompliant Code

```
from flask import request, current_app
import logging

@app.route('/log')
def log():
input = request.args.get('input')
current_app.logger.error("%s", input) # Noncompliant
```

Compliant Code

```
from flask import request, current_app
import logging

@app.route('/log')
def log():
input = request.args.get('input')
if input.isalnum():
current_app.logger.error("%s", input) # Compliant
```

A2- Broken Authentication & Session Management

Broken Authentication

- Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.
- Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
- Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- Minimum passwords length should be at least eight (8) characters long. Combining this length with complexity makes a password difficult to guess and/or brute force.
- Do not ship or deploy with any default credentials, particularly for admin users.
- The application may should different HTTP Error code depending on the authentication attempt response.
- Use a side-channel to communicate the method to reset their password.
- Ensure that generated tokens or codes in forgot/reset password are:
- Randomly generated using a cryptographically safe algorithm.
- Sufficiently long to protect against brute-force attacks.
- Stored securely.
- Single-use and expire after an appropriate period.
- A stronger password can be enforced using the regex below, which requires at least 8 character password with numbers and both lowercase and uppercase letters.

```
var PASS_RE =/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}$/;
```

Session Management

- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login.
- Session IDs should not be exposed in the URL.
- Session IDs should timeout. User sessions or authentication tokens should get properly invalidated during logout.

- Passwords, session IDs, and other credentials should not be sent over unencrypted connections.
- Session IDs should never be under the control of users and clients to create. They should all be generated, controlled, and secured centrally by the authentication and authorisation mechanism.
- Session IDs should not be stored between browser sessions. They should be destroyed when a browser is closed. This prevents an attacker from retrieving a valid session by using the history in a Browser or by finding the Session ID on the client's local storage.

A3- Sensitive Data Exposure

- 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.
- Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
- Encrypt all data in transit with a secure protocol such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and security parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
- Disable caching for the response that contains sensitive data.
- Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt or PBKDF2.
- Verify independently the effectiveness of configuration and settings.
- Make sure to encrypt all sensitive data at rest.

A4- XML External Entities (XXE)

- Disable XML external entity and DTD processing in all XML parsers in the application.
- Implement positive (“whitelisting”) server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
- Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.
- Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system
- Use XML parsers such as `Etree`, `Minidom`, `Xmlrpc`, and `Genshi`

Noncompliant Code

```
from xml.dom import pulldom

data = pulldom.parse('rq.xml')
```

Compliant Code

```
from defusedxml import pulldom

data = parse('rq.xml')
```

A5- Broken Access Control

- Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable webserver directory listing and ensure file metadata (e.g. .git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g. repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.

- JWT tokens should be invalidated on the server after logout.
- When a user changes their 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.
- Access decisions should be made by checking if the current user has the permission associated with the requested application action.
- The authentication mechanism should deny all access by default, and provide access to specific roles for every function.
- In a workflow-based application, verify the users' state before allowing them to access any resources.
- Developers should use only one user or session for indirect object references.
- It is also recommended to check the access before using a direct object reference from an untrusted source.

A6- Security Misconfiguration

- A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
- A task to review and update the configurations appropriate to all security notes, updates and patches as part of the patch management process.
- An automated process to verify the effectiveness of the configurations and settings in all environments.
- Ensure that a strong application architecture is being adopted that provides effective, secure separation between components.
- Review default in HTTP Response headers to prevent internal implementation disclosure.
- Use generic session cookie names

A7- Cross-Site Scripting (XSS)

- Validate user-provided data based on a whitelist.
- Sanitize user-provided data from any characters that can be used for malicious purposes.
- HTML Escape Before Inserting Untrusted Data into HTML Element Content. Escape the following characters with HTML entity encoding to prevent switching into any execution context, such as script, style, or event handlers

```
& --> &amp;
< --> &lt;
> --> &gt;
" --> &quot;
' --> &#x27;
/ --> &#x2F;
```

- Encode user-provided data being reflected as output. Adjust the encoding to the output context so that, for example, HTML encoding is used for HTML content, HTML attribute encoding is used for attribute values, and JavaScript encoding is used for server-generated JavaScript.
- Implement Content Security Policy.

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

- When sanitizing or encoding data, it is recommended to only use libraries specifically designed for security purposes. Also, make sure that the library you are using is being actively maintained and is kept up-to-date with the latest discovered vulnerabilities

Noncompliant Code

```
templates/xss_shared.html

<!doctype html>
<title>Hello from Flask</title>
{% if name %}
<h1>Hello {{ name }}!</h1>
{% else %}
<h1>Hello, World!</h1>
{% endif %}

xss.py

@xss.route('/insecure/no_template_engine_replace', methods =['GET'])
def no_template_engine_replace():
param = request.args.get('param', 'not set')

html = open('templates/xss_shared.html').read()
response = make_response(html.replace('{{ name }}', param)) # Noncompliant:
param is not sanitized
return response
```

Compliant Code

```
templates/xss_shared.html

<!doctype html>
<title>Hello from Flask</title>
{% if name %}
<h1>Hello {{ name }}!</h1>
{% else %}
<h1>Hello, World!</h1>
{% endif %}

xss.py

@xss.route('/secure/no_template_engine_sanitized_Markup_escape', methods
=['GET'])
def no_template_engine_sanitized_Markup_escape():
param = request.args.get('param', 'not set')

param = Markup.escape(param)

html = open('templates/xss_shared.html').read()
response = make_response(html.replace('{{ name }}', param )) # Compliant:
'param' is sanitized by Markup.escape
return response
```

A8- Insecure Deserialization

- The pickle and jsonpickle module is not secure, never use it to deserialize untrusted data.
- Only use the PyYAML module with the default safe loader.
- Instead of using a native data interchange format, use a safe, standard format such as untyped JSON or structured data approaches such as Google Protocol Buffers.
- To ensure integrity is not compromised, add a digital signature (HMAC) to the serialized data that is validated before deserialization (this is only valid if the client doesn't need to modify the serialized data)
- Restrict deserialization to be possible only to specific, whitelisted classes.
- Isolating and running code that deserializes in low privilege environments when possible.
- Logging deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitoring deserialization, alerting if a user deserializes constantly.

- Use safe libraries that do not allow code execution at deserialization.
- Not communicate with the outside world using serialized objects
- Limit access to the serialized source
- if it is a file, restrict the access to it.
- if it comes from the network, restrict who has access to the process, such as with a Firewall or by authenticating the sender first.

Noncompliant Code

```
from flask import request
import pickle
import yaml

@app.route('/pickle')
def pickle_loads():
file = request.files['pickle']
pickle.load(file) # Noncompliant; Never use pickle module to deserialize user
inputs

@app.route('/yaml')
def yaml_load():
data = request.GET.get("data")
yaml.load(data, Loader=yaml.Loader) # Noncompliant; Avoid using yaml.load with
unsafe yaml.Loader
```

Compliant Code

```
from flask import request
import yaml

@app.route('/yaml')
def yaml_load():
data = request.GET.get("data")
yaml.load(data) # Compliant; Prefer using yaml.load with the default safe
loader
```

A9- Using Components with known vulnerabilities

- Run application with the least privilege user
- Prefer packages that contain comprehensive unit tests and review tests for the functions our application uses
- Review code for any unexpected file or database access

- Research about how popular the package is, what other packages use it if any other packages are written by the author, etc
- Use the latest stable version of the packages.
- Watch Github repositories for notifications. This will inform us if any vulnerabilities are discovered in the package in future
- Remove unused dependencies, unnecessary features, components, files, and documentation.
- Continuously inventory the versions of both client-side and server-side components (e.g. frameworks, libraries) and their dependencies using tools like versions, DependencyCheck, retire.js, etc. Continuously monitor sources like CVE and NVD for vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component.
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.
- Every organization must ensure that there is an ongoing plan for monitoring, triaging, and applying updates or configuration changes for the lifetime of the application or portfolio.

A10- Insufficient Logging and Monitoring

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

- Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later. There are commercial and open-source application protection frameworks such as OWASP AppSensor, web application firewalls such as ModSecurity with the OWASP ModSecurity Core Rule Set, and log correlation software with custom dashboards and alerting.

Unvalidated Redirects and Forwards

- Simply avoid using redirects and forwards.
- If used, do not allow the URL as user input for the destination.
- Where possible, have the user provide short name, ID or token which is mapped server-side to a full target URL.
- This provides the highest degree of protection against the attack tampering with the URL.
- Be careful that this doesn't introduce an enumeration vulnerability where a user could cycle through IDs to find all possible redirect targets
- If user input can't be avoided, ensure that the supplied value is valid, appropriate for the application, and is authorized for the user.
- Sanitize input by creating a list of trusted URLs (lists of hosts or a regex).
- This should be based on a white-list approach, rather than a blacklist.
- Force all redirects to first go through a page notifying users that they are going off of your site, with the destination displayed, and have them click a link to confirm.

Noncompliant Code

- Flask

```
from flask import request, redirect

@app.route('move')
def move():
url = request.args["next"]
return redirect(url) # Noncompliant
```

- Django

```
from django.http import HttpResponseRedirect

def move(request):
url = request.GET.get("next", "/")
return HttpResponseRedirect(url) # Noncompliant
```

Compliant Code

- Flask

```
from flask import request, redirect, url_for

@app.route('move')
def move():
endpoint = request.args["next"]
return redirect(url_for(endpoint)) # Compliant
```

- Django

```
from django.http import HttpResponseRedirect
from urllib.parse import urlparse

DOMAINS_WHITELIST = ['www.example.com', 'example.com']

def move(request):
url = request.GET.get("next", "/")
parsed_uri = urlparse(url)
if parsed_uri.netloc in DOMAINS_WHITELIST:
return HttpResponseRedirect(url) # Compliant
return HttpResponseRedirect("/")
```

Server Side Request Forgery(SSRF)

- Use a whitelist of allowed domains, resources and protocols from where the webserver can fetch resources.
- Any input accepted from the user should be validated and rejected if it does not match the positive specification expected.
- If possible, do not accept user input in functions that control where the webserver can fetch resources.

- Disable unused URL schemas.
- Migrate to IMDSv2 and disable old IMDSv1. IMDSv2 is an additional defence-in-depth mechanism for AWS that mitigates some of the instances of SSRF.

Noncompliant Code

```
from flask import request
import urllib

@app.route('/proxy')
def proxy():
url = request.args["url"]
return urllib.request.urlopen(url).read() # Noncompliant
```

Compliant Code

```
from flask import request
import urllib

DOMAINS_WHITELIST = ['domain1.com', 'domain2.com']

@app.route('/proxy')
def proxy():
url = request.args["url"]
if urllib.parse.urlparse(url).hostname in DOMAINS_WHITELIST:
return urllib.request.urlopen(url).read() # Compliant
```

Insecure File Upload

- Whitelist allowed extensions. Only allow safe and critical extensions for business functionality
- Validate the file type, don't trust the Content-Type header as it can be spoofed
- Change the filename to something generated by the application
- The application should perform filtering and content checking on any files which are uploaded to the server. Files should be thoroughly scanned and validated before being made available to other users. If in doubt, the file should be discarded.
- It is necessary to have a list of only permitted extensions on the web application. And, the file extension can be selected from the list.

- All the control characters and Unicode ones should be removed from the filenames and their extensions without any exception. Also, the special characters such as “;”, “:”, “>”, “<”, “/” ,"", additional “.”, “*”, “%”, “$”, and so on should be discarded as well. If it is applicable and there is no need to have Unicode characters, it is highly recommended to only accept Alpha-Numeric characters and only 1 dot as an input for the file name and the extension; in which the file name and also the extension should not be empty at all (regular expression: [a-zA-Z0-9]{1,200}.[a-zA-Z0-9]{1,10}).
- Limit the filename length. For instance, the maximum length of the name of a file plus its extension should be less than 255 characters (without any directory) in an NTFS partition.
- Uploaded directory should not have any “execute” permission and all the script handlers should be removed from these directories.
- Limit the file size to a maximum value to prevent denial of service attacks (on file space or other web application’s functions such as the image resizer).
- In case of having compressed file extract functions, contents of the compressed file should be checked one by one as a new file.
- If it is possible, consider saving the files in a database rather than on the filesystem.
- File uploaders should be only accessible to authenticated and authorised users if possible.
- Adding the “Content-Disposition: Attachment” and “X-Content-Type-Options: nosniff” headers to the response of static files will secure the website against Flash or PDF-based cross-site content-hijacking attacks.

Cross-Site Request Forgery (CSRF)

- Check if your framework has built-in CSRF protection and use it. If the framework does not have built-in CSRF protection add CSRF tokens to all state-changing requests (requests that cause actions on the site) and validate them on the backend.
- Always use SameSite Cookie Attribute for session cookies
- Use custom request headers
- Verify the origin with standard headers
- Use unpredictable Synchronizer Token Pattern. In this method, the website generates a random token in each form as a hidden value. This token is associated with the user’s current session.

- Use double submit cookies
- Remember that any Cross-Site Scripting (XSS) can be used to defeat all CSRF mitigation techniques.
- Do not use GET requests for state-changing operations.
- CSRF can be avoided by creating a unique token in a hidden field which would be sent in the body of the HTTP request rather than in an URL, which is more prone to exposure.
- Forcing the user to re-authenticate or proving that they are users to protect CSRF. For example, CAPTCHA.
- For a Django application, it is recommended to protect all the views with `django.middleware.csrf.CsrfViewMiddleware` and to not disable the CSRF protection on specific views:

```
def example(request): # Compliant
return HttpResponse("default")
```

- For a Flask application, the `CSRFProtect` module should be used (and not disabled further with `WTF_CSRF_ENABLED` set to false):

```
app = Flask(__name__)
csrf = CSRFProtect()
csrf.init_app(app) # Compliant
```

- It is recommended to not disable the CSRF protection on specific views or forms:

```
@app.route('/example/', methods=['POST']) # Compliant
def example():
return 'example '

class unprotectedForm(FlaskForm):
class Meta:
csrf = True # Compliant

name = TextField('name')
submit = SubmitField('submit')
```

Cross-origin resource sharing

When using Django

- The Access-Control-Allow-Origin header should be set only for a trusted origin and for specific resources.

Noncompliant Code

```
CORS_ORIGIN_ALLOW_ALL = True # Noncompliant
```

Compliant Code

```
CORS_ORIGIN_ALLOW_ALL = False # Compliant
```

When using Flask

- Use flask_cors.CORS() to set appropriate access configuration. All the trusted domain name should be mentioned who can access that specific resource.
- Wildcard should be set to False (“send_wildcard”: “False”).
- Null origin should not be set.
- User-supplied origin should not be used as origin unless developer wants that

Noncompliant Code

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*"}}) # Noncompliant
```

Compliant Code

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app, resources={r"/*": {"origins": "*", "send_wildcard": "False"}}) #
Compliant
```

Secure Flag missing from cookie

- It is recommended to use HTTPs everywhere so setting the secure flag to true should be the default behaviour when creating cookies.
- Set the secure flag to true for session-cookies.

When using Flask

Noncompliant code

```
from flask import Response

@app.route('/')
def index():
    response = Response()
    response.set_cookie('key', 'value') # noncompliant
    return response
```

Compliant Code

```
from flask import Response

@app.route('/')
def index():
    response = Response()
    response.set_cookie('key', 'value', secure=True) # Compliant
    return response
```

When using Django

- Set the SESSION_COOKIE_SECURE = True in the settings file

Noncompliant code

```
from django.conf import settings

settings.configure(SESSION_COOKIE_SECURE = False) # noncompliant
```

Compliant Code

```
from django.conf import settings
settings.configure(SESSION_COOKIE_SECURE = True) # Compliant
```

HTTPOnly Flag missing from cookie

- By default the HttpOnly flag should be set to true for most of the cookies and it's mandatory for session / sensitive-security cookies and csrf cookies.

When using Flask

Noncompliant Code

```
from flask import Response

@app.route('/')
def index():
    response = Response()
    response.set_cookie('key', 'value') # noncompliant
    return response
```

Compliant Code

```
from flask import Response

@app.route('/')
def index():
    response = Response()
    response.set_cookie('key', 'value', httponly=True) # Compliant
    return response
```

When using Django

- Set SESSION_COOKIE_HTTPONLY = True and CSRF_COOKIE_HTTPONLY = True in settings file of Django.

Noncompliant Code

```
from django.conf import settings

settings.configure(SESSION_COOKIE_HTTPONLY = False) # noncompliant
settings.configure(CSRF_COOKIE_HTTPONLY = Flase) # noncompliant
```

Compliant Code

```
from django.conf import settings

settings.configure(SESSION_COOKIE_HTTPONLY = True) # noncompliant
settings.configure(CSRF_COOKIE_HTTPONLY = True) # noncompliant
```

References

- https://rules.sonarsource.com/python
- https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html
- https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/

2. Java

A1- Injections

SQL Injection

- Use of Stored Procedures

```
  // 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...
  }
```

- Whitelist Input Validation
- Escaping Wildcard characters in Like Clauses
- Escaping All User Supplied Input

Noncompliant Code

```
  String query = "SELECT user_id FROM user_data WHERE user_name = '"
               + req.getParameter("userID")
               + "' and user_password = '" + req.getParameter("pwd") +"'";
  try {
      Statement statement = connection.createStatement( … );
      ResultSet results = statement.executeQuery( query );
  }
```

Compliant Code

```
  Codec ORACLE_CODEC = new OracleCodec();
  String query = "SELECT user_id FROM user_data WHERE user_name = '"
  + ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("userID"))
  + "' and user_password = '"
  + ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("pwd")) +"'";
```

- Enforcing Least Privilege
- Performing Whitelist Input Validation as a Secondary Defense
- Use of Prepared Statements (with Parameterized Queries)

Noncompliant Code

```
String query = "SELECT * FROM users WHERE username = '"+ username +"' AND
password = '" + password + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

Compliant Code

```
String query = "select * from db_user where username=? and password=?";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setString(1, username);
stmt.setString(2, password);
ResultSet rs = stmt.executeQuery();
```

HQL Injection

- Use of Prepared Statements (with Parameterized Queries)

Noncompliant Code

```
Query unsafeHQLQuery = session.createQuery("from Inventory where
productID='"+userSuppliedParameter+"'");
```

Compliant Code

```
Query safeHQLQuery = session.createQuery("from Inventory where
productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

OS Command Injection

- Avoid calling OS commands directly

Noncompliant Code

```
ProcessBuilder b = new ProcessBuilder("C:\DoStuff.exe -arg1 -arg2");
```

Compliant Code

```
ProcessBuilder pb = new ProcessBuilder("TrustedCmd", "TrustedArg1",
"TrustedArg2");
Map<String, String> env = pb.environment();
pb.directory(new File("TrustedDir"));
Process p = pb.start();
```

- Escape values added to OS commands specific to each OS
- Parameterization in conjunction with Input Validation
- Use structured mechanisms that automatically enforce the separation between data and command.

Noncompliant Code

```
String folder = request.getParameter("folder");
String cmd = "mkdir " + folder;
Runtime.getRuntime().exec(cmd);
```

Compliant Code

```
String folder = request.getParameter("folder");
String cmd = "mkdir " + ESAPI.encoder().encodeForOS(new WindowsCodec(),
folder);
Runtime.getRuntime().exec(cmd);
```

LDAP Injection

- Escape all variables using the right LDAP encoding function

```
public String escapeDN (String name) {
 //From RFC 2253 and the / character for JNDI
 final char[] META_CHARS = {'+', '"', '<', '>', ';', '/'};
 String escapedStr = new String(name);
 //Backslash is both a Java and an LDAP escape character,
 //so escape it first
 escapedStr = escapedStr.replaceAll("\\\\\\\\","\\\\\\\\");
 //Positional characters - see RFC 2253
 escapedStr = escapedStr.replaceAll("\^#","\\\\\\\\#");
 escapedStr = escapedStr.replaceAll("\^ | $","\\\\\\\\ ");
 for (int i=0 ; i < META_CHARS.length ; i++) {
        escapedStr = escapedStr.replaceAll("\\\\" +
                     META_CHARS[i],"\\\\\\\\" + META_CHARS[i]);
 }
 return escapedStr;
}
```

Note, that the backslash character is a Java String literal and a regular expression escape character.

```
public String escapeSearchFilter (String filter) {
 //From RFC 2254
 String escapedStr = new String(filter);
 escapedStr = escapedStr.replaceAll("\\\\\\\\","\\\\\\\\5c");
 escapedStr = escapedStr.replaceAll("\\\\\*","\\\\\\\\2a");
 escapedStr = escapedStr.replaceAll("\\\\(","\\\\\\\\28");
 escapedStr = escapedStr.replaceAll("\\\\)","\\\\\\\\29");
 escapedStr = escapedStr.replaceAll("\\\\" +
               Character.toString('\\u0000'), "\\\\\\\\00");
 return escapedStr;
}
```

- Use Frameworks that Automatically Protect from LDAP Injection
- Minimize the privileges assigned to the LDAP binding account in your environment.
- Input validation can be used to detect unauthorized input before it is passed to the LDAP query.

Noncompliant Code

```
String filter = "(&(uid=" + user + ")(userPassword=" + pass + "))";
NamingEnumeration<SearchResult> results = ctx.search("ou=system", filter, new
SearchControls());
return results.hasMore();
```

Compliant Code

```
String filter = "(&(uid={0})(userPassword={1}))";
NamingEnumeration<SearchResult> results = ctx.search("ou=system", filter, new
String[]{user, pass}, new SearchControls();
return results.hasMore();
```

Code Injection

Noncompliant Code

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("javascript");
engine.eval("print('"+ firstName + "')");
```

Compliant Code

```
// Whitelisting
if (!firstName.matches("[\\w]*")) {
    throw new IllegalArgumentException();
    }
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("javascript");
engine.eval("print('"+ firstName + "')");
```

Log Injection

- Filter the user input used to prevent injection of Carriage Return (CR) or Line Feed (LF) characters.
- Limit the size of the user input value used to create the log message.
- Make sure all XSS defenses are applied when viewing log files in a web browser.

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
...
Logger logger = LogManager.getLogger(MyClass.class);
logger.info(logMessage);
...
```

NoSQL Injection

- Do not use string concatenation to build API call expression but use the API to create the expression.

Example: MongoDB

```
String userInput = "Brooklyn";

ArrayList<String> specialCharsList = new ArrayList<String>() { {
    add("'");
    add("\"");
    add("\\");
    add(";");
    add("{");
    add("}");
    add("$");
} };
specialCharsList.forEach(specChar ->
Assert.assertFalse(userInput.contains(specChar)));
Assert.assertTrue(userInput.length() <= 50);

try(MongoClient mongoClient = new MongoClient()){
    MongoDatabase db = mongoClient.getDatabase("test");
    Bson expression = eq("borough", userInput);
    FindIterable<org.bson.Document> restaurants =
db.getCollection("restaurants").find(expression);
    restaurants.forEach(new Block<org.bson.Document>() {
        @Override
        public void apply(final org.bson.Document doc) {
            String restBorough = (String)doc.get("borough");
            Assert.assertTrue("Brooklyn".equals(restBorough));
        }
    });
}
```

HTML/JavaScript/CSS Injection

- Either apply strict input validation (whitelist approach)
- Use output sanitizing+escaping if input validation is not possible (combine both every time is possible).

Example:

```
String userInput = "You user login is owasp-user01";
Assert.assertTrue(Pattern.matches("[a-zA-Z0-9\\s\\-]{1,50}", userInput));

Assert.assertEquals(0, StringUtils.countMatches(userInput.replace(" ", ""), "-
-"));

String outputToUser = "You <p>user login</p> is <strong>owasp-
user01</strong>";
outputToUser += "<script>alert(22);</script><img src='#'
onload='javascript:alert(23);'>";

PolicyFactory policy = new HtmlPolicyBuilder().allowElements("p",
"strong").toFactory();

String safeOutput = policy.sanitize(outputToUser);

safeOutput = Encode.forHtml(safeOutput);
String finalSafeOutputExpected = "You <p>user login</p> is <strong>owasp-
user01</strong>";
Assert.assertEquals(finalSafeOutputExpected, safeOutput);
```

Server Side Request Forgery(SSRF)

- Use a whitelist of allowed domains, resources and protocols from where the web server can fetch resources.
- Any input accepted from the user should be validated and rejected if it does not match the positive specification expected.
- If possible, do not accept user input in functions that control where the web server can fetch resources.
- Disable unused URL schemas

** Noncompliant Code **

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
  URL url = new URL(req.getParameter("url"));
  HttpURLConnection conn = (HttpURLConnection) url.openConnection(); //
Noncompliant
}
```

A2 - Broken Authentication

- Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.
- Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
- Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.

Noncompliant Code

This example violates the principle of least privilege because an unprivileged caller could also cause the authentication library to be loaded. An unprivileged caller cannot invoke the System.loadLibrary() method directly because this could expose

```
AccessController.doPrivileged(new PrivilegedAction() {
    public Object run() {
      try {
        String passwordFile = System.getProperty("user.dir") + File.separator
+ "PasswordFileName";
        f[0] = new FileInputStream(passwordFile);
        // Check whether oldPassword matches the one in the file
        // If not, throw an exception
        System.loadLibrary("authentication");
      } catch (FileNotFoundException cnf) {
        // Forward to handler
      }
      return null;
    }
  }); // End of doPrivileged()
```

Compliant Code

This compliant solution moves the call to System.loadLibrary() outside the doPrivileged() block. Doing so allows unprivileged code to perform preliminary password reset checks using the file but prevents it from loading the authentication library.

```
  AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        try {
          String passwordFile = System.getProperty("user.dir") + File.separator
+ "PasswordFileName";
          f[0] = new FileInputStream(passwordFile);
          // Check whether oldPassword matches the one in the file
          // If not, throw an exception
        } catch (FileNotFoundException cnf) {
          // Forward to handler
        }
        return null;
      }
    }); // End of doPrivileged()

    System.loadLibrary("authentication");
```

Authentication Solution and Sensitive Accounts

- Do NOT allow login with sensitive accounts (i.e. accounts that can be used internally within the solution such as to a back-end / middle-ware / DB) to any front-end user-interface
- Do NOT use the same authentication solution (e.g. IDP / AD) used internally for unsecured access (e.g. public access / DMZ)

Implement Proper Password Strength Controls

- Minimum length of the passwords should be enforced by the application. Passwords shorter than 8 characters are considered to be weak
- Maximum password length should not be set too low, as it will prevent users from creating passphrases. A common maximum length is 64 characters due to limitations in certain hashing algorithms.
- Allow usage of all characters including unicode and whitespace. There should be no password composition rules limiting the type of characters permitted.
- Implement weak-password checks, such as testing new or changed passwords against a list of the top 10000 worst passwords.

- Ensure credential rotation when a password leak, or at the time of compromise identification.
- Include password strength meter to help users create a more complex password and block common and previously breached passwords
- Align password length, complexity and rotation policies with NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets or other modern, evidence based password policies.
- Do not ship or deploy with any default credentials, particularly for admin users.

Compare Password Hashes Using Safe Functions

Ensure that the comparison function:

- Has a maximum input length, to protect against denial of service attacks with very long inputs.
- Explicitly sets the type of both variable, to protect against type confusion attacks.
- Returns in constant time, to protect against timing attacks.

Authentication and Error Messages

- The application may should different HTTP Error code depending on the authentication attempt response.
- Do not return different error messages.

Authentication Responses

Using any of the authentication mechanisms (login, registration, 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.

Password Managers

Web applications should at least not make password managers job more difficult than necessary by observing the following recommendations:

- Use standard HTML forms for username and password input with appropriate type attributes.
- Avoid plugin-based login pages (such as Flash or Silverlight).
- Allow any printable characters to be used in passwords.
- Allow users to paste into the username and password fields.
- Allow users to navigate between the username and password field with a single press of the Tab key.

Forgot Password Services

- Return a consistent message for both existent and non-existent accounts.
- Ensure that the time taken for the user response message is uniform.
- Use a side-channel to communicate the method to reset their password.
- Use URL tokens for the simplest and fastest implementation.
- Ensure that generated tokens or codes are:
- Randomly genererated using a cryptographically safe algorithm.
- Sufficiently long to protect against brute-force attacks.
- Stored securely.
- Single use and expire after an appropriate period.

Session management

- Force Session Logout On Web Browser Window Close Events
- Disable Web Browser Cross-Tab Sessions
- Renew the Session ID After Any Privilege Level Change
- All sessions should implement an idle or inactivity timeout.
- Session IDs are vulnerable to hijacking when they are passed across the network with http requests. As with hashed passwords it's essential that session ID tokens are protected with SSL to ensure they remain secure.
- Session IDs should never be under the control of users and clients to create. They should all be generated, controlled, and secured centrally by the authentication and authorisation mechanism.
- Session IDs should not be stored between browser sessions. They should be destroyed when a browser is closed. This prevents an attacker retrieving a valid session by using the history in a Browser or by finding the Session ID on the client's local storage.

A3 - Sensitive Data Exposure

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

Unrestricted File Upload

- The file types allowed to be uploaded should be restricted to only those that are necessary for business functionality.
- Never accept a filename and its extension directly without having an allow list filter.
- The application should perform filtering and content checking on any files which are uploaded to the server. Files should be thoroughly scanned and validated before being made available to other users. If in doubt, the file should be discarded.
- It is necessary to have a list of only permitted extensions on the web application. And, file extension can be selected from the list.
- All the control characters and Unicode ones should be removed from the filenames and their extensions without any exception. Also, the special characters such as ";", ":", ">", "<", "/" ,"\", additional ".", "*", "%", "$", and so on should be discarded as well. If it is applicable and there is no need to have

Unicode characters, it is highly recommended to only accept Alpha-Numeric characters and only 1 dot as an input for the file name and the extension; in which the file name and also the extension should not be empty at all (regular expression: [a-zA-Z0-9]{1,200}.[a-zA-Z0-9]{1,10}).

- Limit the filename length. For instance, the maximum length of the name of a file plus its extension should be less than 255 characters (without any directory) in an NTFS partition.
- It is recommended to use an algorithm to determine the filenames. For instance, a filename can be a MD5 hash of the name of file plus the date of the day.
- Uploaded directory should not have any “execute” permission and all the script handlers should be removed from these directories.
- Limit the file size to a maximum value in order to prevent denial of service attacks (on file space or other web application’s functions such as the image resizer).
- Restrict small size files as they can lead to denial of service attacks. So, the minimum size of files should be considered.
- Use Cross Site Request Forgery protection methods.
- Prevent from overwriting a file in case of having the same hash for both.
- Use a virus scanner on the server (if it is applicable). Or, if the contents of files are not confidential, a free virus scanner website can be used. In this case, file should be stored with a random name and without any extension on the server first, and after the virus checking (uploading to a free virus scanner website and getting back the result), it can be renamed to its specific name and extension.
- Try to use POST method instead of PUT (or GET!)
- Log users’ activities. However, the logging mechanism should be secured against log forgery and code injection itself.
- In case of having compressed file extract functions, contents of the compressed file should be checked one by one as a new file.
- If it is possible, consider saving the files in a database rather than on the filesystem.
- If files should be saved in a filesystem, consider using an isolated server with a different domain to serve the uploaded files.
- File uploaders should be only accessible to authenticated and authorised users if possible.

- Write permission should be removed from files and folders other than the upload folders.
- Ensure that configuration files such as ".htaccess" or "web.config" cannot be replaced using file uploaders. Ensure that appropriate settings are available to ignore the ".htaccess" or "web.config" files if uploaded in the upload directories.
- Ensure that files with double extensions (e.g. "file.jsp.txt") cannot be executed especially in Apache.
- Ensure that uploaded files cannot be accessed by unauthorised users.
- Adding the "Content-Disposition: Attachment" and "X-Content-Type-Options: nosniff" headers to the response of static files will secure the website against Flash or PDF-based cross-site content-hijacking attacks. It is recommended that this practice be performed for all of the files that users need to download in all the modules that deal with a file download. Although this method does not fully secure the website against attacks using Silverlight or similar objects, it can mitigate the risk of using Adobe Flash and PDF objects, especially when uploading PDF files is permitted.
- CORS headers should be reviewed to only be enabled for static or publicly accessible data. Otherwise, the "Access-Control-Allow-Origin" header should only contain authorised addresses. Other CORS headers such as "Access-Control-Allow-Credentials" should only be used when they are required. Items within the CORS headers such as "Access-Control-Allow-Methods" or "Access-Control-Allow-Headers" should be reviewed and removed if they are not required.

Noncompliant Code

```
public class UploadAction extends ActionSupport {
  private File uploadedFile;
  // setter and getter for uploadedFile

  public String execute() {
    try {
      // File path and file name are hardcoded for illustration
      File fileToCreate = new File("filepath", "filename");
      // Copy temporary file content to this file
      FileUtils.copyFile(uploadedFile, fileToCreate);
      return "SUCCESS";
    } catch (Throwable e) {
      addActionError(e.getMessage());
      return "ERROR";
    }
  }
}
```

Compliant Code

```
  public class UploadAction extends ActionSupport {
    private File uploadedFile;
    // setter and getter for uploadedFile

    public String execute() {
      try {
        // File path and file name are hardcoded for illustration
        File fileToCreate = new File("filepath", "filename");

        boolean textPlain = checkMetaData(uploadedFile, "text/plain");
        boolean img = checkMetaData(uploadedFile, "image/JPEG");
        boolean textHtml = checkMetaData(uploadedFile, "text/html");

        if (!textPlain && !img && !textHtml) {
          return "ERROR";
        }

        // Copy temporary file content to this file
        FileUtils.copyFile(uploadedFile, fileToCreate);
        return "SUCCESS";
      } catch (Throwable e) {
        addActionError(e.getMessage());
        return "ERROR";
      }
    }

    public static boolean checkMetaData(
      File f, String getContentType) {
      try (InputStream is = new FileInputStream(f)) {
        ContentHandler contenthandler = new BodyContentHandler();
        Metadata metadata = new Metadata();
        metadata.set(Metadata.RESOURCE_NAME_KEY, f.getName());
        Parser parser = new AutoDetectParser();
        try {
          parser.parse(is, contenthandler, metadata, new ParseContext());
        } catch (SAXException | TikaException e) {
          // Handle error
          return false;
        }

        if
(metadata.get(Metadata.CONTENT_TYPE).equalsIgnoreCase(getContentType)) {
          return true;
        } else {
          return false;
        }
      } catch (IOException e) {
        // Handle error
        return false;
      }
    }
  }
```

A4 - XML External Entities (XXE)

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

XXE attacks

Noncompliant Code

```
class XXE {
  private static void receiveXMLStream(InputStream inStream,
                                       DefaultHandler defaultHandler)
      throws ParserConfigurationException, SAXException, IOException {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser saxParser = factory.newSAXParser();
    saxParser.parse(inStream, defaultHandler);
  }

  public static void main(String[] args) throws ParserConfigurationException,
      SAXException, IOException {
    try {
      receiveXMLStream(new FileInputStream("evil.xml"), new DefaultHandler());
    } catch (java.net.MalformedURLException mue) {
      System.err.println("Malformed URL Exception: " + mue);
    }
  }
}
```

Compliant Code

```
class CustomResolver implements EntityResolver {
  public InputSource resolveEntity(String publicId, String systemId)
      throws SAXException, IOException {

    // Check for known good entities
    String entityPath = "file:/Users/onlinestore/good.xml";
    if (systemId.equals(entityPath)) {
      System.out.println("Resolving entity: " + publicId + " " + systemId);
      return new InputSource(entityPath);
    } else {
      // Disallow unknown entities by returning a blank path
      return new InputSource();
    }
  }
}
```

XPath Injection

- Treat all user input as untrusted, and perform appropriate sanitization.
- When sanitizing user input, verify the correctness of the data type, length, format, and content. For example, use a regular expression that checks for XML tags and special characters in user input. This practice corresponds to input sanitization. See IDS52-J. Prevent code injection for additional details.
- In a client-server application, perform validation at both the client and the server sides.
- Extensively test applications that supply, propagate, or accept user input.

Noncompliant Code

```
XPathFactory factory = XPathFactory.newInstance();
XPath xpath = factory.newXPath();
XPathExpression expr = xpath.compile("//users/user[username/text()='" +
userName + "' and password/text()='" + pwd + "' ]");
Object result = expr.evaluate(doc, XPathConstants.NODESET);
NodeList nodes = (NodeList) result;
```

Compliant Code

```
XQuery xquery = new XQueryFactory().createXQuery(new File("login.xq"));
  Map queryVars = new HashMap();
  queryVars.put("userName", userName);
  queryVars.put("password", pwd);
  NodeList nodes = xquery.execute(doc, null, queryVars).toNodes();
```

A5 - Broken Access Control

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

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

Role-Based Access Control (RBAC)

- Roles must be only be transferred or delegated using strict sign-offs and procedures.
- When a user changes their 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.

Discretionary Access Control (DAC)

- Assurance while granting trusts to different user roles.
- Assurance for DAC must be carried out using strict access control reviews.

Mandatory Access Control (MAC)

- Classification and sensitivity assignment at an appropriate and pragmatic level
- Assurance for MAC must be carried out to ensure that the classification of the objects is at the appropriate level.

Permission Based Access Control

- Access decisions should be made by checking if the current user has the permission associated with the requested application action.
- The has relationship between the user and permission may be satisfied by creating a direct relationship between the user and permission (called a grant),

Missing Functional Level Access Controls

- The authentication mechanism should deny all access by default, and provide access to specific roles for every function.
- In a workflow based application, verify the users' state before allowing them to access any resources.

Insecure Direct Object References

- Developers should use only one user or session for indirect object references.
- It is also recommended to check the access before using a direct object reference from an untrusted source.

A6 - Security Misconfiguration

- A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment. This process should be automated to minimize the effort required to setup a new secure environment.
- A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.

- A task to review and update the configurations appropriate to all security notes, updates and patches as part of the patch management process. In particular, review cloud storage permissions (e.g. S3 bucket permissions).
- A segmented application architecture that provides effective, secure separation between components or tenants, with segmentation, containerization, or cloud security groups (ACLs).
- Sending security directives to clients, e.g. Security Headers.
- An automated process to verify the effectiveness of the configurations and settings in all environments.
- Ensure that a strong application architecture is being adopted that provides effective, secure separation between components.
- It can also minimize the possibility of this attack by running automated scans and doing audits periodically.

Clickjacking

- Preventing the browser from loading the page in frame using the X-Frame-Options or Content Security Policy (frame-ancestors) HTTP headers.
- Preventing session cookies from being included when the page is loaded in a frame using the SameSite cookie attribute.
- Implementing JavaScript code in the page to attempt to prevent it being loaded in a frame (known as a “frame-buster”).

A7 - Cross Site Scripting (XSS)

- Implement Content Security Policy.

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

- Never Insert Untrusted Data Except in Allowed Locations.
- HTML Escape Before Inserting Untrusted Data into HTML Element Content. Escape the following characters with HTML entity encoding to prevent switching into any execution context, such as script, style, or event handlers

```
 & --> &amp;
 < --> &lt;
 > --> &gt;
 " --> &quot;
 ' --> &#x27;
 / --> &#x2F;
```

- Sanitize HTML Markup with a Library Designed for the Job

OWASP Java Html Sanitizer

```
PolicyFactory sanitizer = Sanitizers.FORMATTING.and(Sanitizers.BLOCKS);
String cleanResults = sanitizer.sanitize("<p>Hello, <b>World!</b>");
```

- Use HTTPOnly cookie flag

Noncompliant Code

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setHttpOnly(false);
```

Compliant Code

```
Cookie c = new Cookie(COOKIENAME, sensitivedata);
c.setHttpOnly(true);
```

- Applications that use user provided data to construct the response header should always validate the data first. Validation should be based on a whitelist.

Noncompliant Code

```
String value = req.getParameter("value");
resp.addHeader("X-Header", value); //
```

Compliant Code

```
String value = req.getParameter("value");
String whitelist = "safevalue1 safevalue2";
if (!whitelist.contains(value))
    throw new IOException();
resp.addHeader("X-Header", value);
```

- Encode user provided data being reflected as output. Adjust the encoding to the output context so that, for example, HTML encoding is used for HTML content, HTML attribute encoding is used for attribute values, and JavaScript encoding is used for server-generated JavaScript.

a. Example 1:

Noncompliant Code

```
String name = req.getParameter("name");
PrintWriter out = resp.getWriter();
out.write("Hello " + name);
```

Compliant Code

```
String name = req.getParameter("name");
String encodedName = org.owasp.encoder.Encode.forHtml(name);
PrintWriter out = resp.getWriter();
out.write("Hello " + encodedName);
```

b. Example 2:

Noncompliant Code

```
String url = "https://example.com?query=" + q;
return url;
```

Compliant Code

```
String encodedUrl = "https://example.com?query=" +
Base64.getUrlEncoder().encodeToString(q.getBytes());
return encodedUrl;
```

A8 - Insecure Deserialization

The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types. If that is not possible, consider one of more of the following:

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

```
private final void readObject(ObjectInputStream in) throws java.io.IOException
{
    throw new java.io.IOException("Cannot be deserialized");
}
```

- Harden Your Own java.io.ObjectInputStream

```
  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);
      }
  }
```

A9 - Using Components with known vulnerabilities

There should be a patch management process in place to:

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

A10 - Insufficient Logging and Monitoring

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

Cross Site Request Forgery (CSRF)

- Check if your framework has built-in CSRF protection and use it.
- If framework does not have built-in CSRF protection add CSRF tokens to all state changing requests (requests that cause actions on the site) and validate them on backend.
- Always use SameSite Cookie Attribute for session cookies
- Use custom request headers
- Verify the origin with standard headers
- Use double submit cookies
- Consider implementing user interaction based protection for highly sensitive operations
- Remember that any Cross-Site Scripting (XSS) can be used to defeat all CSRF mitigation techniques.
- See the OWASP XSS Prevention Cheat Sheet for detailed guidance on how to prevent XSS flaws.
- Do not use GET requests for state changing operations.

- CSRF can be avoided by creating a unique token in a hidden field which would be sent in the body of the HTTP request rather than in an URL, which is more prone to exposure.
- Forcing the user to re-authenticate or proving that they are users in order to protect CSRF. For example, CAPTCHA.

Noncompliant Code

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable(); // It will disable the CSRF protection
  }
}
```

Compliant Code

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    // http.csrf().disable(); // By default CSRF protection is enabled
  }
}
```

Unvalidated Redirects and Forwards

- Simply avoid using redirects and forwards.
- If used, do not allow the URL as user input for the destination.
- Where possible, have the user provide short name, ID or token which is mapped server-side to a full target URL.
- This provides the highest degree of protection against the attack tampering with the URL.
- Be careful that this doesn't introduce an enumeration vulnerability where a user could cycle through IDs to find all possible redirect targets

- If user input can't be avoided, ensure that the supplied value is valid, appropriate for the application, and is authorized for the user.
- Sanitize input by creating a list of trusted URLs (lists of hosts or a regex).
- This should be based on a white-list approach, rather than a blacklist.
- Force all redirects to first go through a page notifying users that they are going off of your site, with the destination clearly displayed, and have them click a

Noncompliant Code

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
  String location = req.getParameter("url");
  resp.sendRedirect(location); // Noncompliant
}
```

Compliant Code

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
    String whiteList = "http://localhost/safe";
    String location = req.getParameter("url");

    if (!location.startsWith(whiteList))
      throw new IOException();

    resp.sendRedirect(location); // Compliant
}
```

References

- https://wiki.sei.cmu.edu/confluence/display/java
- https://rules.sonarsource.com/java/RSPEC-2078
- https://cheatsheetseries.owasp.org/cheatsheets
- https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/

3. JavaScript

JavaScript Recommendations for OWASP TOP 10 2017

A7 - Cross Site Scripting (XSS)

DOM based XXS

- DOM based XSS happens when XSS becomes possible based on DOM "environment"-based manipulation.in the victim's browser used by the original client side script, so that the client side code runs in an "unexpected" manner. The page response does not change , but the client code contained in the page gets executed due to manipulation in the DOM environment.
    a. Think of it as a dormant payload, which becomes active only when the DOM manipulates it in certain ways.
- INput and Output so called Source and Sink
    a. Dom XSS will appear when source that can be controlled by the user is used in a dangerous sink

```
- Popular Sources
  document.URL
  document.documentURI
  location.href
    location.search
      location.*
      window.name
      document.referrer

- Popular sinks
  HTML Modification sinks
  document.write / (element).innerHTML / document.writeln / <ifrane>_srcdoc
and DOMParser.parserFromString

    HTML modification to behaviour change
      (element).src

    Execution Related sinks
      eval / setTimout / setInterval / execScript / new Fucntion ()
```

- Prevention using DOMPurify
  a. For Fixing voilation in the code we use trusted type where we can specify the code which can be detected and can be terminated
  b. DOMPurify gives the ability by using as a library , where you can customize and use the literary to pinpoint , DOMPurify supports trusted types which will return Sanitized HTML wrapped in a TrustedHTML
  c. In the below example you can create a policy where you can use DOMPurify library

```
   if (window.trustedTypes && trustedTypes.createPolicy) { // Feature testing
   trustedTypes.createPolicy('default', {
   createHTML: (string, sink) => DOMPurify.sanitize(string,
{RETURN_TRUSTED_TYPE: true})
   });
   }
```

- basically here we are using the Default policy name, where the createHTML will execute with the default policies which are present in the DOMPurify which will escape the strings in the TrustedType.

Trusted Types

JavaScript Recommendations for OWASP TOP 10 2017

- TrustedType minimizes the risk for attack of DOM XSS , by detecting the vulnerability similar to the regular programming error.
- Trusted Types works by using the risky sinks as we have already mention the above sinks and sources point , before using this Trusted Types will process the data before passing it to the sinks

- invalid

```
  anElement.innerHTML  = location.href;
```

- valid

```
  anEliment.innerHTML = aTrustedHTML;
```

- list of TrustedTypes

```
TrustedHTML / TrustedScript / TrustedScriptURL
```

- with this TrustedType enabled browser will only accept the TrusteHTML object for the sinks
- mainly TrustedTypes are created ,by creating policies , you can either create piles of policies or you can create single policies and add TrustedType object all in one .
- for example you can create a trusted type policie for TrustedHTML and can be added.

```
if (window.trustedTypes && trustedTypes.createPolicy) { // Feature testing
```

- You can create a Trusted Tyoe with CSP ,for example - You can add a report-only CSP header , which can eventually report the manipulation/violation to the report uri which you have mentioned

A6- Security Misconfiguration

Clickjacking

- Preventing the browser from loading the page in frame using the X-Frame-Options or Content Security Policy (frame-ancestors) HTTP headers.
- Preventing session cookies from being included when the page is loaded in a frame using the SameSite cookie attribute.
- X-Frame-Options
  1. The X-Frame-Options HTTP header can be used to indicate whether or not a browser should be allowed to render a page in a <frame>, <iframe> or <object> tag. It was designed specifically to help protect against clickjacking.
  2. DENY " The page cannot be displayed in a frame, regardless of the site attempting to do so."
  3. SAMEORIGNE " The page can only be displayed in a frame on the same origin as the page itself."
  4. ALLOW-FROM uri “The page can only be displayed in a frame on the specified origins.”
- CSP: frame-ancestors

```
Content-Security-Policy: frame-ancestors <source>;
Content-Security-Policy: frame-ancestors <source> <source>;
```

```
Content-Security-Policy: frame-ancestors 'none';

Content-Security-Policy: frame-ancestors 'self' https://www.ASDF;
```

- Same site cookie
  1. Same site cookies allows you to declare if your cookie should be restricted to a first-party or same-site context example :-
  2. If the user is on www.web.dev and requests an image from static.web.dev then that is a same-site request
  3. SameSite attribute on a cookie provides three different ways to control this behavior. You can choose to not specify the attribute, or you can use Strict or Lax to limit the cookie to same-site requests.
  4. SameSite strict

```
Set-Cookie: promo_shown=1; SameSite=Strict
```

- this comes in to picture when , user visits a site , then usually the cookie will be sent with the same reuqest , however when you are following a lonk or any sort of link the cookie will not be sent.
- SameSite lax

```
Set-Cookie: promo_shown=1; SameSite=LAX
```

- This will come in to picture when the user follows the link on some site , that perticular request will include the cookie.
- This makes Lax a good choice for cookies affecting the display of the site with Strict being useful for cookies related to actions your user is taking.
  1. Samesite cookie none
     a. This comes in to picture when you widely want to support the cookie over the request , without any restrictions

```
Set-Cookie: widget_session=abc123; SameSite=None; Secure
```

CORS

- Access-control-Allow-origin
- Set origin to its intent domain other then anything

```
Access-Control-Allow-Origin: https://mozilla.org
  Vary: Origin
```

- Access-Control-Expose-Headers

```
  Access-Control-Expose-Headers: X-My-Custom-Header, X-Another-Custom-Header
```

- Access-Control-Allow-Credentials only when request is authenticated

```
Access-Control-Allow-Credentials: true
```

PostMessage XSS

- Windows.postMessage method helps to solve the Cross-Origin communication, providing a controlled method of communication between two windows.
- As a simple example - a page from webiste.1 wants information from website.2 which will open in a different window or a iframe as website.2, the “Windows.postMessage” will allow to sent the string or object to a different window or frame using javascript.
- webiste1 will request the website2 for information, where the website2 should first obtain reference to recipient window and then call the postmessage

```
    otherWindows.postmessage(message, targetOrigin, [transfer])
```

- in the above code “otherWindow” is a reference to other window, “Message” is an object or string to send to the other window, “targetOrigin” is a target message, “Transfer” object which is transferred with the message.
- eaxmples for the postMessage
  1. parent.postMessage()
  2. iframe.contentWindow.postmessage()
  3. window.opener.postMessage()
  4. event.source.postMessage()

NoN-compliant - page which sends message website1.html

```
var childWin = window.open("http://website2.html","child");
childWin.postMessage("Message for ya!","*");
```

Non-Compliant - page which receives the message website2.html

```
function checkMessage(message) {
     alert("Got a message! Contents: " + message.data);
}
window.addEventListener("message", receiveMessage, false
```

- Always specify the exact origin, not “*”, when you use a post message to send the data to other windows.

Compliant - website1.HTML

```
var childWin = window.open("http://website2.html","child");
childWin.postMessage("Message for ya!","website2.html");
```

Non-Compliant - page which receives the message website2.html

```
function checkMessage(message) {
     alert("Got a message! Contents: " + message.data);
}
window.addEventListener("message", receiveMessage, false);
if (event.origin !== "http://website1.HTML")
    return;
```

Prototype Pollution

- prototype pollution was introduced a way back where the in the javascript, this was used as an extension method to the prototype of the base object like “Object”, “string” or “function”.
- Prototype Pollution starts where the attacker has control over the parameter value of “a” and value “obj[a][b] = value “
- Most vulnerable library used are as below
  1. “hoek”
     a. hoek.merege
     b. hoek.applyToDefaults
  2. lodash
     a. lodash.defaultsDepp
     b. loadash.merege
     c. lodash.mergeWith
     d. lodash.set
     e. lodash.setWith
- Never let the path argument be user-input unless the user-input is whitelisted
- example where you define a object, function , where the, the “constructor” or “proto” are the two prototype which points to the prototype of the class
- example as below

```
fucntion TESTPAYA() {

}
TESTPAYA.prototype.myfunction = function() {
  return X;
}

var inst = new TESTPAYA();
inst.constructor                          //this will return the function
TESTPAYA
inst.constructor.prototype                //returns the prototype of TESTPAYA
inst.constructor.prototype.myfunction()   //return X

OR

var inst = new TESTPAYA();

inst.__proto__                             // returns the prototype of
TESTPAYA
inst.__proto__.myFunc()                    // returns X
inst.__proto__ = { “a” : “123” };           // changing the prototype at
runtime.
inst.hasOwnProperty(“__proto__”)            // false. We haven’t redefined the
property. It’s still the original getter/setter magic property
```

- For mitigating the prototype Freeze the porotype, means One of the API introduced “object.freeze”, when that function is called in an object, any further modification on that object will silently fail.
- mitigation.js

```
fucntion TESTPAYA() {

}
TESTPAYA.prototype.myfunction = function() {
  return X;
}

var inst = new TESTPAYA();
inst.constructor                          //this will return the function
TESTPAYA
inst.constructor.prototype                //returns the prototype of TESTPAYA
inst.constructor.prototype.myfunction()   //return X

OR

var inst = new TESTPAYA();

inst.__proto__                             // returns the prototype of
TESTPAYA
inst.__proto__.myFunc()                    // returns X
inst.__proto__ = { “a” : “123” };           // changing the prototype at
runtime.
inst.hasOwnProperty(“__proto__”)            // false. We haven’t redefined the
property. It’s still the original getter/setter magic property
```

- For mitigating the prototype Freeze the porotype, means One of the API introduced “object.freeze”, when that function is called in an object, any further modification on that object will silently fail.
- mitigation.js

```
- Object.freeze(Object.prototype);
- Onject.freeze(object);
- ({})._proto_.test = 123
- ({}).test //undefined
```

- Use Null Object Object.create(null)
- It is possible to create an object in javascript that do not have any Prototype. Use “Object.create” function.
- Object created through this API won’t have the “proto” and “constructor” attributes. Creating an object in this fashion can help mitigate prototype pollution attacks.

```
- var obj = Object.create(null);
- obj.__proto__ // undefined
- obj.constructor // undefined
```

- Use Map instead of object
- The Map primitive was introduced in ES6. The Map data structure stores key/value pairs and it is not susceptible to Object prototype pollution. When a key/value structure is needed, Map should be preferred to an Object.
- Require schema validation of JSON input
- Avoid using the unsafe recursive merge function
- Upgrade lodash to version 4.17.11 or higher.

you can read more for the prototype Pollution more from the research paper of Olivier Arteau's

SUMMARY

This concludes the end of the e-book, where we went over the best practices of Secure Coding of top programming languages like Python, Java & JavaScript.
Our Secure Code Wiki contains other programming languages in a user-friendly portal. Do bookmark it and practice, practice, practice. You can look through our other resources to stay updated on Cyber Security Industry.

We hope this e-book gave you valuable knowledge and insight into the world of Cyber Security.

About Payatu

Payatu is a Research-powered cybersecurity services and training company specialized in IoT, Embedded Web, Mobile, Cloud, & Infrastructure security assessments with a proven track record of securing software, hardware and infrastructure for customers across 20+ countries.

Our deep technical security training and **state-of-the-art** research methodologies and tools ensure the security of our client's assets.

At Payatu, we believe in following one's passion, and with that thought, we have created a world-class team of researchers and executors who are willing to go above and beyond to provide best-in-class security services. We are a passionate bunch of folks working on the latest and cutting-edge security technology.

Payatu Security Consulting Pvt. Ltd.

www.payatu.com
info@payatu.com
+91 20 41207726