National University of Computer and Emerging Sciences

# SSD Lab – Fall 2025

# Cyber Security Department – CY-B

# Lab 08

## Secure Coding Practices

### Instructor: Mr. Usman Naeem

In this lab you are expected to learn the following:

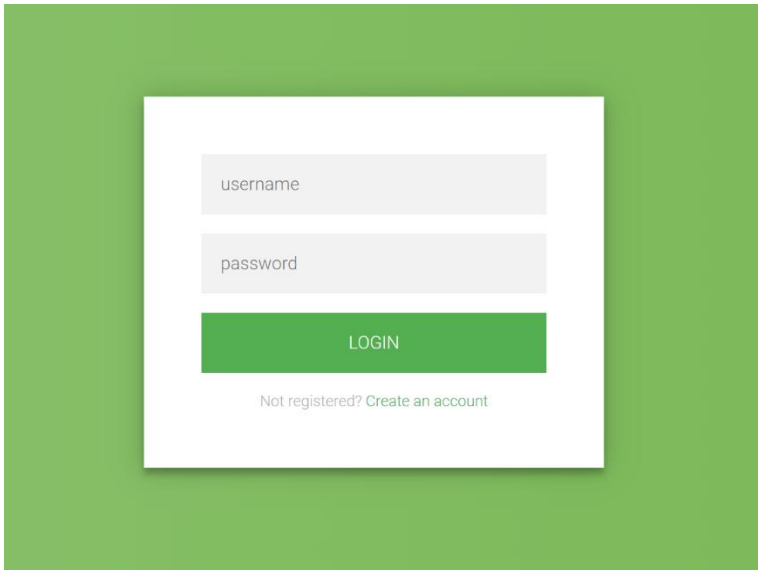
- Implement secure coding practices from the OWASP Checklist in a basic Flask CRUD application.

## Prerequisites:

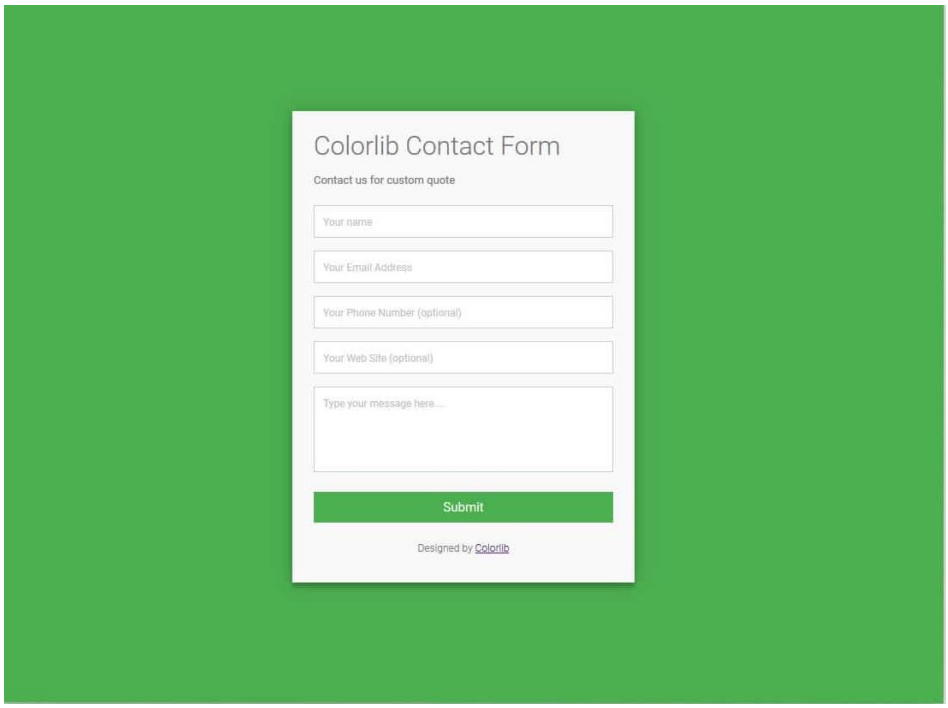
- Basic Flask CRUD application with routes, templates, and models.
- Familiarity with Python and Flask.
- Knowledge of OWASP Top 10 vulnerabilities.

**NOTE:** To perform this task, you should have the following pages in your FLASK APP, if you don't have them already make them and also connect them to the data base, they should be fully functional.

### A Basic Login Page

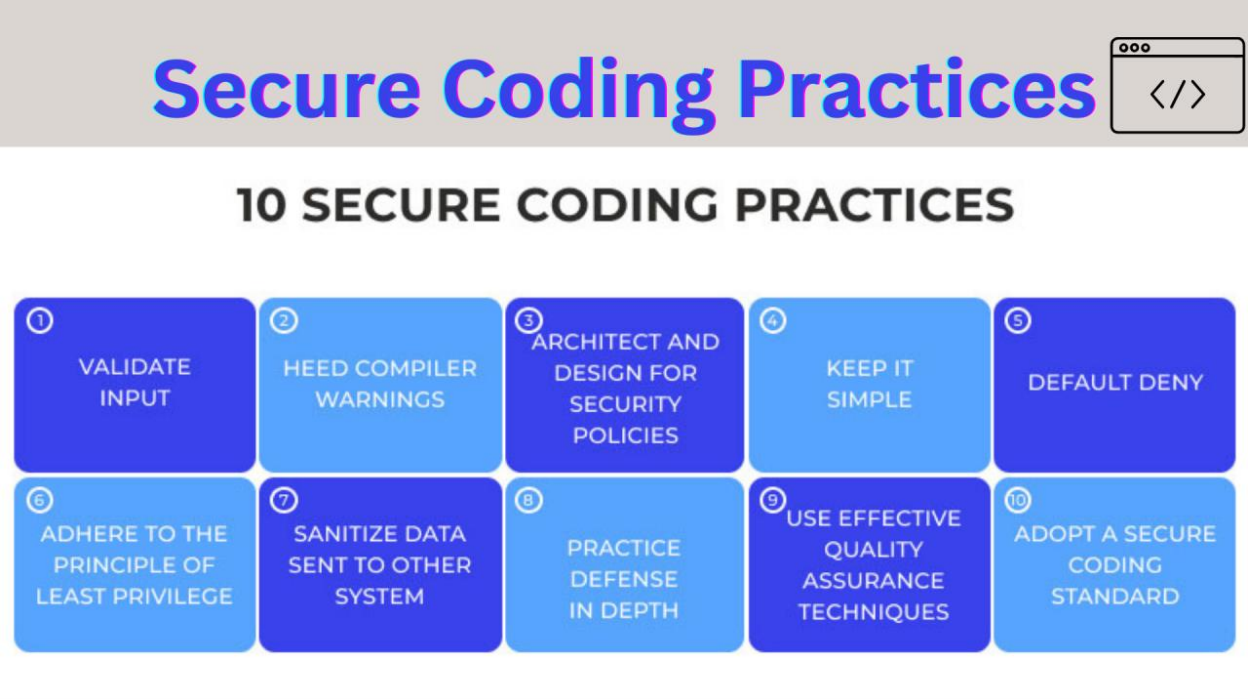


**A page that takes all the contact details of a user in a form:**



## What is Secure Coding?

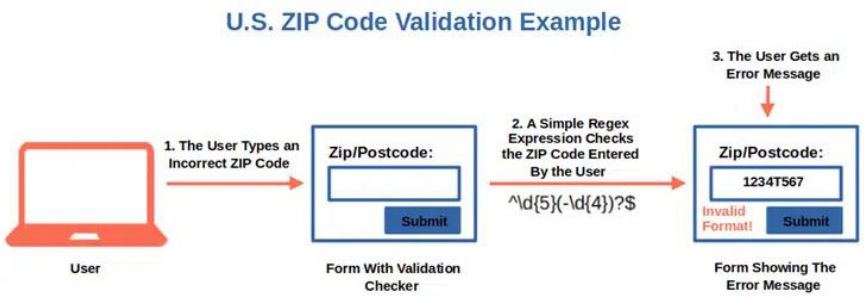
Secure coding standards govern the coding practices, techniques, and decisions that developers make while building software. They aim to ensure that developers write code that minimizes security vulnerabilities. Development tasks can be solved in many different ways, with varying levels of complexity. Some solutions are more secure than others, and secure coding standards encourage developers and software engineers to take the safest approach, even if it is not the fastest.




## 10 Secure Coding Best Practices

OWASP provides a secure coding practices checklist that includes 14 areas to consider in your software development life cycle. Here are the top **10 Secure Coding Practices You Can Implement Right Now.**

# 1. Input Validation: Never Trust User Input




Securing user input is a big deal for any application. It's basically the first line of defense. Get this under control and you'll easily avoid most vulnerabilities. And it doesn't matter where the input comes from: trusted or untrusted sources, everything has to be validated. Why? Because everyone makes mistakes and not everyone has good intentions.

Including input validation in your code will enable you to check that the input entered by the user (or attacker) is how it should be. If not, the data will be rejected, typically while displaying a 400 HTTP error message. Checks generally are based on:

- Data length (e.g., telephone numbers).
- Characters set (e.g., email addresses).
- Format (e.g., dates).
- Restrictions (e.g., no code or scripts allowed).

Have you ever seen an error message like "Please enter a valid email address" when you entered your email address on a web form and forgot to type the "@" special character? This is the result of implementing input validation. Because being trusting is good when it comes to friends and family, but it's better to not trust when it comes to security.

Looking for some input validation coding examples? Check out the following resources:

- GitHub's blog article on input validation
- OWASP's input validation cheat sheet
- RedHat's input validation video

## 2. Manage Authentication and Password


Managing authentication and passwords can be tricky, but there are ways to ensure that your data and applications are accessed only by those who are authorized and properly authenticated to do so. Implementing proper authentication and password management processes will make your application less vulnerable to attacks like brute-force and credential stuffing. As such, the following tactics also can help you to mitigate the risks of data breaches:

- **Use transport layer security (TLS) client authentication.** This process involves the server sending its TLS certificate to the client during the TLS handshake process to validate the server's authentic digital identity.
- **Implement proper (i.e., generic) authentication error messages.** If your error message clearly states that a specific username doesn't exist, the user will know it, but it means the attacker will, too. And this will give him an advantage. Never give out too much information. This will ensure that if the authentication fails, the user (or bad guy) will get a generic error message that doesn't provide them with info they can use to try to access the application.

- **Store, control and manage your passwords safely.** Asking your employees to create strong passwords only takes you so far. You also need secure password storage. Storing password hashes instead of plain text passwords in your database and implementing a good password recovery mechanism for users will help keep your sensitive information away from prying eyes.
- **Transmit password data securely.** If the login landing page or other authenticated pages aren't using a strong encryption mechanism like TLS to secure the connection, the attacker will be able to view the unencrypted session ID and get access to sensitive data.
- **Never store credentials within the app's script.** Even if embedding credentials as plain text into codes can be handy during development, it's never a good idea. Why? Because often people forget to remove the credentials before publishing their code on sites like GitHub. This serves up your credentials as a feast to bad guys, and that can have serious consequences for you and your customers. For a real-world example of this type of situation, just look at what happened to Uber back in 2016…

Do you want to know more about authentication and password management? Check the links below to get useful tips and examples:

- OWASP's Authentication cheat sheet.
- Google's suggestions for good authentication and password management.

## 3. Sanitize Data First, Then Send Inputs to Other Systems

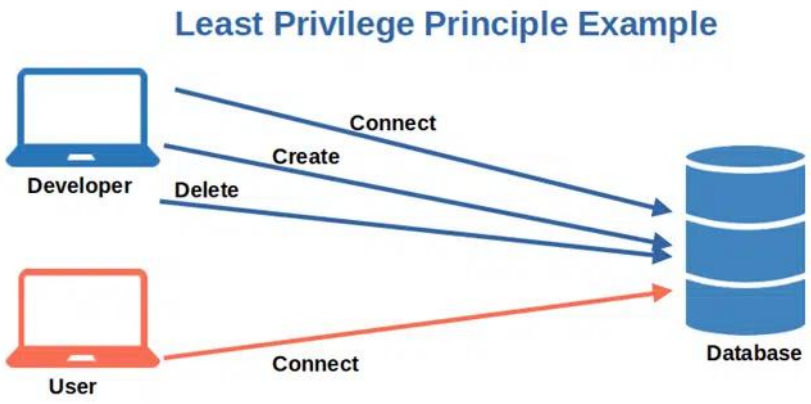

While input validation will check the format and size of the data entered, data sanitation will remove any unsafe characters. It's like a two-step quality control process at a fruit stand: first, the quality and size of the fruits are checked, then all those rotten (unsafe) to ensure that they don't spread mold to the other products.

Likewise, data sanitation will ensure that if a breach happens, it'll be contained and the attackers won't be able to exploit unused functionality for SQL or other injection attacks. How can you do that?

- **Use a whitelist (allowlist).** Including the desired characters or strings in a whitelist will ensure that only those you selected will be retained. Think about telephone numbers. People write them down in different ways: separated by hyphens, dots, or brackets. With a whitelist, you can ensure that only the numbers will be retained, for example.
- **Use a** blacklist (blocklist). It's the opposite of the whitelist. Basically, your list will include all inputs that might pose a threat to your system, enabling you to remove unwanted HTML tags or non-conforming inputs. This can help you detect and stop some of the more obvious input-based attacks, even if keeping the list updated can be a challenge.

- **Escape to keep your code and system safe.** Probably the best solution among the three, you won't have to bother with long lists of wanted or unwanted input here. Based on the assumption that every input is unsafe (do you remember? Never trust user input), it takes the data and secures it prior to putting it in a query. How? Essentially, by adding a special character (e.g., \\ or \\n and more) before a string or character to ensure that it's interpreted as you want (e.g., as text and not as the end of a string). This process is known as escaping.

# 4. Adopt the Principle of Least Privilege (PoLP)



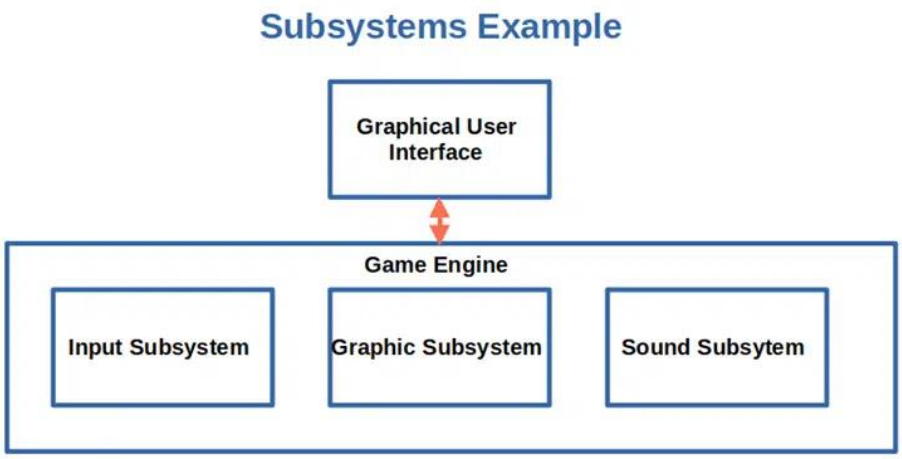

Access to secure facilities is strictly regulated in highly secure facilities like banks, government offices, or military installations. Access is limited to the minimum to reduce risks and to help avoid breaches. The same approach should be applied to applications: a customer service agent shouldn't have access to the company's payroll data, and your chief financial officer doesn't need access to your network's active directory or SSH portal.

Every process should be executed using the minimum privileges necessary to complete. Additional permissions should be only granted for the time taken to complete a specific task reducing the chance of an attacker exploiting those permissions. How can you achieve this?

- **Validate permissions on every request.** .NET Core and Java filters are a good way to perform permission checks correctly.
- **Create tests to validate permissions before release.** This will help you ensure that all permissions set up in the design phase have been applied correctly.
- **Periodically review permissions.** After the app has been deployed, plan and execute regular permission reviews. People change job roles and, when they do, they might not need the same permissions.

Learn more about the principle of least privilege

# 5. Architecture: Make It Unique and Secure




Do you want a secure application? Think and act like an Egyptian pyramid architect from ancient times. To keep their tombs safe from treasure hunters, they were always coming up with new ways to make their architecture unique and challenging for unwanted intruders to figure out.

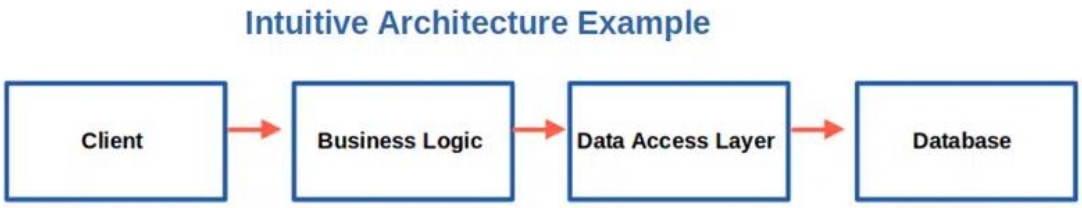
Therefore, to make an invulnerable application, don't copy an old architecture; design your own and add your own security policies. We get it; doing this can be difficult, but a good architecture doesn't have to be overly complicated as long as it's well thought

through. With the Egyptians, it worked so well that some treasures and tombs still remain uncovered today.

But how do you go about creating a unique infrastructure?

- **Use subsystems.** If you need several levels of privileges, ensure you divide your system into subsystems, each with a distinct privilege set and able to communicate with the others.
- **Follow OWASP' secure architecture (SA) practice.** Stuck on your design? OWASP's SA resource can guide you through the architectural design of your application.
- **Load only secure plugins and libraries.** Use only secure directories to avoid an attacker tricking your users into downloading malicious code that may be then loaded and executed by your application.

# 6. Keep Your Architecture Intuitive But Effective

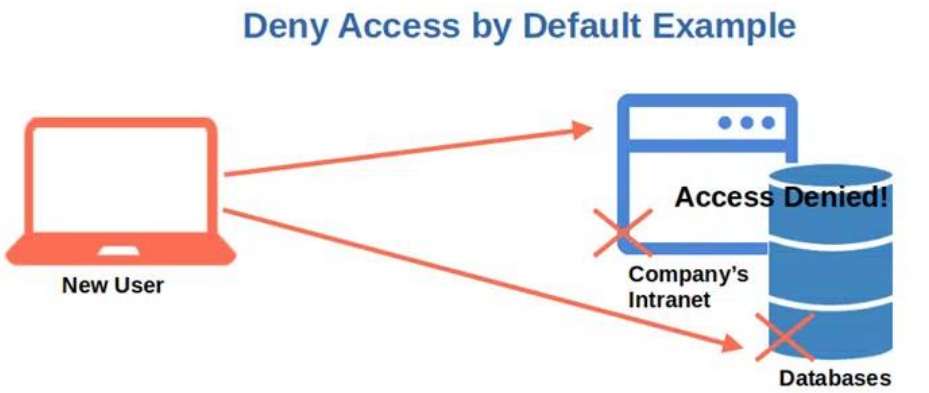



Wow! You've really impressed your audience by showing a demo of your latest, full of cool features application. Even your boss was fascinated. However, while all those bells and whistles may have looked nice during your presentation, you may have forgotten to consider that an application with a simple design is much easier to manage and secure.

Because the more complex the design is, the more likely it is that errors will be made during implementation, configuration, or usage. The same goes for securing the application: the more complex it is, the more time you'll have to invest in implementing security mechanisms.

- **Use a no-frills and small design.** This has been proved efficient in reducing the number of flaws by J. H. Saltzer and M. D. Schroeder's study, "Protection of Information in Computer Systems"
- **Don't over-complicate your security controls.** Overly complex security controls that are difficult to maintain can harm the system administration of your infrastructure, lowering the security of your system.
- **Be user-friendly**. Once again, an overcomplicated design will be reflected on the users that will be tempted to circumvent the security control. Not good.

# 7. Deny Access by Default




Configuring your application to make the access decision based on permission rather than exclusions is much more efficient and secure. Why? Because maintaining a list of exclusions is much more time-consuming and prone to errors than giving permission to a user when needed.

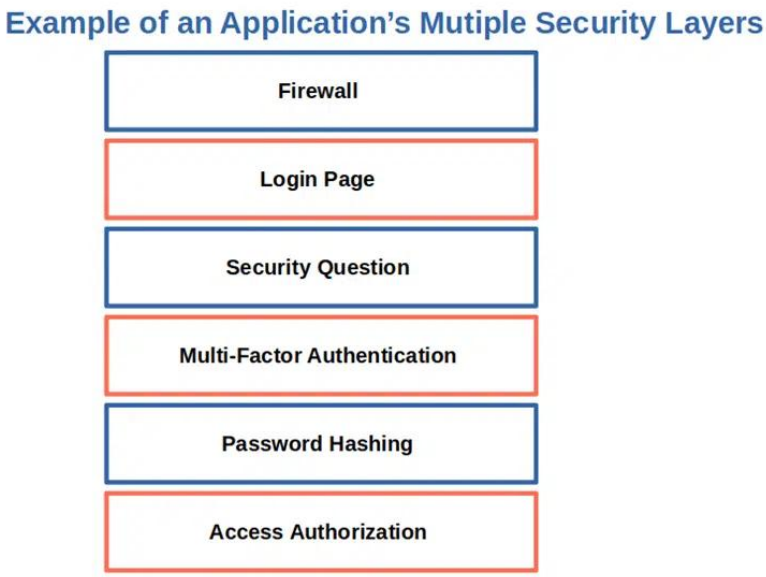
This may leave you wondering what this process entails. Configuring your app so it's permission-based helps you:

- **Keep unauthenticated users out.** No unauthenticated users should be able to access your application's admin page.

- **Apply this policy to new user accounts too.** Do you have a new colleague? Ensure that when they get their new ID and password, they can't access any sensitive systems by default until their account is properly configured.
- **Keep new features sealed.** Have you just released a new feature? Make sure no one can access it until it's properly configured.

To learn more about it, check OWASP's proactive control list.

## 8. Go Deep with Your Defense: Create Multiple Security Layers

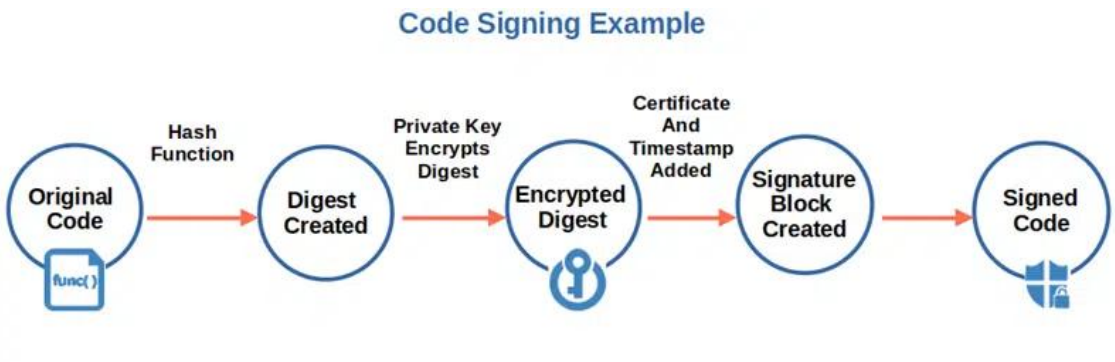



Did you know that the U.S. Transportation Security Administration (TSA) has implemented 20 layers of security to secure its aviation transportation system? Why? Damage control: If one layer fails, the vulnerability may be contained or even stopped by the next one. It's like when a bad guy manages to go through the check-in process using a counterfeit passport and then he's stopped at the gate by law enforcement officers who were alerted by Customs. The same can be applied to application development.

There are a few ways you can create layers within your organization's IT systems:

- **Configure the security settings of each application.** One size doesn't fit all. An application that connects to the internet needs more sophisticated protections compared to one that doesn't connect to the internet.
- **Pair secure programming with secure runtime environments.** This winning combination will help you reduce the risk of undetected vulnerabilities that could be exploited once the code is released.
- **Don't forget authentication checkers.** Once the application is released, if an attacker manages to find a way to get through an unpatched vulnerability, the authentication checks you've implemented may save your day (and your business).

# 9. Secure Your Work and Communication

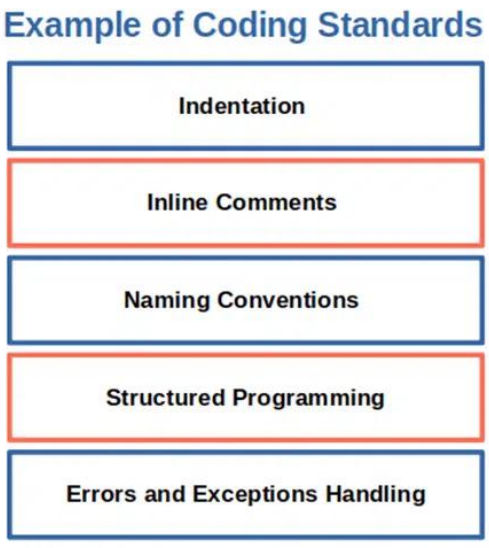



Protecting your work is paramount when you've invested so much time and effort in coding and developing a new app. To do so, you don't need to reinvent the wheel. Take advantage of built-in security features and known techniques such as:

- **Using strong, readily available cryptography.** Ensure you encrypt your data both while it's in transit and in storage (i.e., secure your data at rest). Always be sure to use strong and valid encryption and hashing algorithms to secure your data and protect its integrity.

- **Hardening your database.** Your database is precious and even a small incident (e.g., code injection or table dropping) can have serious consequences. If reading access is enough, never give write access; also be sure to parameterize your queries and close connections as soon as possible.
- **Using trusted security certificates.** Always use digital security certificates (think SSL/TLS certificates for web apps) released by a well-known certificate authority (CA). Ensure that they're properly configured and keep an eye on their expiration dates.
- **Signing your code before releasing it.** Use a code signing certificate to digitally sign your app before releasing it to prevent tampering. Your digital signature confirms to customers that what they're downloading is safe and authentic.

# 10.Check the Quality of Your Code and Follow Coding Standards




Attackers are just waiting for you to make a mistake so that they can jump on the opportunity and use it to their advantage. But errare humanun est! In other words,

making mistakes is human nature and vulnerabilities evolve all the time. However, there are ways to find errors and fix them before they can be exploited.

- **Review your code regularly.** Schedule regular code reviews to check for security issues. You can also combine it with automated tools that scan your code (many are free) for vulnerabilities.
- **Use effective quality assurance mechanisms.** Fuzzing and penetration testing, independent security reviews, and code audits are a few examples of good techniques that'll help you identify flaws in your code.
- **Use coding standards developed by international bodies.** Coding standards — for example, SEI CERT coding for JAVA, C++, and Android OS or MISRA C for C programming language — will make your application more secure and help you ensure that no vulnerability is left undiscovered. This will reduce costs and development time. You can also prepare a list of coding guidelines specific to your organization.

# What you have to do for Lab Task?

Take the Flask Application that you built in Lab 7, and implement the below 5 security practices on it (with any technique available out there). Show each step with the help of screen shots, attach them in a WORD file.

## Instructions:

### 1. Secure Input Handling:

**Goal:** Prevent Injection Attacks (SQL Injection, XSS).

- **Task:** Ensure user input is properly validated and sanitized. Update forms to implement input validation (using wtforms or custom validators).
- **OWASP Guide:** Refer to Input Validation checklist.

**Steps:**

1. Identify forms in the application.
2. Add validation for email, names, and any other inputs (e.g., check for SQL keywords in text fields, avoid unescaped characters for HTML).
3. Test by trying to input malicious SQL queries.

### 2. Use Parameterized Queries:

**Goal:** Prevent SQL Injection.

- **Task:** Refactor any raw SQL queries in the application to use parameterized queries (if using raw SQLAlchemy).
- **OWASP Guide:** SQL Injection Prevention

**Steps:**

1. Review any database interactions.

2. Convert raw SQL statements to parameterized queries.

**3. Session Management and CSRF Protection:**

**Goal:** Protect against Cross-Site Request Forgery.

- **Task:** Implement CSRF tokens in all forms and ensure session management is secure.
- **OWASP Guide:** CSRF Prevention

**Steps:**

1. Enable CSRF protection in Flask using Flask-WTF or by manually generating tokens.
2. Review session configuration to ensure secure cookie settings (session.permanent = True, SESSION_COOKIE_SECURE = True).

**4. Secure Error Handling:**

**Goal:** Avoid Information Disclosure.

- **Task:** Customize error pages to prevent exposure of sensitive data.
- **OWASP Guide:** Error Handling

**Steps:**

1. Create custom error pages (404, 500).
2. Ensure error messages do not expose sensitive information.

**5. Secure Password Storage:**

**Goal:** Prevent Weak Password Storage.

- **Task:** Implement password hashing using bcrypt or argon2 for all user credentials.

- **OWASP Guide:** Password Storage

**Steps:**

1. Install and configure flask-bcrypt.
2. Ensure all passwords are hashed before storage.

## Deliverables:

- Students should submit the updated Flask project(code) with applied security features.
- A short write-up explaining each feature implemented (how did you come up with this technique) and how it improves security.

This setup balances practical implementation with theoretical understanding of secure coding practices.