SECURE CODING USING PYTHON

CONTENTS

1. **Introduction to Secure Coding**
2. **Most Common Software Vulnerabilities**
3. **High Risk Python Security Vulnerabilities**
4. **Secure Coding Standards**
5. **Useful Tools & Plugins**
6. **Coding Best practices**
7. **Reference**
8. **Introduction to Secure Coding**

Python is a high-level programming language that offers a lot of flexibility to write our code the way we want, which might open up some loopholes if security practices are not being followed.

When building software, writing secure code is essential for protecting sensitive data and maintaining the correct behavior of the software. Writing secure code can be hard and even the best developers can not be 100% certain about security vulnerabilities in their code.

No matter how small the application is and or how good the developers are, there is always a possibility of a security vulnerability that could be exploited.

**What Is Secure Coding?**

Secure coding is the practice of writing software that's protected from vulnerabilities.

**What Is Secure Software?**

Software that has been developed in such a way that it will continue to function normally even when subjected to malicious attacks.

This helps to ensure the security of software by:

* Managing access control
* Providing data protection
* Safeguarding against viruses and other cybersecurity vulnerabilities

**Risks of Insecure Software**

An insecure Software lets hackers in and allows them to take direct control of a device or provide an easy access path to another device.

This can result in:

* Denial of service
* Compromised secrets
* Loss of service
* Damage to the system

1. **Most Common Software Vulnerabilities**

**Broken Access Control**

User restrictions must be properly enforced. If they are broken, it can create a software vulnerability. Untrustworthy agents can exploit that vulnerability.

**Cryptographic Failures**

Sensitive data — such as addresses, passwords, and account numbers — must be properly protected. If it isn't, untrustworthy agents take advantage of the vulnerabilities to gain access.

**Injection**

Injection flaws occur when untrusted data is sent as part of a command or query. The attack can then trick the targeted system into executing unintended commands. An attack can also provide untrustworthy agents access to protected data.

**Insecure Design**

Insecure design refers to risks related to design flaws, which often includes the lack of at least one of the following:

* Threat modeling
* Secure design patterns
* Secure design principles
* Reference architecture

**Security Misconfiguration**

Security misconfigurations are often the result of:

* Insecure default configurations.
* Incomplete or impromptu configurations.
* Open Cloud storage.
* Misconfigured HTTP headers.
* Wordy error messages that contain sensitive information.

**Vulnerable and Outdated Components**

Components are made up of libraries, frameworks, and other software modules. Often, the components run on the same privileges as your application. If a component is vulnerable, it can be exploited by an untrustworthy agent. This causes serious data loss or server takeover.

**Identification and Authentication Failures**

Authentication and session management application functions need to be implemented correctly. If they aren't, it creates a software vulnerability that can be exploited by untrustworthy agents to gain access to personal information.

**Software and Data Integrity Failures**

Software and data integrity failures refer to assumptions made about software updates, critical data, and CI/CD pipelines without verifying integrity. In addition, deserialization flaws often result in remote code execution. This enables untrustworthy agents to perform replay, injection, and privilege escalation attacks.

**Security Logging and Monitoring Failures**

Insufficient logging and monitoring processes are dangerous. This leaves your data vulnerable to tampering, extraction, or even destruction.

**Server-Side Request Forgery**

Server-side request forgery refers to data that shows a relatively low incidence rate with above average testing coverage, and an above-average rating for Exploit and Impact potential.

1. **High Risk Python Security Vulnerabilities**

**SQL Injections (SQLi)**

A malicious user can controls the execution of SQL statements for an application at the backend database server. There are four sub-classes in SQLi:

1. In-band SQL Injection / Classic SQLi
2. Inferential / Blind SQL injection
3. DBMS SQLi
4. Compounded SQLi (Eg: Strom Worn)
   * 1. SQLi with inadequate authentication
     2. SQLi with DDoS attacks
     3. SQLi with DNS hijacking
     4. SQLi with XSS

**Cross Site Scripting (XSS)**

In XSS, a malevolent user can trick any web application to steal stored cookies, saved passwords, and script code that served unsuspecting users of that application.

**Cross Site Request Forgery (CSRF)**

This security vulnerability occurs when a compromised website is forced to perform an action by another logged-in user like clicking on a button. Also, it includes the hacking or logging into of a website with others’ login credentials.

**LDAP (Lightweight Directory Access Protocol) Injections**

This vulnerability occurs when a malicious user inserts/modifies LDAP statements that lead to speculations.

**Command Injections**

Here, a malicious user executes OS commands on a web server by abusing it in order to insert their own commands to gain complete control over the server.

**XPathi**

This occurs when a malevolent user intentionally passes data to a website. They can use that interaction to find out how the data is structured in XML, or they can access secured data that they can’t access normally.

1. **Secure Coding Standards**

Secure coding standards are rules and guidelines used to prevent security vulnerabilities. Used effectively, these security standards prevent, detect, and eliminate errors that could compromise software security.

Below are key secure coding standards.

**OWASP and OWASP Top 10**

OWASP is an international nonprofit organization that educates software development teams on how to conceive, develop, acquire, operate, and maintain secure applications. In addition, the OWASP Top 10 is an annual report of the 10 most critical web application and API security risks.



**CWE and CWE Top 25**

Common Weakness Enumeration is a list of software security weaknesses in software and hardware, which includes programming languages C, C++, and Java. The list is compiled by feedback from the CWE Community. In addition, the CWE Top 25 is a compilation of the most widespread and critical weaknesses that could lead to severe software vulnerabilities.

**CERT**

CERT Coding Standards supports commonly used programming languages such as C, C++, and Java. In addition, for each guideline included in the secure coding standard, there is a risk assessment to help determine the possible consequences of violating that specific rule or recommendation.

**CVE**

CVE is a list of cybersecurity vulnerabilities and exposures found in a specific software product. The list is linked to information from several different vulnerability databases, which allows users to more easily compare security tools and services.

**NVD**

NVD is the U.S. government repository of standards-based vulnerability management data and it is connected with the CVE list and provides additional content, including how to fix vulnerabilities, severity scores, and impact ratings. In order to calculate severity scores, Common Vulnerability Scoring System must be used.

CVSS is an open industry standard for assessing the severity of software vulnerabilities. For each vulnerability, a severity score is assigned.

**DISA STIG**

DISA is a combat support agency that provides IT and communication support to all institutes and individuals working for the DoD. It oversees the IT and technological aspects of organizing, delivering, and managing defense-related information. This includes STIG guidelines, which provide guidance on how an organization should handle and manage security software and systems.

**PA-DSS**

PA-DSS is a global security standard that applies to the development of payment application software.

**IEC 62443**

IEC 62443 is a set of security standards used to defend industrial networks against cybersecurity threats. The set of security standards provides a thorough and systematic set of cybersecurity recommendations.

1. **Useful Tools & Plugins**

Below are few security scanners for finding security vulnerabilities in Python applications.

**Python Taint (PYT) – Static Analysis Tool:** This utility is used for identifying command injection, XSS, SQLi, interprocedural, path traversal HTTP attacks in Python web apps. Python Taint is based on the Control flow graphs, data flow analysis and fixed points that are theoretical foundations built using the Flask framework.

**Tinfoil Security Website Scanner:** Tinfoil is an affordable security scanner for Python & Django that helps find holes in web servers and applications and also tells you the ways to fix them.

**Bandit – AST Based Static Analyzer:** It’s an OpenStack security linter that identifies the common security risks in Python programming. It is distributed using pip. To install bandit from source, we can use the command python setup.py install after downloading the pypi source tarball. You can even access the reports from bandit.

**Pyntch – Static Code Analyzer:** Pyntch (PYthoN Type CHecker) helps in detecting runtime errors such as exceptions, not found attributes and variable type missmatchings. It supports Python 2.x currently. It won’t address style issues like Pychecker or Pyflakes, but it works pretty fast and efficient in scanning thousands of lines within a minute.

**Spaghetti Security Scanner:** Spaghetti an open-source web application security scanner built on Python version 2.7. It detects default files, misconfigurations, and insecure files, and it supports numerous frameworks including Django, CherryPy, CakePHP, and others. The tool is capable of finding attacks like admin panel, cookie security, credit card/email/private IP disclosures, SQL injections, ShellShock, Struts-shock, Apache ModStatus, Anonymous cipher, and others.

**Rough Auditing Tools for Security (RATS):** It’s a free tool that scans languages like C, C++, PHP, Perl and Python and emphasizes the errors that are related to security like TOC (Time of Check), TOU (Time of Use), Buffer overflows and Acunetix. Manual code introspection is still important, but this tool still greatly assists us.

**PyDbgEng – Windows Debugging Engine’s Python Wrapper:** It helps in debugging user mode, kernel mode, software and hardware breakpoints, etc. With the help of this PyDbgEbg, you can do fault injection, fuzzing of applications, and unpacking executables automatically.

**python-ptrace:** It’s an opensource debugging tool that uses ptrace developed and written in Python. Here, ptrace works as an tracer that hands the system calls in Linux, BSD and Darwin.

**vdb / vtrace – Debugger for Exploit Malware Analysis:** Here, VDB refers to a dynamic debugging element; vtrace refers to a platform that’s used in debugging frameworks implemented in Python. Vdb utilizes vtrace.

**Immunity Debugger – Python Penetration Testing Tool: It** uses python scripts and supports Windows with Graphical user interface and command line debuggers.

**Mona.py – Open Immunity Debugger**: This a PyCommand that replaces the pvefindaddr and resolves performance issues.

1. **Coding Best practices**

**Use the Most Recent Major Version of Python**

Python 3 was released date back in 2008 and starting from Jan 1, 2020 the Python Foundation announced that Python 2 will stop receiving security updates or support from the community.

If you are still using old versions of Python below Python 3, then you should start considering how to migrate your codebase to Python 3. Start using Python 3 for your new projects or you leave yourself open to security vulnerabilities.

**Use a Virtual Environment**

When building any Python projects, it’s always advisable to use a virtual environment as it helps to prevent conflict in Python modules and as well as have the same modules both on local and production environments.

Using a virtual environment prevents having malicious Python dependencies in your projects and shipping the same to production by using `pip freeze` to generate requirements.txt. If you have malicious packages in your Python environments, using a virtual environment will prevent having the same packages in your Python codebase since it’s isolated.

To create a virtual environment you can either use Virtualenv or Pipenv which help create isolated virtual environments. Pipenv helps to manage, have a predictable and up-to-date environment.

With Pipenv, you can manage your installations, virtual environments, look through your dependency tree, and scan your dependencies for known vulnerabilities.

You can set up Virtualenv:

pip install virtualenv

virtualenv -p /path/to/python <env\_name>

**Import Packages the Right Way**

When working with external or internal Python modules, you should always ensure you are importing them the right way and using the right paths. We have two types of import paths in Python and they are absolute, relative.

Absolute imports specifies the path of the resource to be imported using its full path from the project’s root folder while relative import specifies the resource to be imported relative to the current location of the project where the import statement is.

/\* Absolute Import \*/

from package1 import module1

from package1.module2 import function1

/\* Relative Import \*/

from .some\_module import some\_class

from ..some\_package import some\_function

Now there are two types of relative imports: implicit and explicit.

Implicit imports does not specify the resource path relative to the current module while Explicit imports specify the exact path of the module you want to import relative to the current module.

Implicit import has been disapproved and removed from Python 3, because if the module specified is found in the system path, it will be imported and that could be very dangerous.

Since it’s possible for a malicious module with the same name to be in a popular open source library and find its way to the system path. If the malicious module is found before the real module it will be imported and could be used to exploit applications that has it in their dependency tree.

To prevent this, ensure you use either absolute import or explicit relative imports as it guarantees you import the real and intended module.

from safe\_module import package, function, class

or

from  ..relative\_module import package, function, class

If you are still using Python 2, ensure you remove the use of implicit relative imports as this as been removed in Python 3.

**String Formatting In Python**

Python has one of the most powerful and flexible methods to format strings and if you are not careful enough while using, you might end up opening up a security vulnerability in your code.

Python3 introduced f-strings  and str.format() as a flexible way to format strings and its actually very interesting. However, this opens up a way for data exploit when dealing with user inputs.

If the application built on Python allows users control of the format string, they can be misused to leak sensitive data. For instance, let’s take a look at the exploit code below:

CONFIG = {

“API\_KEY”: “secret\_key”

}

class User:

name = “”

email = “”

def \_\_init\_\_(self, name, email):

self.name = name

self.email = email

def \_\_str\_\_(self):

return self.name

name = “Toby”

email = “oyetoketoby80@gmail.com”

user = User(name, email)

print(f”{user.\_\_init\_\_.\_\_globals\_\_[‘CONFIG’][‘API\_KEY’]}”)

/\* secret\_key \*/

With this, sensitive global data from a CONFIG dictionary can be accessed via the argument.

However, Python has a built-in string module that can be used to fix and prevent this. Using the Template class from the string module:

from string import Template

name\_template = Template("Hello, my name is $name.")

greeting = name\_template.substitute(name="Tobi")

/\* Hello, my name is Toby \*/

from string import Template

name\_template = Template(“Hello, my name is $name.”)

greeting = name\_template.substitute(name=”Tobi”)

/\* Hello, my name is Toby \*/

The string module is good for handling user inputs and generated data.

## Handle Python HTTP Requests Safely

When building Python project that requires sending HTTP requests, it’s always advisable to do it safely and know the library you are using handles security to prevent security issues.

When using HTTP requests library like Requests, you should not pin the versions down in your requirements.txt has that might install outdated and vulnerable version of the module.

For instance, Requests uses Certifi for handling SSL verification, ensure you are sending it to a non-exploited site. By default, Requests handles the SSL certificate verification and can be disabled if you trust the source.

url = “http://trusted\_url”

requests.get(url, safe=False)

This ensures you are not sending requests to an exploited source that could send back exploited code in the Response headers or body.

So to prevent this ensure you are using the latest version of your HTTP requests library, confirm if the library is handling the SSL verification of the source you sent requests to, if you are using standard library urllib,  you should follow best practices to prevent request smuggling.

## Look Out for Exploited and Malicious Packages

Packages can be very helpful and save you time as you don’t have to re-invent the wheel. Packages can be easily installed through the Pip package.  They offer various benefits like saving time, making your codebase compact and smaller, easier application design and better performance.

Most Python Packages are published to PyPI which serves as a code repository for Python Packages and does not go through any form of security review or check.

This means that anyone out there with malicious thought can easily build and publish a package to PyPI with a malicious code or sometimes publish a package with a similar name to a popular package and imitate the package features.

Double-check each Python packages you are installing and importing to prevent having exploited packages in your code. Also, you can use security tools to scan your Python dependencies to screen out exploited packages.

## Handling Data Deserialization Safely

When handling data deserialization in Python, I’ll recommend only deserializing data from a trusted source as its possible that a malicious arbitrary code could be hidden in the data.

Deserialization process in Python recreates Python objects by reading its representation from a file on disk, network interface or string. The resulting objects contain constructors and methods that are executable.

So if data contains malicious code, on deserialization it could run the code thereby exploiting user data or doing something worse.

To fix it, ensure you are using deserialization packages that ensures the safety of the data in sandbox before fully deserializing the data. One of the best packages to do this is PyCrypto as it securely deserializing your data and prevent the running of arbitrary code.

The same goes for Pickle and YAML data type. Pickle lets you to serialize and deserialize a Python object structure. If you are deserializing a pickled python object structure from an untrusted source, that can result in malicious code execution.

YAML is another type of data type mostly used for data configurations and be handled using the PyYAML package. However if you have a YAML objects with malicious code, using the yaml.load function won’t help but lets you run malicious code if found.

This can be prevented by using the yaml.safe\_load for preventing running malicious code when deserializing YAML data in Python.

## Keep Up-To Date Open Source Vulnerabilities in Your Python Packages

One of the simplest ways to prevent and get rid of open source vulnerabilities is having the latest updates of the open source that already fixed the vulnerability. Open source is a good way for developers and communities with one interest in mind to build, contribute and publish software openly for better use of the community.

However, sometimes there’s a possibility that a security loophole might pop up that could be very dangerous as any software or application using the project may be open up for attacks.

For this reason, open source vulnerabilities are always published as soon as they are discovered and a fix and prevention method are usually rolled out in the next version usually security patch release which should end up in the next major release.

The sooner you have the latest update of the open source package, the better you are secured. Always ensure you are updated with vulnerabilities of the open source package you are using, so as to know when to upgrade to the next version.

**Here are the Python security tips we'll explore**

* Always sanitize external data.
* Scan your code.
* Be careful when downloading packages.
* Review your dependency licenses.
* Do not use the system standard version of Python.
* Use Python's capability for virtual environments.
* Set DEBUG = False in production.

Reference

<https://www.securecoding.com/blog/python-security-practices-you-should-maintain/>

<https://github.com/python/cpython/issues/>

<https://www.perforce.com/blog/kw/common-software-vulnerabilities>

<https://infosecwriteups.com/most-common-python-vulnerabilities-and-how-to-avoid-them-5bbd22e2c360>

<https://www.tripwire.com/state-of-security/vulnerability-management/digging-for-security-bugs-vulnerabilities-in-python-applications/>

<https://www.perforce.com/blog/qac/secure-coding-standards>