Python Security Basics

Build Python Applications That Protect Data, Secrets, and Users

Python makes it easy to build scripts, APIs, web applications, automation tools, data pipelines, and backend systems.

But working code is not automatically secure code.

A Python application can function correctly while still exposing:

- passwords
- API keys
- user data
- database credentials
- internal files
- application behavior
- sensitive logs
- system resources

Security is therefore not a feature you add only after an application is finished.

It is a way of making development decisions from the beginning.

A useful mental model is:

```
Secure Python Application
        ↓
Protect secrets
        +
Validate input
        +
Control access
        +
Handle data safely
        +
Manage dependencies
        +
Limit permissions
        +
Fail safely
        +
Keep software updated
```

---

1. What Is Application Security?

Application security means protecting an application and its data from unauthorized access, misuse, modification, or disruption.

Consider a simple Python API:

```
Client
   ↓
Python API
   ↓
Database
```

A secure design asks:

- Who is allowed to access the API?
- What input can the client send?
- Which data can each user access?
- Where are passwords stored?
- Where are API keys stored?
- What happens when an error occurs?
- Which files can the application access?
- Which packages does the application trust?
- What information appears in logs?

Security is not one Python function.

It is a collection of practices across the entire application.

---

2. Security vs Functionality

Suppose an application accepts a filename:

```
filename = input("Enter filename: ")

with open(filename) as file:
    print(file.read())
```

Functionally, this may work.

But should the user be allowed to open **any file on the machine**?

For example:

```
/etc/passwd
```

or another sensitive location?

The important security question is:

> What is this input allowed to do?

Secure development is largely about defining and enforcing those boundaries.

---

3. The CIA Triad

A common security model is the **CIA triad**.

It consists of:

Confidentiality

Only authorized people should access information.

Example:

```
User A should not see User B's private documents.
```

Integrity

Data should not be changed by unauthorized users or processes.

Example:

```
A user should not be able to change another user's account balance.
```

Availability

Authorized users should be able to access the system when needed.

Example:

```
An attacker should not be able to easily exhaust the application's resources.
```

Think:

```
Confidentiality
      +
Integrity
      +
Availability
      =
Security
```

---

4. Never Trust Input

One of the most important security principles is:

> Treat external input as untrusted.

External input can come from:

- HTTP requests
- forms
- command-line arguments
- uploaded files
- environment variables
- APIs
- JSON
- CSV files
- databases
- user-controlled configuration
- messages from other systems

Even if input normally comes from your own frontend, do not assume it is trustworthy.

A user can bypass a frontend and call your backend directly.

---

5. Validation vs Sanitization

These concepts are related but different.

Validation

Validation asks:

> Is this input allowed?

Example:

```
def validate_age(age):
    return isinstance(age, int) and 0 <= age <= 120
```

Sanitization

Sanitization transforms input into a safer form.

Example:

```
username = username.strip()
```

Validation and sanitization are not interchangeable.

If a value is invalid, simply modifying it until it looks acceptable may hide a problem.

---

6. Validate at Trust Boundaries

A trust boundary is where data enters a component you control.

For example:

```
Browser
   ↓
API
   ↓
Service
   ↓
Database
```

The API is a trust boundary.

Validate incoming data there.

Example:

```
def create_user(data):
    username = data.get("username")

    if not isinstance(username, str):
        raise ValueError("Username must be a string")

    if not 3 <= len(username) <= 30:
        raise ValueError("Invalid username length")

    ...
```

---

7. Prefer Allow Lists

An allow list defines what is permitted.

Example:

```
ALLOWED_ROLES = {"user", "admin"}


def validate_role(role):
    if role not in ALLOWED_ROLES:
        raise ValueError("Invalid role")

    return role
```

This is generally safer than trying to predict every bad value.

Think:

```
Allow:
user
admin

Reject:
everything else
```

---

8. Be Careful With Regular Expressions

Regular expressions are useful for validation.

For example:

```
import re

USERNAME_PATTERN = re.compile(r"^[a-zA-Z0-9_]{3,30}$")


def is_valid_username(username):
    return bool(USERNAME_PATTERN.fullmatch(username))
```

But complicated regular expressions can become difficult to maintain and can sometimes cause performance problems.

Use the simplest validation rule that meets the requirement.

---

9. Never Use `eval()` on Untrusted Input

One of the most important Python security rules:

Do not use `eval()` to execute user-controlled expressions.

Dangerous:

```
user_input = input("Enter expression: ")

result = eval(user_input)
```

`eval()` executes Python expressions.

If an attacker controls the expression, they may be able to execute unintended code.

For simple arithmetic, use a dedicated parser or implement only the operations you actually need.

---

10. Be Careful With `exec()`

`exec()` dynamically executes Python code.

Example:

```
exec(user_input)
```

This is extremely dangerous when `user_input` is untrusted.

Avoid dynamically executing arbitrary Python code unless the use case genuinely requires it and the input is fully controlled.

---

11. Avoid Unsafe Deserialization

Serialization converts objects into a storable or transferable representation.

Deserialization reconstructs them.

Python's `pickle` module can execute code during deserialization.

Therefore:

> Never unpickle untrusted data.

Dangerous pattern:

```
import pickle

with open("uploaded_file.pkl", "rb") as file:
    data = pickle.load(file)
```

If the file comes from an untrusted source, this can be dangerous.

For data exchange, safer formats often include:

```
JSON
CSV
```

depending on the data requirements.

---

12. JSON Is Not the Same as Pickle

JSON represents data.

Example:

```
{
  "name": "Hafsa",
  "role": "developer"
}
```

Python can parse it:

```
import json

data = json.loads(json_text)
```

JSON does not provide the same arbitrary object execution behavior as Python pickle.

However, JSON input is still untrusted input and should be validated.

---

13. Protect Passwords

Never store passwords as plain text.

Bad:

```
users = {
    "hafsa": "mypassword123"
}
```

If the database is compromised, the passwords are immediately exposed.

Instead, passwords should be stored using a dedicated password hashing algorithm.

Modern Python applications commonly use libraries such as:

- Argon2
- bcrypt
- PBKDF2 through appropriate libraries/frameworks

The important principle is:

```
Password
   ↓
Password hashing function
   ↓
Password hash
   ↓
Database
```

When a user logs in:

```
Entered password
       ↓
Verify against stored hash
       ↓
Accept / reject
```

You should not decrypt stored passwords because passwords should not need to be reversibly encrypted.

---

14. Password Hashing Is Different From Encryption

These concepts are often confused.

Hashing

Designed as a one-way transformation.

```
password → hash
```

Used for password verification.

Encryption

Designed to be reversible with a key.

```
plaintext → ciphertext
              ↓
              key
              ↓
           plaintext
```

Used when data needs to be recovered later.

Passwords are generally **hashed**, not reversibly encrypted.

---

15. Don't Invent Your Own Password Hashing Algorithm

Do not create:

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

and assume this is sufficient password storage.

General-purpose hashes are designed to be fast.

Password hashing should deliberately make password guessing more expensive.

Use a well-established password hashing library or the password-hashing system provided by your framework.

---

16. Protect API Keys and Secrets

Never hardcode secrets:

```
API_KEY = "sk-secret-value"
```

If this code enters Git:

```
Git repository
      ↓
Potential exposure
```

Instead, use environment variables.

```
import os

API_KEY = os.environ["API_KEY"]
```

For optional configuration:

```
API_KEY = os.getenv("API_KEY")
```

---

17. Use `.env` Carefully

For local development, environment files are commonly used:

```
API_KEY=secret-value
DATABASE_URL=...
```

Then:

```
import os

api_key = os.getenv("API_KEY")
```

Add the local secret file to `.gitignore`:

```
.env
```

Commit:

```
.env.example
```

instead:

```
API_KEY=
DATABASE_URL=
```

This documents required configuration without exposing actual secrets.

---

18. Rotate Exposed Secrets

Suppose you accidentally commit:

```
API_KEY = "real-secret"
```

Deleting the line from the latest commit is not necessarily enough.

The secret may still exist in Git history.

The correct response is generally:

1. revoke or rotate the exposed credential
2. remove the secret from the current code
3. remove it from repository history when appropriate
4. check where else it may have been exposed
5. replace it with a new secret

The key lesson:

> Treat an exposed secret as compromised.

---

19. SQL Injection

Suppose a Python application builds SQL like this:

```
query = f"SELECT * FROM users WHERE username = '{username}'"
```

This is dangerous because user input is being inserted directly into SQL.

Use parameterized queries instead.

For example:

```
cursor.execute(
    "SELECT * FROM users WHERE username = ?",
    (username,)
)
```

The exact syntax depends on the database driver.

The principle is:

```
User input
   ↓
Parameter
   ↓
Database driver
   ↓
SQL
```

Do not construct SQL by concatenating untrusted values.

---

20. Why Parameterized Queries Work

With parameterized queries, the database driver treats the input as data rather than part of the SQL command.

Conceptually:

```
SQL structure
+
User value
```

instead of:

```
SQL structure + user-controlled SQL text
```

This is one of the most important protections against SQL injection.

---

21. Authentication vs Authorization

These terms are different.

Authentication

Answers:

> Who are you?

Example:

```
Log in with username and password.
```

Authorization

Answers:

> What are you allowed to do?

Example:

```
An ordinary user cannot delete another user's account.
```

Mental model:

```
Authentication
      ↓
Who are you?
      ↓
Authorization
      ↓
What can you access?
```

A system can authenticate users correctly while still having broken authorization.

---

22. Never Trust User Roles From the Client

Dangerous design:

```
{
  "username": "hafsa",
  "role": "admin"
}
```

If the server simply trusts this field, a user may attempt to change:

```
"user"
```

to:

```
"admin"
```

Authorization decisions should be based on trusted server-side state.

The server must determine what the authenticated user is allowed to do.

---

23. Principle of Least Privilege

Give each component only the permissions it needs.

For example, if a Python service only needs to read a database table, it should not necessarily have permission to:

- delete every table
- create users
- modify database configuration

Similarly, an application process should not automatically have unrestricted access to the operating system.

Think:

```
Required permissions
       ↓
Grant only those
       ↓
Everything else
       ↓
Denied
```

---

24. File Access Security

Suppose users can request a file:

```
filename = input("Filename: ")

with open(filename) as file:
    return file.read()
```

This can become a path traversal problem.

An attacker may attempt something like:

```
../../secret.txt
```

Instead of directly trusting paths, constrain file access to an intended directory.

For example:

```
from pathlib import Path

BASE_DIR = Path("uploads").resolve()


def get_safe_path(filename):
    candidate = (BASE_DIR / filename).resolve()

    if BASE_DIR not in candidate.parents:
        raise ValueError("Invalid file path")

    return candidate
```

Path validation must be designed carefully for the specific application and operating system.

---

25. Uploaded Files Are Untrusted

If your Python application accepts file uploads, do not assume the file is safe because its filename looks normal.

Consider:

```
profile.jpg
```

The filename does not prove:

- its content
- its MIME type
- its size

- whether it is malicious
- whether it contains unexpected data

Security controls can include:

- file size limits
- allowed file types
- content validation
- safe filenames
- storage outside executable directories
- malware scanning where appropriate

---

26. Never Trust File Extensions Alone

This is weak:

```
if filename.endswith(".jpg"):
    accept_file()
```

An attacker can rename another type of file to:

```
malicious.jpg
```

Extension checking can be one layer, but it should not be the only security control.

---

27. Limit Resource Usage

Security also includes availability.

A user might submit:

```
10 GB upload
```

or:

```
10 million records
```

or a request that triggers extremely expensive processing.

Use limits where appropriate:

```
Maximum upload size
Maximum request size
Maximum number of records
Timeouts
```

```
Rate limits
Maximum processing time
```

This reduces resource exhaustion risks.

---

28. Timeouts Matter

Network requests can hang.

Bad:

```
import requests

response = requests.get(url)
```

Better:

```
response = requests.get(
    url,
    timeout=10
)
```

A timeout prevents your application from waiting indefinitely for a response.

The appropriate timeout depends on the operation.

---

29. Be Careful With External URLs

Suppose a server accepts:

```
url=https://example.com
```

and then fetches it:

```
requests.get(url)
```

This can become dangerous in server-side applications because attackers may attempt to make the server request internal services.

This class of problem is commonly called **SSRF: Server-Side Request Forgery**.

If your application fetches user-provided URLs, consider:

- allow-listing domains
- validating schemes
- blocking private/internal addresses
- restricting redirects
- applying timeouts

- limiting response sizes

The exact controls depend on the architecture.

---

30. Don't Log Secrets

Avoid:

```
  logger.info("API key: %s", api_key)
```

or:

```
  logger.info("User login data: %s", request_data)
```

Logs can be stored in:

- files
- cloud services
- monitoring platforms
- containers
- third-party systems

Sensitive data in logs can become a secondary security problem.

Log useful context without exposing secrets.

---

31. Be Careful With Error Messages

Development:

```
  DatabaseError: password authentication failed for user 'admin'
```

may contain useful debugging information.

Production users generally should not receive internal details such as:

- database hostnames
- stack traces
- filesystem paths
- SQL queries
- secret values
- internal service names

A safer external response might be:

```
  Something went wrong. Please try again later.
```

while the detailed error is recorded securely in internal logs.

---

32. Exception Handling Should Not Hide Security Problems

Avoid:

```
try:
    authenticate(user)
except Exception:
    return False
```

This may hide unexpected failures.

For security-sensitive code, understand which exceptions are expected and which indicate a system problem.

Specific exception handling is usually safer and easier to audit.

---

33. Keep Dependencies Updated

Third-party packages can contain security vulnerabilities.

For example:

```
Your application
      ↓
requests
      ↓
dependency
      ↓
another dependency
```

A vulnerability may exist several levels deep.

Use:

- current supported versions
- dependency scanning
- lock/constraint strategies appropriate to the project
- regular updates

Do not blindly upgrade production dependencies without testing them.

---

34. Know What You Install

Avoid installing packages just because their names look convenient.

Before adding a package, consider:

- Is it maintained?

- Is it widely used?
- Does it actually solve the problem?
- Does it introduce many dependencies?
- Does it have known security issues?
- Is the package name correct?

Typosquatting can cause developers to install malicious packages with names similar to legitimate ones.

---

35. Dependency Confusion and Supply Chain Security

Your application depends on software you did not write.

That means your security also depends partly on:

```
Python
+
Package manager
+
Third-party packages
+
Transitive dependencies
+
Build tools
```

This is called the software supply chain.

Good practices include:

- pinning or constraining dependencies appropriately
- reviewing dependency changes
- using trusted package sources
- scanning dependencies
- removing unused packages

---

36. Keep Secrets Out of Source Control

A secure repository should not contain:

```
.env
private keys
password files
cloud credentials
```

```
database passwords
access tokens
```

Use:

```
environment variables
secret managers
deployment platform secrets
```

depending on the environment.

---

37. Don't Depend on Obscurity

Changing:

```
/admin
```

to:

```
/super-secret-admin-page-92831
```

does not provide authorization.

Attackers can discover endpoints through:

- source code
- documentation
- traffic
- scanning
- leaked information

Security must come from actual access controls.

---

38. Secure Defaults

A secure default means the application starts in a reasonably safe state.

For example:

```
DEBUG = False
```

should generally be the production default rather than:

```
DEBUG = True
```

Other secure defaults can include:

- deny access unless explicitly allowed

- HTTPS in production
- reasonable timeouts
- restrictive permissions
- disabled unnecessary features

---

39. Development vs Production

Security settings often differ between environments.

Development might use:

```
DEBUG=True
local database
verbose errors
test credentials
```

Production should generally use:

```
DEBUG=False
real secret management
restricted permissions
secure transport
controlled logging
```

Do not copy development configuration blindly into production.

---

40. HTTPS and Transport Security

If sensitive data travels between a client and server, use encrypted transport.

For web applications:

```
HTTP
```

is not equivalent to:

```
HTTPS
```

HTTPS helps protect data while it travels across the network.

A Python application itself may not configure TLS directly if it sits behind a web server, reverse proxy, or cloud platform.

The security responsibility still exists at the system level.

---

41. Secure Cookies and Sessions

For web applications, session cookies can contain or represent authentication state.

Security-related cookie settings commonly include:

```
Secure
HttpOnly
SameSite
```

Secure

Send the cookie over HTTPS.

HttpOnly

Helps prevent JavaScript from directly reading the cookie.

SameSite

Controls when browsers send cookies in cross-site contexts.

Frameworks often provide these controls.

Use framework-supported security mechanisms rather than implementing authentication cookies manually unless you have a strong reason.

---

42. Cross-Site Scripting

XSS occurs when attacker-controlled content is interpreted as executable browser code.

For example, a Python web application might store:

```
<script>alert("attack")</script>
```

and later render it directly into a page.

The key defense is to:

- escape untrusted output
- use framework templating safely
- avoid unsafe HTML rendering
- validate content where appropriate

Do not manually concatenate user input into HTML.

---

43. Cross-Site Request Forgery

CSRF occurs when a user's authenticated browser is tricked into sending an unwanted request to a site.

Web frameworks often provide CSRF protection.

Use the framework's established mechanism rather than inventing your own token system without understanding the security requirements.

---

44. Security Headers and Framework Defaults

Web applications can use security-related HTTP headers to control browser behavior.

Depending on the application, examples include:

```
Content-Security-Policy
Strict-Transport-Security
X-Content-Type-Options
Referrer-Policy
```

Modern frameworks and middleware may help configure these.

Security headers are part of the broader web security layer, not a replacement for authentication or input validation.

---

45. Don't Build Cryptography Yourself

Cryptography is difficult to implement correctly.

Avoid inventing:

```
def my_encryption(data):
    ...
```

or creating your own password hashing algorithm.

Use well-reviewed libraries and established protocols.

Examples of standard building blocks include:

```
TLS
AES
RSA
Ed25519
Argon2
```

The correct algorithm depends on the specific problem.

---

46. Randomness Matters for Security

Normal randomness is not always appropriate for security-sensitive values.

Avoid:

```
import random

token = random.randint(100000, 999999)
```

for security tokens.

Use Python's `secrets` module:

```
import secrets

token = secrets.token_urlsafe(32)
```

The `secrets` module is designed for generating security-sensitive random values.

---

47. Secure Tokens

For example:

```
import secrets

reset_token = secrets.token_urlsafe(32)
```

Tokens may be used for:

- password reset links
- email verification
- temporary access
- session-related operations

Tokens should be:

- unpredictable
- sufficiently long
- protected from exposure
- expired when appropriate
- invalidated when appropriate

---

48. Timing Considerations

Security-sensitive comparisons may need constant-time comparison.

For example:

```
import secrets

if secrets.compare_digest(
```

```
    provided_token,
    expected_token
):
    ...
```

This can help reduce timing side-channel risks for appropriate comparison scenarios.

Do not assume every string comparison requires this. Use it where secrets or authentication tokens are being compared.

---

49. Rate Limiting

Authentication endpoints can be targeted with repeated requests.

For example:

```
/login
```

An attacker might send thousands of attempts.

Applications can use controls such as:

- rate limits
- temporary lockouts
- progressive delays
- CAPTCHA where appropriate
- monitoring and alerting

The exact strategy depends on the application and threat model.

---

50. Security Through Layers

Do not depend on one security mechanism.

For example:

```
Input validation
       +
Authentication
       +
Authorization
       +
Parameterized queries
       +
Secure configuration
       +
```

```
Dependency updates
       +
Logging
       +
Monitoring
```

If one layer fails, another may still limit the damage.

This is called **defense in depth**.

---

51. Threat Modeling Basics

Before building a feature, ask:

What are we protecting?

Examples:

```
User accounts
Payment information
API keys
Private resources
```

Who might attack it?

Examples:

```
Unauthenticated user
Malicious authenticated user
Compromised account
Automated attacker
```

What could go wrong?

Examples:

```
Data exposure
Unauthorized modification
Account takeover
Resource exhaustion
```

What controls reduce the risk?

Examples:

```
Authentication
Authorization
Validation
```

```
Rate limiting
Encryption
Logging
```

You do not need a complicated security document for every small script.

Even a few deliberate questions can uncover major risks.

---

52. Security Is About Boundaries

Consider this application:

```
User
 ↓
API
 ↓
Service
 ↓
Database
```

Each arrow is a boundary.

Ask:

```
What data enters here?
Who controls it?
What should be allowed?
What should be rejected?
```

This mindset helps you find vulnerabilities before they become bugs.

---

53. Secure Python Project Structure

A security-conscious Python project might contain:

```
secure-app/
│
├── .gitignore
├── .env.example
├── README.md
├── pyproject.toml
│
├── src/
│   └── app/
│       ├── config.py
```

```
│       ├── auth.py
│       ├── validators.py
│       ├── services.py
│       ├── repositories.py
│       └── security.py
│
└── tests/
    ├── test_auth.py
    ├── test_validators.py
    └── test_services.py
```

The exact structure is not a security control by itself.

Good boundaries and explicit responsibilities are what matter.

---

54. A Security Review Workflow

Before releasing a Python application:

```
1. Identify sensitive data
        ↓
2. Identify trust boundaries
        ↓
3. Validate external input
        ↓
4. Review authentication
        ↓
5. Review authorization
        ↓
6. Review secrets
        ↓
7. Review database queries
        ↓
8. Review file access
        ↓
9. Review dependencies
        ↓
10. Review logs and errors
        ↓
11. Test failure cases
        ↓
12. Deploy with secure configuration
```

Security review should be continuous rather than a one-time event.

---

55. Common Python Security Mistakes

Mistake 1: Hardcoded API keys

```
API_KEY = "secret"
```

Use environment variables or a secret manager.

Mistake 2: `eval()` on user input

```
eval(user_input)
```

Do not execute untrusted Python code.

Mistake 3: Unsafe pickle loading

```
pickle.load(untrusted_file)
```

Do not deserialize untrusted pickle data.

Mistake 4: SQL string concatenation

```
f"SELECT ... {username}"
```

Use parameterized queries.

Mistake 5: Plain-text passwords

Never store raw passwords.

Mistake 6: Trusting client roles

Never assume a request field saying `"role": "admin"` proves authorization.

Mistake 7: Unrestricted file paths

Do not allow arbitrary filesystem access.

Mistake 8: No network timeout

External requests can hang or consume resources.

Mistake 9: Logging secrets

Logs can expose sensitive information.

Mistake 10: Ignoring dependencies

Outdated packages can contain known vulnerabilities.

---

56. Mini Project: Secure Resource API Design

Imagine a Python API for haas.dev resources.

Users can:

```
View public resources
Search resources
Create private resources
Edit their own resources
Delete their own resources
```

Security requirements:

```
Public resources → anyone can read

Create resource → authenticated users only

Edit resource → resource owner only

Delete resource → resource owner or authorized admin

API keys → never stored in source code

Database → parameterized queries

Input → validated

Requests → rate limited where appropriate
```

The important part is not the framework.

The important part is defining the security rules before implementing the endpoints.

---

57. 5 Minute Security Challenge

Take a Python application you have already built.

Look for these five things:

```
1. Hardcoded secrets
2. Unvalidated user input
3. Unsafe file access
4. Broad exception handling
5. Dependencies that may be outdated
```

For each issue, write:

```
Risk:
What could go wrong?

Control:
What should prevent it?

Test:
How can I verify the control?
```

This turns security from a vague concept into an engineering process.

---

58. Exercises

Exercise 1: Secret Management

Take:

```
API_KEY = "123456"
```

Refactor it to use an environment variable.

Exercise 2: Input Validation

Create a function that accepts:

```
username
age
email
```

Validate:

- correct types
- reasonable lengths
- allowed values
- required fields

Exercise 3: Safe File Access

Build a function that allows users to read files only from:

```
uploads/
```

Reject paths that escape the directory.

Exercise 4: SQL Safety

Take a string-built SQL query and rewrite it using parameterized parameters for your chosen database driver.

Exercise 5: Authentication vs Authorization

Design a system where:

```
User A
```

can edit their own resource but cannot edit:

```
User B's resource
```

Write the authorization rule before writing the implementation.

---

59. Mini Quiz

1. Should user input be trusted?

Answer: No. Treat external input as untrusted until validated.

2. What is authentication?

Answer: Verifying who a user or system is.

3. What is authorization?

Answer: Determining what an authenticated user or system is allowed to do.

4. Why should passwords be hashed?

Answer: Passwords should not be stored in recoverable plain-text form.

5. Why should parameterized SQL queries be used?

Answer: They separate SQL structure from user-provided data and help prevent SQL injection.

6. Why should `.env` normally be excluded from Git?

Answer: It may contain secrets such as API keys and passwords.

7. Which Python module is intended for security-sensitive random values?

Answer: `secrets`.

8. Why is `pickle.load()` dangerous with untrusted data?

Answer: Python pickle deserialization can execute arbitrary code.

---

60. Interview Questions

Beginner

1. What is application security?
2. What is the CIA triad?
3. Why should user input be treated as untrusted?
4. What is the difference between authentication and authorization?
5. Why should secrets not be hardcoded?
6. What is `.env` used for?
7. What is SQL injection?
8. Why should passwords not be stored as plain text?

Intermediate

9. What is the principle of least privilege?
10. What is defense in depth?
11. Why is `eval()` dangerous?
12. Why is unsafe pickle deserialization dangerous?
13. What is path traversal?
14. What is SSRF?
15. Why are network timeouts important?
16. What is rate limiting?
17. Why should sensitive data not be logged?
18. What is the difference between hashing and encryption?
19. Why is dependency management part of security?
20. What is a trust boundary?

Practical

21. How would you secure a Python file-upload endpoint?
22. How would you protect a Python API from SQL injection?
23. How would you store API keys?
24. How would you design authorization for user-owned resources?
25. What would you check during a security review of a Python application?

---

61. Security Cheat Sheet

Never do this

```
eval(user_input)
```

```
exec(user_input)
```

```
pickle.load(untrusted_data)
```

```
password = "user-password"
```

```
API_KEY = "secret"
```

```
query = f"SELECT * FROM users WHERE id = {user_id}"
```

Prefer this

Secrets

```
import os

api_key = os.environ["API_KEY"]
```

Secure randomness

```
import secrets

token = secrets.token_urlsafe(32)
```

Constant-time comparison

```
secrets.compare_digest(a, b)
```

Parameterized SQL

```
cursor.execute(
    "SELECT * FROM users WHERE id = ?",
    (user_id,)
)
```

Timeouts

```
requests.get(url, timeout=10)
```

Validation

```
if role not in ALLOWED_ROLES:
    raise ValueError("Invalid role")
```

---

62. The Security Mental Model

When writing Python, ask:

```
WHO
 ↓
is accessing this?

WHAT
 ↓
data are they controlling?

WHERE
 ↓
does that data cross a trust boundary?

WHAT
 ↓
could an attacker make it do?

WHICH
 ↓
control prevents that?

HOW
 ↓
will I test the control?
```

Security becomes much easier when you stop thinking only about individual vulnerabilities and start thinking about **trust, boundaries, permissions, and failure modes**.

Key Takeaways

- Never assume external input is trustworthy.
- Validate data at trust boundaries.
- Prefer allow lists when defining permitted values.
- Never use `eval()` or `exec()` with untrusted input.
- Never deserialize untrusted pickle data.

- Never store passwords in plain text.
- Use established password hashing mechanisms.
- Keep API keys and secrets outside source code.
- Never commit real secrets to Git.
- Use parameterized database queries.
- Separate authentication from authorization.
- Apply the principle of least privilege.
- Restrict filesystem access.
- Treat uploaded files as untrusted.
- Use timeouts for network operations.
- Avoid exposing secrets and internal details through logs or error messages.
- Keep dependencies updated and review what you install.
- Use `secrets` for security-sensitive random values.
- Do not implement your own cryptography.
- Use secure defaults in production.
- Apply defense in depth instead of relying on one security mechanism.
- Think about security before implementation, not only after an incident.

---

haas.dev

https://dev-roast-app.vercel.app

Visit haas.dev for more resources and guides.