A collection of example code that demonstrates how to properly handle security issues and incorporate security best practices into the code.

Input Validation: Never Trust User Input

Input Validation is conducted to make sure that data which is properly formed, will be in the workload, which will prevent ‘twisted’ data from being in the database, which could set off a malfunction of several downstream components. In the data flow, input validation should occur as early as it can. There are two levels which input validation is applied on : syntactical and semantic. The first level, should provide the right syntax of organized fields. The second should impose that the values in their certain context should be correct.

Example Code

def validate\_phone\_number(phone\_number):

if len(phone\_number) != 10:

return False

for digit in phone\_number:

if not digit.isdigit():

return False

return True

while True:

phone\_number = input("Please enter your phone number: ")

if validate\_phone\_number(phone\_number):

print("Phone number is valid.")

break

else:

print("Invalid phone number. Please enter a 10-digit number.")

In this code, we use a function : ‘validate\_phone\_number()’ which takes a string, as the input and then it checks whether the phone number entered is a valid number or not. The function has the length checked so that it’s exactly 10 numbers and that each character entered for validation is a digit and not letters.

With the while loop, it will make the user keep on entering a phone number until a valid number is entered, which is based on the function explained above. Once the input entered is valid, then the loop will break and program will output ‘ Phone number is valid’. If it isn’t valid this is what will be outputted ‘ Invalid Phone Number. Please enter a 10-digit number’.

Manage Authentication and Passwords

When it comes to authentication and passwords, it can be difficult to integrate them especially since hackers are getting smarter with bypassing authentication, however there are ways to make sure that your information and applications are only seen by users who are authorised to do so. By efficient implementation of authentication and password management, your information will be more protected against malware attacks like brute force. The following methods are able to aid you in lower the chances of your information being breached :

* **Use TLS (transport layer security) client authentication.** This procedure involves the server sending its [TLS certificate](https://codesigningstore.com/what-is-a-digital-certificate) to the client during the TLS handshake process in order to see if the server’s authentic digital identity is confirmed to be true.
* **Implement sensible authentication error messages.** When you enter credentials and if they aren’t correct, having an error message saying ‘ The following credentials don’t exist’, then the attacker will know that a certain username doesn’t exist, and what this does is that it will give them an edge over you by you giving then too much information. Use an error message, which doesn’t provide them with information, which they can use to their advantage to exploit the application.
* **Store, control and manage your passwords safely.** By asking your employees to have strong passwords won’t get you very far. Having a safe password storage will get you even further. In your database, instead of plan text using password hashes will level up your security of your password and keep users who are trying to see the passwords away from the database.
* **Never store credentials within the app’s script. It can be incredibly easy putting the credentials in plain text during app development, however when it is live people often forget to remove the credentials from the script making it easily accessible to users who have ill intention to steal the credentials for exploitation purposes and the customers and the organization will be impacted heavily.**

**Example Code**

import getpass

# Define a dictionary to store usernames and passwords

users = {

"Alice": "password123",

"Bob": "password456",

"Charlie": "password789"

}

# Define a function to validate a user's login credentials

def authenticate\_user(username, password):

if username in users and password == users[username]:

return True

else:

return False

# Prompt the user to enter their username and password

username = input("Enter your username: ")

password = getpass.getpass("Enter your password: ")

# Attempt to authenticate the user

if authenticate\_user(username, password):

print("Authentication successful!")

else:

print("Authentication failed.")

In the code, we used the getpass module so that we can prompt the user to enter their password without it appearing on the terminal. The ‘users’ dictionary is able to store usernames and passwords as important pairs. We used the ‘authenticate\_user()’ function which will take the usernames and passwords and check to see if they have a match in the ‘users’ dictionary. So when the program starts, the user will be entering their credentials in, the password will be hidden as they type it in, then with the function mentioned above it will see if the credentials entered is correct or not and then output ‘ Authentication Successful’, or ‘Authentication Failed’.

Deny Access by default

By giving permission to users when needed is far more efficient compared to having a list of exclusions since it is time consuming making the list and it is very easy to make errors because you are making it manually. By having your app permission based we are able to:

* **Keep unauthenticated users out.** Users who don’t have permission shouldn’t be able to access admin pages of your application.
* **Apply this policy to new user accounts too.** With new users, you can configure their account so that they cannot access any sensitive information which they aren’t authorised to do so.

**Example Code**

def require\_permission(func):

def wrapper(\*args, \*\*kwargs):

user = kwargs.get('user', None)

if user and user['permission'] == 'admin':

return func(\*args, \*\*kwargs)

else:

raise ValueError('Access denied. You do not have permission to access this resource.')

return wrapper

@require\_permission

def admin\_only\_resource(\*args, \*\*kwargs):

print("This is a resource that only admins can access.")

user1 = {'name': 'Alice', 'permission': 'user'}

user2 = {'name': 'Bob', 'permission': 'admin'}

try:

admin\_only\_resource(user=user1)

except ValueError as e:

print(str(e))

try:

admin\_only\_resource(user=user2)

except ValueError as e:

print(str(e))

In this code, we used the ‘user\_permissions’ dictionary so that the permissions of each user is stored as a string. Then the ‘check\_permission()’ function will take the username and a permission and then validate whether that user has access to that permission it is asking for. When the program runs, if the user does not have access then ‘'Access denied. You do not have permission to access this resource’, will be outputted. If it shows that they do have permission, then ‘Access Granted’, will be outputted. However by default, users do not have any permission unless they have been specifically granted which is ‘Denying Access By Default’.

Go Deep with your defense: create multiple security layers

* **Configure the security settings of each application.** One size doesn’t fit all. An application requires a more complex approach to their protection, especially if they are being connected to the internet, in comparison to an application not connecting.
* **Pair secure programming with secure**[**runtime environments**](https://www.techopedia.com/definition/5466/runtime-environment-rte)**.** 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**](https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/defense-in-depth)**.** When an application is released, with the authentication checkers, it will save your life if a hacker goes through an unpatched vulnerability, since they will be able to void his actions

**Example Code**

import hashlib

# Define a function to hash a password using SHA-256

def hash\_password(password):

return hashlib.sha256(password.encode()).hexdigest()

# Define a dictionary of user credentials

user\_credentials = {

"Alice": {

"password\_hash": hash\_password("password123"),

"two\_factor\_enabled": True,

"two\_factor\_secret": "ab12cd34ef56"

},

"Bob": {

"password\_hash": hash\_password("password456"),

"two\_factor\_enabled": False

},

"Charlie": {

"password\_hash": hash\_password("password789"),

"two\_factor\_enabled": True,

"two\_factor\_secret": "gh78ij90kl12"

}

}

# Define a function to authenticate a user with a password and two-factor code

def authenticate\_user(username, password, two\_factor\_code=None):

if username not in user\_credentials:

return False

user = user\_credentials[username]

if user["password\_hash"] != hash\_password(password):

return False

if user["two\_factor\_enabled"]:

if two\_factor\_code is None:

return False

# Verify the two-factor code using a separate security layer (e.g. Google Authenticator)

# For this example code, we're just using a hard-coded secret

if two\_factor\_code != user["two\_factor\_secret"]:

return False

return True

# Prompt the user to enter their username, password, and two-factor code (if enabled)

username = input("Enter your username: ")

password = input("Enter your password: ")

if user\_credentials[username]["two\_factor\_enabled"]:

two\_factor\_code = input("Enter your two-factor code: ")

else:

two\_factor\_code = None

# Authenticate the user using multiple security layers

if authenticate\_user(username, password, two\_factor\_code):

print("Authentication successful!")

else:

print("Authentication failed.")

In this code, the function : hash\_password() ‘hashes’ a password using SHA-256 which is used for cryptographic security. Each user’s password is stored using the ‘user-credentials dictionary’. If two factor authentication is activated then that is also stored as well.

The function: ‘ authenticate\_user()’ takes in the username and password and two-factor code, depending on if its activated or not checks if they are valid to allow access. Inside the user\_credentials dictionary, usernames are stored, where it will be checked if the password hash matches.

So, when the program is running the user is required to enter their credentials along with the code of the two factor authentication if they have activated it. The authenticate\_user() function is used to see if the user is validated to have access using the multiple security layers. If the user is validated, then the output is "Authentication successful!" Otherwise, it will output "Authentication failed." By creating multiple security layers, we lower the chances of unauthorized access occurring, making it more likely that only authorized users can access the system.

Check the quality of your code and follow coding standards

**1. Choose Industry-Specific Coding Standards**

Choosing the right coding standard is very important because it depends on the industry and the product it is being implemented on. For example, the standards for coding programs for luxury car brands will be different compared to standards for gaming software. Changing to industry-specific coding standards will make writing correct code which meets product expectations much easier. Writing code that will satisfy the business requirements and end users will make it so much easier to run.

**2. Focus on Code readability**

Readable code is easy to follow and optimizes space and time. Here are a few ways to achieve that:

1. Write as few lines as possible.
2. Use sensible naming procedures
3. Blocks of code, for the same purpose separated into paragraphs
4. Use indentation to mark the beginning and end of blocks of code.
5. Don’t use multiple functions for the same task. It’s a waste of space and time
6. Any repetitive tasks, have them automated to save time
7. It’s much easier to write code in short lines, because we are able to read it better

**3. Turn Daily Backups into an instinct**

Events can occur which can cause data loss, software glitches, damage to the hardware and many more. In order to prevent this, it is important have your code saved online and in another physical location regardless of how big or small the job is because you cannot be arrogant when it comes to the protection of your code.

**Example Code**

# Import the flake8 package

import flake8.api

# Define the file or directory to check for code quality

file\_or\_directory = "example.py"

# Create a StyleGuide object with the desired coding standards

style\_guide = flake8.api.StyleGuide(

select=["E", "W"], # Check for errors (E) and warnings (W) only

max\_line\_length=80, # Enforce a maximum line length of 80 characters

exclude=[".git", "\_\_pycache\_\_"] # Exclude certain directories from checking

)

# Run the StyleGuide object on the specified file or directory

result = style\_guide.check\_files([file\_or\_directory])

# Print the results

if result.total\_errors == 0:

print("Code quality check passed!")

else:

print(f"Code quality check failed with {result.total\_errors} errors.")

In this code, the flake8 package is used to check the quality of the code, applying coding standards. Using the file\_or\_directory variable to specify the file/directory we are checking. Using the StyleGuide object, we can have our own coding standards for our benefit. Using the ‘check\_files() method, we run the ‘StyleGuide’ object, and the result is stored in the variable called ‘result’. Then using the ‘total\_errors’ attribute, we check if there are any errors of the result object. If there aren’t any errors, then the out output is ‘ Code quality check passed’. However, if there are errors then it will show the number of errors that it has found