



**BRNO UNIVERSITY OF TECHNOLOGY**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF INTELLIGENT SYSTEMS**

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

**SECURE CODING GUIDELINES FOR PYTHON**

POKYNY PRO BEZPEČNÉ KÓDOVÁNÍ - PYTHON

**BACHELOR'S THESIS**

BAKALÁŘSKÁ PRÁCE

**AUTHOR**

AUTOR PRÁCE

**JAN ZÁDRAPA**

**SUPERVISOR**

VEDOUČÍ PRÁCE

**Mgr. KAMIL MALINKA, Ph.D.**

**BRNO 2022**

# Bachelor's Thesis Specification



Student: **Zádrapa Jan**  
Programme: Information Technology  
Title: **Secure Coding Guidelines for Python**  
Category: Security

Assignment:

1. Study the relevant areas of secure coding in Python.
2. Get familiar with standards and methods for secure programming (e.g., OWASP for web applications, NIST 800-160), including existing guidelines and tools.
3. Design comprehensive secure programming guidelines for Python (including examples) covering all relevant areas (the main goal is to create a quality learning tool). Take into account the issue of usable security.
4. Implement the proposed learning tool and evaluate its usability.
5. Design and implement real-world examples of exploits using the selected vulnerabilities.

Recommended literature:

- <https://owasp.org/>
- NIST SP 800-160
- GALANSKÁ, Katarína. Usability of Usable Security Guidelines from IT Professional Point of View. Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Kamil Malinka, Ph.D.

Requirements for the first semester:

- Items 1 to 3.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Malinka Kamil, Mgr., Ph.D.**  
Head of Department: Hanáček Petr, doc. Dr. Ing.  
Beginning of work: November 1, 2021  
Submission deadline: May 11, 2022  
Approval date: November 3, 2021

## Abstract

With the number of cyberattacks and their costs rising, the demand for secure coding also rises. Python is an indivisible part of this problem as the favourite programming language. Many programmers can code in Python, but they can not code securely. Python does not have any official secure coding guidelines, and its educational materials on this topic are insufficient. This thesis aims to inform about the most significant Python coding vulnerabilities and bring solutions to these vulnerabilities. It also aims to raise the public's awareness with the help of new secure coding guidelines and educational tool. The educational tool as a web application should be well arranged and usable for the public. The tool also includes real-life examples of exploits from vulnerabilities explained in the guidelines.

## Abstrakt

S narůstajícím počtem kybernetických útoků a vzrůstající cenou jejich dopadů se zvyšuje také poptávka po znalosti bezpečného programování. Python jako aktuálně nejoblíbenější programovací jazyk se stal nedílnou součástí této problematiky. Spousta programátorů umí Python používat, ale neumí jej používat bezpečně. Tomuto problému nepomáhá ani to, že samotný Python nemá dostatek pokynů a výukových materiálů pro bezpečnostní problematiku. Cílem této práce je informovat o největších bezpečnostních hrozbách programování v Pythonu a zároveň zajistit řešení těchto situací. Zaměření práce je na poučení veřejnosti pomocí výukových materiálů v podobě pokynů a výukové pomůcky. Výuková pomůcka v podobě webové aplikace by měla být přehledná a použitelná pro veřejnost. Součástí aplikace je také několik příkladů implementace zranitelností z reálného světa.

## Keywords

Python, secure coding, programming, security, web security, web application, coding usability, Django, vulnerabilities

## Klíčová slova

Python, bezpečné programování, programování, bezpečnost, webová bezpečnost, webová aplikace, použitelné programování, Django, zranitelnosti

## Reference

ZÁDRAPA, Jan. *Secure Coding Guidelines for Python*. Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Kamil Malinka, Ph.D.

# Rozšířený abstrakt

## Úvod

Jazyk Python je v dnešní době jeden z nejrozšířenějších programovacích jazyků. Programovat se v něm dá v podstatě cokoliv a na jakoukoliv platformu. Zároveň rok za rokem přibývá čím dál více kybernetických útoků, jejichž dopady jsou stále dražší. Jen minulý rok stály kybernetické útoky firmy miliony dolarů a spoustu uživatelů stály ztrátu jejich osobních dat. Jazyk Python je možná bezpečnější než některé jiné jazyky, ale to neznamená, že je bezpečný. Nejproblematičtějším prvkem v řetězci vývoje jsou samotní programátoři, kteří neumí programovat s myšlením na bezpečnost.

Mnoho firem se snaží naučit programátory bezpečnému programování, ale bohužel pro to není dostatek prostředků. Oficiální pokyny neexistují a dostupné výukové pomůcky nestačí. Programátoři tak často používají své naučené metody a postupy, které nemusí být efektivní natož bezpečné. Vzhledem k tomu, že se nové zranitelnosti objevují denně, je vhodné programátory naučit jinému přístupu, který by dokázal riziko bezpečnostní chyby snížit.

Současné materiály jsou většinou ve formě seznamů zranitelností a špatných technik programování, ale to samotný programátor nevyhledává, protože ho k tomu nikdo nepřinutí. Proto je cílem této práce nastudování problematiky a různých zranitelností a informovat o problematice bezpečného programování. Další částí je vytvořit pokyny i pomůcku, kterou by firma mohla používat bez složitého vyhledávání na internetu v seznamech zranitelností na stránkách organizací typu OWASP nebo CVE. Tato práce by se měla používat pro výukové účely pro studenty programování nebo programátorů ve firmách.

## Popis řešení

Samotné řešení se skládá s několika základních bodů. Zaprvé bylo nutné se seznámit se samotným pojmem bezpečného kódování a různých oblastí, kterých se to v jazyce Python dotýká. Při zjištění, že se to týká prakticky všech oblastí bylo třeba se rozhodnout, které části Pythonu využít. Nejprůběžnějšími byly zvoleny následující části: základy (tzn. samotné nastavení Pythonu než programátor vůbec něco začne psát), standardní knihovna, která ukrývá mnoho nebezpečných zranitelností a webové programování, které je v dnešní době velice populární.

Dalším bodem bylo nastudovat si existující standardy a různé výukové nástroje. Z tohoto výzkumu byly patrné dvě věci. První, že nejlepší organizace pro tuto problematiku jsou OWASP a NIST, které pokrývají snad veškeré kategorie, které byly zvoleny v předchozím bodě. Z výzkumu, který se zajímal o existující řešení vyšel následující závěr. Neexistují žádné oficiální pokyny pro bezpečné programování v Pythonu a výukové pomůcky jsou většinou ve formě online materiálů. Z toho vyplynulo řešení praktické části práce, že zvolenou pomůckou bude také webová aplikace, ve které budou samotné pokyny pro bezpečné kódování. Existující řešení jsou spíše praktického rázu, kdežto toto řešení by mělo předat dost teoretických znalostí, které budou poté otestovány v aplikaci. Programátor by si měl odnést to, že jsou bezpečnější způsoby implementace i zdánlivě jednoduchých konstrukcí. Třetí část je samotné napsání pokynů pro bezpečné programování. Jak bylo vyřešeno v bodech výše, pokyny se skládají ze tří částí. Od základů jako aktuální verze Pythonu až po SQL injektování kódu v části třetí. Měly by být pokryté důležité části Pythonu. Co se do pokynů nevešlo je obsaženo ve čtvrté části, která je pojmenována jako další a ob-

sahuje seznam z odkazy na další zranitelnosti. Použitelnost by měla být zakomponována na příkladech, které se snaží být co nejpoužitelnější a kód jako takový co nejjednodušší a nejefektivnější.

Samotná výuková pomůcka je implementována v jazyce Python a frameworku Django s pomocí Bootstrapu pro zjednodušení práce s grafickými prvky. Skládá se ze tří částí. První částí je teoretická část, která obsahuje pokyny vytvořené v předchozím bodě řešení. Druhá část obsahuje testy, které mají ukázat uživateli, zda-li danou problematiku pochopil. Poslední část je datový sklad, který obsahuje všechny příklady, které byly vytvořeny v rámci této práce.

Poslední část práce se týká implementování příkladu ze života s využitím zranitelností uvedených v návodu. Byly vybrány čtyři různé příklady z různých částí Pythonu pro co nejlepší reprezentaci problémů. První implementovaná zranitelnost byla SQL injekce, která byla připravena pomocí další webové aplikace. Tato aplikace se dá rovněž využít jako názorná výuková pomůcka pro uživatele, protože je interaktivní a dostupná ke stažení. Druhý problém byl přiložen k webové aplikaci z předchozího příkladu, kdy se aplikace vylepšila a zamezilo se SQL injekci, ale nastal jiný problém s názvem ReDoS. Třetí z implementovaných zranitelností ve větším měřítku byla poměrně nová zranitelnost, která umožňuje vpisování spustitelného kódu do komentářů. Tato zranitelnost byla implementována pomocí jednoduchého Python skriptu simulující přihlašovací formulář. Jako poslední byla vybrána zranitelnost nalezená v modulu Pickle, která je předvedena na klient-server aplikaci ve frameworku Flask.

## Výsledky práce

Výzkum problematiky ukázal, že téma bezpečného programování ještě často není hlavní téma při vývoji softwaru. Často se spíše řeší uživatelská použitelnost než samotná bezpečnost. Hlavním výsledkem této práce je uživatelsky přívětivá výuková webová aplikace, která obsahuje spoustu teoretických poznatků stejně jako obsahuje závěrečný test, který je schopen potrápít i zkušenějšího programátora. Předpokládaný dopad a cíl této práce je ten, že se minimálně začne o tématu více mluvit a také, že se tato práce bude dát použít jako výuková pomůcka pro školy a nebo firmy.

## Závěr

Chyby programátorů při návrhu a implementaci různých aplikací v jazyce Python stály firmy milióny dolarů. Při výzkumu dostatečnosti výukových materiálů a standardů, které již existují, byly odhaleny nedostatky ve výukových materiálech. Materiály byly zaměřeny pouze na jedno téma a nebo nebyly dostupné zdarma.

Na základě těchto nedostatků byly vytvořeny pokyny, které se nevěnují pouze jedné oblasti jazyka Python, ale rovnou několika z nich. Tyto pokyny i s příklady a uvedená výuková pomůcka, která na tyto pokyny navazuje, by mohla sloužit jako základ pro větší projekt typu oficiálních pokynů pro jazyk Python. Pro jednoho člověka je to až moc práce a není schopen zachytit veškeré aktuální problémy bezpečnosti jazyka Python.

Tato práce byla přihlášena na konferenci Excel@FIT za účelem upozornění na tuto problematiku. Výuková pomůcka je zdarma veřejně dostupná<sup>1</sup>.

---

<sup>1</sup>Dostupná na adrese: <http://secopy.herokuapp.com>

# Secure Coding Guidelines for Python

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mgr. Kamil Malinka Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....  
Jan Zádrapa  
May 3, 2022

## Acknowledgements

I would like to thank my supervisor, Mgr. Kamil Malinka, Ph.D. for his support, advice, and helpful points while working on this thesis.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Secure coding</b>	<b>5</b>
2.1	Definition of secure coding . . . . .	5
2.2	Motivation for this thesis . . . . .	5
2.3	Goal of this thesis . . . . .	6
2.3.1	Requirements of this thesis . . . . .	7
2.4	Security vs. usability . . . . .	7
<b>3</b>	<b>Existing methods and standards</b>	<b>8</b>
3.1	OWASP . . . . .	8
3.1.1	OWASP Top 10 . . . . .	8
3.2	NIST . . . . .	10
3.2.1	NIST for cybersecurity . . . . .	11
3.3	CVE . . . . .	11
<b>4</b>	<b>Existing tools for secure coding in Python</b>	<b>12</b>
4.1	Existing tools . . . . .	12
4.1.1	Codebashing . . . . .	12
4.1.2	Secure Code Warrior . . . . .	13
4.1.3	Avatao . . . . .	13
4.2	Other (not educational) tools . . . . .	14
4.2.1	Static code analysis tools . . . . .	14
4.2.2	Guidelines for other languages . . . . .	15
4.3	Summary . . . . .	15
<b>5</b>	<b>Secure Coding Guidelines for Python</b>	<b>16</b>
5.1	Basics . . . . .	16
5.1.1	Version of Python . . . . .	16
5.1.2	Virtual environment . . . . .	17
5.1.3	Importing modules . . . . .	17
5.2	Standard Python . . . . .	19
5.2.1	Input validation . . . . .	19
5.2.2	Standard library vulnerabilities . . . . .	21
5.3	Web programming . . . . .	25
5.3.1	Python frameworks for web programming . . . . .	25
5.3.2	OWASP Top 10 . . . . .	26
5.4	Others . . . . .	28

<b>6</b>	<b>Well-known vulnerabilities in other languages</b>	<b>29</b>
6.1	Buffer overflow . . . . .	29
6.2	String . . . . .	30
6.3	Integer overflow . . . . .	31
<b>7</b>	<b>SeCoPy (Secure Coding in Python)</b>	<b>32</b>
7.1	Design . . . . .	32
7.1.1	Requirements analysis . . . . .	32
7.1.2	Beta version . . . . .	33
7.1.3	Final version . . . . .	34
7.2	Implementation of SeCoPy . . . . .	34
7.3	Content of the application . . . . .	36
7.4	Security . . . . .	37
7.5	Usability . . . . .	37
7.6	Testing . . . . .	37
7.6.1	Results of testing . . . . .	38
<b>8</b>	<b>Real-life exploits</b>	<b>39</b>
8.1	SQL Injection . . . . .	39
8.1.1	Solution . . . . .	40
8.2	ReDoS . . . . .	41
8.2.1	Prevention . . . . .	42
8.3	Comment code execution - 'Trojan source' . . . . .	42
8.3.1	Solution . . . . .	43
8.4	Pickle module . . . . .	44
8.4.1	Solution . . . . .	45
<b>9</b>	<b>Conclusion</b>	<b>46</b>
	<b>Bibliography</b>	<b>47</b>



# Chapter 1

## Introduction

The popularity of Python programming has been increasing in the latest years. The official Czech Python website says: „It fuels webs and rockets.“ [49]. Thanks to its simplicity and usability, it is clear that Python is a very complex programming weapon, and programmers can create basically everything.

Unfortunately, as the popularity increased, so increased the popularity of cyberattacks. The Covid-19 world pandemic did not help either. The word digitization has been spoken a lot, and demand for web and network services is bigger than ever [50]. Attackers have taken advantage of this situation and started attacking even more. In April 2020, the number of cyberattacks in Switzerland almost tripled. There were about 300 - 350 attacks compared to the average of 100 attacks, says Delloite company website [18]. It also says that half of the people working from home were victims of phishing attacks. This statistic is only one of many examples found on the internet that the need for cybersecurity is enormous. Secure programming is a massive part of this problem.

The big problem is that many people can code very well but do not know how to code securely. Many companies do not care about security at all or care insufficiently. Apps have to be usable first and secure only if there is time left. This approach has to be changed. Many types of standards are provided by organizations such as OWASP, NIST, and others [23, 58, 15]. However, they show that most of the webs and applications have security holes that can potentially be a gateway to some attack.

There need to be some guidelines for secure programming in Python that can be used for educational purposes for schools or companies. The goal of this thesis is to introduce the threats, investigate the problems and solutions, and give secure coding guidelines for Python and educational tools as output.

The second chapter is about the definition of secure coding and why it is important to code securely. It also includes the motivation for this thesis and its main goal. Chapter 3 is about official methods and standards that we can use to learn about the problem. Several methods and standards are essential for developers to be at least aware of. The fourth chapter is about existing tools and guidelines and their analysis. Due to the lack of guidelines for Python, this thesis was possible, but several tools for learning out there are good to mention.

The second part of this thesis includes chapters five onwards. The fifth chapter is essential to this thesis as it consists of the guidelines themselves. The sixth chapter is about other languages' vulnerabilities. Python has already covered some of the vulnerabilities that bother C language. A good programmer should be aware of these as well. Chapter seven involves designing a learning tool called SeCoPy (Secure Coding for Python). It describes

its development and security problems connected to programming a web application. The eighth chapter consists of real-life exploits. There are several dangerous exploits dealt with. Every exploit is described, and the reader is given advice on preventing it. The final chapter is the conclusion. It summarizes why it is important to have an educational tool and the impact of this thesis.

# Chapter 2

## Secure coding

This chapter explains why secure coding is important and defines it to the reader. It also shows the goals of this thesis, such as teaching the user and showing him what the main security concerns in Python are. The last part of this chapter belongs to the comparison between usable and secure coding and why it is a problem for one or the other to be the preferred one.

### 2.1 Definition of secure coding

*Secure coding is a set of practices that applies security considerations to how the software will be coded and encrypted to best defend against cyber-attacks or vulnerabilities [35].*

For those who do not like pure definitions, the more the programmer codes securely, the less chance that some attack can occur. Secure coding is applied throughout the whole Python. It is not only a matter of, for example, web programming, which is nowadays put the most focus on. It is also a matter of standard programming in the standard Python library and other parts of Python as well.

### 2.2 Motivation for this thesis

Despite its popularity and usability, there are still security problems that Python suffers. Python programmers should be informed about possibilities that they could and should use for secure coding in Python. This article shows that in 2018 Python had about 5 % open source vulnerabilities [53]. This study was made using GitHub tracking software and a vulnerability database. Despite it being only 5.45 % and C language, for example, has a vulnerability rate of around 47 %. Still, it is not 0 %, and there are many important projects and web applications that use Python and can be vulnerable.

Other interesting numbers are about users and programmers. These numbers are available here [17].

- 95 % of cybersecurity breaches result from human error
- 85 % of breaches involve a human element
- 61 % of breaches are stolen credentials
- 20 % of employees are likely to click on phishing email links, 5 % of them enter their credentials

Security breaches are really costly, and humans make mistakes. It is possible that users tend to make more mistakes than programmers, but a programmer’s mistake can cause harm not only to him but to the whole company. It is proven that companies with a more automation approach have smaller expenses on data breaches than the companies without this approach [11].

The Figure 2.1 shows the average data breach cost. A reader can see that the cost has been

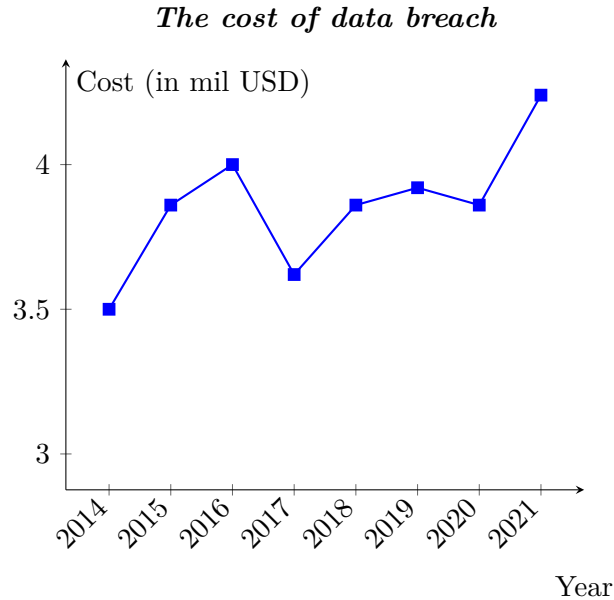


Figure 2.1: Average data breach cost from 2014 to 2021.

about 3.5 million USD in the last eight years. Last year, 2021, the cost has dramatically risen to 4.2 million USD. It is the most in the last 17 years, so security is crucial because it is costly to make mistakes [11].

To show how important secure coding is, it is appropriate to mention an example. Even the biggest websites have server-side (back-end) programmed in Python [41]. A beautiful example would be Facebook, which back-end is partly programmed in Python and suffered a massive data leak [9]. We do not know if the exploit was in Python code, but we know that 533 million users worldwide have their data exposed. E-mail addresses, telephone numbers, birthdays, names and other sensitive data. The question is: Could the attack be avoided? Was the programmer informed about secure coding, and has the programmer code securely?

This concern is the primary motivation for this thesis. Programmers should know about the danger and wrong coding practices, and they should know how to solve these exploits. With Python still on the rise, we can assume that the number of attacks on Python-made apps and systems will be rising too. We will not even notice some of them, but some can hurt even everyone. This is the problem we are facing now.

## 2.3 Goal of this thesis

This thesis aims to raise awareness about Python coding vulnerabilities and bring a solution to some of them. It is not humanly possible to cover all existing vulnerabilities, and

with new technologies, there will be even more vulnerabilities. Nevertheless, programmers should be able to avoid as many of them as possible because damage repair is costly and unnecessary if we can prevent attacks by writing secure code. For example, it is good to point out this latest security discovery that shows that exploits can be executed through code comments [55]. What programmer does not use comments in their code?

### 2.3.1 Requirements of this thesis

The main requirement is to create guidelines that can help with boosting awareness and education about Python's vulnerable modules and techniques. These guidelines should cover as many vulnerabilities and techniques as possible. However, even now, it is clear that there will not be covered everything, such as Python is an enormous language.

Another requirement is to create a secure and usable application that can be used in schools, universities and companies as a learning tool. It should be simple and free. The learning tool should include feedback for the user, such as tests or sandbox. It also should include some examples for the user to try or download and then try on his own.

## 2.4 Security vs. usability

A problem that many developers face is usability versus security. A program or application must be secure because it can contain sensitive data about users. On the other hand, it should be usable because users want to use apps and programs that are easy to use and do not need tons of identification and authorization. How do developers solve this problem?

According to the book, Secure Coding with static analysis [14], developers tend to put usability in front of security. This approach is not ideal, but the line between usable and secure applications is thin. Research on this topic is quite interesting because these two references [32, 39] define the concept of usable security. Usable security has hardly any definition, but it combines security and usability.

A good example where is usability at the cost of security is CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart). This test is globally known, but there are many versions of CAPTCHA. Secure versions are more difficult for the user, and it takes more time to complete them, whereas usable versions are filled even in one click. However, these versions are prone to automation bot tool attacks.

One of the goals is to create a web application that would be secure and, at the same time, usable for users and security will not be noticeable for the standard user. There are some techniques for usable development that will be mentioned in later chapters (Chapter 7).

## Chapter 3

# Existing methods and standards

Several existing standards and methods can be used as a helping hand when developing software. Some are official standards like ISO and NIST, and others are community-based and open like OWASP or CVE [36, 58, 23, 15]. This chapter aims to introduce these standards and explain their importance when developing secure software.

### 3.1 OWASP

The first significant project dealing with secure coding is the Open Web Application Security Project (OWASP) [23]. A community-based project founded in 2001 by Mark Curphey. His main goal was to raise awareness about security breaches in applications by identifying the most critical risks. It is a non-profit organization based in the USA with branches all over the world.

The main goal of OWASP is to inform developers about the most critical security risks in development. For this purpose, OWASP developed several projects, with the most significant one being the OWASP Top 10. The OWASP Top 10 is used as a measurement mainly for web application developers.



Figure 3.1: OWASP logo.

#### 3.1.1 OWASP Top 10

Web application developers widely use the OWASP Top 10 as a guideline when developing a web application. The Top 10 was first published in 2003, and it contains the most dangerous exploits sorted by the percentage of occurrence in applications. The Top 10 is

issued every four years, with the latest edition published in 2021. This chapter introduces the list of Top 10 of 2021 and describes the less relevant vulnerabilities for Python. Some of these more relevant vulnerabilities for Python are thoroughly inspected in the secure coding guidelines in Chapter 5 and SQL injection is even exploited in Chapter 8.

The OWASP Top 10 2021:

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server-Side Request Forgery

### **Insecure design**

This category has almost nothing to do with programming itself. Poorly designed GUI and other design flaws like not having a maximal number on the reservation through the website can cause damage to a company. The insecure design allows an attacker to misuse the website for his benefit or can cause damage. This category also includes automatic access bots [65].

It is appropriate to show an example from the website [65]. The Cinema chain allows book seats without deposit for up to 15 seats. The attacker can book all the cinema seats without paying a single dollar, causing the cinema's massive income loss. The protection against it consists of thoroughly testing misuses of the website and limiting resources that can user consume.

### **Security misconfiguration**

Application is misconfigured when its implementation allows attackers and users to see its vulnerabilities. The application is poorly configured, or some security hardening components are missing. The app contains unnecessary features, test data and accounts. Errors are shown to the user with exceptions from the code. These are a few examples of misconfiguration [66]. The solution is about updating security configuration, deleting test data and default users, and better error handling.

### **Vulnerable and outdated components**

This vulnerability appears when outdated software and components exist in the application. It is essential to know which component the application (client and server) is using, including OS, database, server API, libraries and others. For example, the application uses a module with outdated secure modules. These modules have some vulnerabilities that have been patched. However, the application is not updated, so an attacker can try to exploit this well-known vulnerability and still succeeds only because of an outdated module. The only solution is to update all modules to the latest version. The latest software versions have the most significant chance of being secure and having all vulnerabilities patched [67].

### **Software and data integrity failures**

This vulnerability relies on the integrity infrastructure of a web application. Applications frequently use modules, plugins, sources from other websites and other parts. It is crucial when developing software to have modules updated, but it is needed that these modules have the correct dependencies. One of the possible exploits of this vulnerability is changing the third party module to malicious, and then auto-update service automatically adds this malicious module into the application. The application becomes malicious as a whole. Prevention from this vulnerability is quite simple. Use digital signatures, which can help with keeping the integrity intact. Another prevention is to use only trusted software [69].

### **Security logging and monitoring failures**

This vulnerability is kind of hard to understand. It can not be tested at all, or it is not easy. The problem is that the application should be logging what it is doing — every error, suspicious activity, configuration change, etc. [70] The suggestion is that logging is essential. Every application should have log files that are not stored only locally but on a server because logging can save us much trouble when dealing with security breaches. It is also desirable to have backups in case of the attacker wants to delete or change the logs.

## **3.2 NIST**

The National Institute of Standards and Technology is a US agency and sciences laboratory. It was formed in 1901, and its mission is rising US industrial competitiveness [42]. It is divided into several sectors. Sectors are as follows:

- Communications Technology Laboratory (CTL)
- Engineering Laboratory (EL)
- Information Technology Laboratory (ITL)
- Center for Neutron Research (NCNR)
- Material Measurement Laboratory (MML)
- Physical Measurement Laboratory (PML)

It is enough to know that each sector is creating standards for our purpose.



### 3.2.1 NIST for cybersecurity

There are many sections where is NIST creating standards. One of these sections is cybersecurity. NIST develops standards, guidelines, best practices and others. NIST cooperates with other agencies like NICE (National Initiative for Cybersecurity Education). Their goal is to educate the public about cybersecurity and train developers. They are also aiming at cryptography and cybersecurity measurement [58].

### NIST 800-160

The NIST 800-160's title is Systems Security Engineering. The main goal of this standard is to raise awareness about security problems when developing a system. The standard suggests strategies for how to accomplish a secure system. It is standard for developers and management, but it is comprehensive. It deals with many problems [52].

## 3.3 CVE

The Common Vulnerabilities and Exposures (CVE) is a list maintained by The Mitre Corporation, and it was launched in September 1999 [15]. Since then, almost 173 thousand vulnerabilities have been added to this list. Every vulnerability has its CVE ID, publishing date and update, type, and score. The score sets the severity of the vulnerability, and its scale is 0 to 10, with ten being the most relevant. There is a special list for Python [16], which consists of 68 vulnerabilities. This small number confirms that Python is a relatively secure language. Some of them are even for older versions of Python, but some exceptions can be exploited even in the latest Python versions.

#	CVE ID	CWE ID	# of Exploits	Vulnerability Type(s)	Publish Date	Update Date	Score	Gained Access Level	Access	Complexity
1	<a href="#">CVE-2008-5031</a>	<a href="#">189</a>		Overflow	2008-11-10	2019-10-25	10.0	None	Remote	Low
Multiple integer overflows in Python 2.2.3 through 2.5.1, and 2.6, allow context-dependent attackers to have an unknown impact via a large integer value in the implemented by (1) the string_expandtabs function in Objects/stringobject.c and (2) the unicode_expandtabs function in Objects/unicodeobject.c. NOTE: this v for CVE-2008-2315.										
2	<a href="#">CVE-2016-5636</a>	<a href="#">190</a>		Overflow	2016-09-02	2019-02-09	10.0	None	Remote	Low
Integer overflow in the get_data function in zipimport.c in CPython (aka Python) before 2.7.12, 3.x before 3.4.5, and 3.5.x before 3.5.2 allows remote attacker: value, which triggers a heap-based buffer overflow.										
3	<a href="#">CVE-2010-1449</a>	<a href="#">190</a>		Overflow	2010-05-27	2020-02-18	7.5	None	Remote	Low
Integer overflow in rgbimgmodule.c in the rgbimg module in Python 2.5 allows remote attackers to have an unspecified impact via a large image that triggers a of an incomplete fix for CVE-2008-3143.12.										

Figure 3.2: CVE Top 3 for Python.

## Chapter 4

# Existing tools for secure coding in Python

Python is a widespread language, so there are already many tools that can help programmers code securely. Unfortunately, most of them are just static code analysis tools. It is good to know that other programming languages also have guidelines because some errors are similar to Python. This chapter contains a detailed showcase of Codebashing [13] and Secure Code Warrior [37], which seem like the best tools for interactive learning of secure coding. It also contains mentions about Avatao [4] and static code analysis tools and guidelines. Figures 4.1 and 4.3 have inverted colours for better readability.

## 4.1 Existing tools

### 4.1.1 Codebashing



Figure 4.1: Screenshot from SQL lesson.

The first tool that is good to point out is called Codebashing [13]. Codebashing is a tool developed for developers who want to learn about security problems developed by Checkmarx. The primary purpose of this tool is to teach secure coding techniques in various languages, including Python (Django). The first two lessons are free. For registration, the user has to have a company email. I tested this tool with BUT student email. There is a quiz question at the end of every lesson focused on lesson fundamentals and then the lesson's summary. I think that the main problem with this tool is the paywall after two

lessons.

The first lesson is about SQL injection and its prevention against it, as shown in Figure 4.1. The application shows the problem in a nice illustrative way. There are two people. Alice is our hero, and Bob is the bad guy. The main goal is to repair the code example to state that it is safe. We are also given much helpful information about this type of attack. The first lesson can be seen in the picture above. The developer is taught about input validation and where is executed malicious SQL query. It is the most important basic when working with user forms on a website. The second lesson is about XML external entity attack (XXE attack). The lesson shows that letting users upload XML files can be dangerous.

These first two lines can redirect the &bar to the file with the user's passwords. The lesson

```
<!DOCTYPE foo [<!ELEMENT foo ANY>
<!ENTITY bar system "file:///etc/passwd" >]>
<trades>
  <metadata>
    <name>Apple Inc</name>
    <stock>APPL</stock>
    <trader>
      <foo>&bar;</foo>
      <name>C. K. Frode</name>
    </trader>
```

Figure 4.2: Injecting malicious XML.

itself can explain a lot, so I will not bother to explain what is going on there, but it is a severe vulnerability because system files could be read. Easy defence is to disallow inline DTD (Document Type Definition).

### 4.1.2 Secure Code Warrior

The second educational tool is called Secure Code Warrior [37]. The same name company created this tool. This tool is even more interactive than the Codebashing. There is only a free trial for Python Django, but developers can choose between almost 30 different languages or frameworks. There is Django, Flask, basic Python, and Python API for Python.

The goal is to repair insecure code. The first task for the user is to find a security flaw in the code. Several parts of the code are marked with warning signs, and his goal is to mark the right one. Then when the correct code is marked, the user has to repair the part of the code he has just marked. There are four possible solutions, and he has to choose the right one. There is a theoretical text about the current problem on the left side of the screen. Once this is done, the user can move to the next part.

### 4.1.3 Avatao

The difference between Avatao [4] and the tools above is that Avatao is only for eight languages. Fortunately, Python is one of them. This tool is like Codebashing, but it contains more explainable text, which should be more educational. The tool aims more on

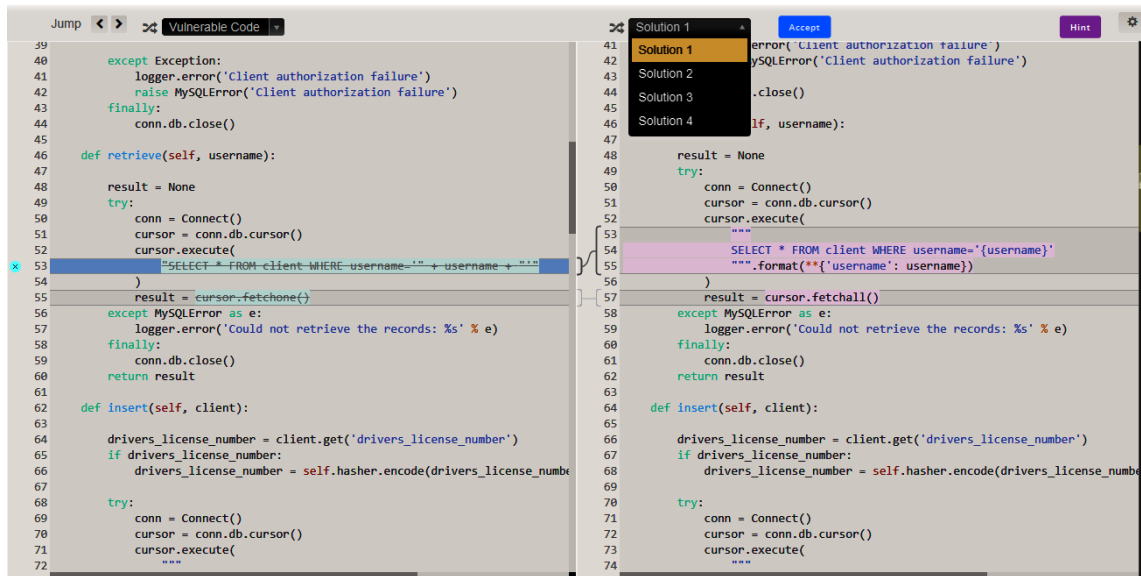


Figure 4.3: Repairing code in Secure Code Warrior.

the standard Python than previous tools. The trial lesson was about input checking, and I appreciate that this concern could be learned for free.

## 4.2 Other (not educational) tools

There are also other tools than only educational ones. Special categories of these tools are static analysis tools. It would be wrong not to mention them because static analysis, even if it is not the best solution, can help prevent many attacks. There are several tools for Python that can help us. Another important part is existing guidelines for other languages. They can be helpful because there are languages with similarities to Python, so it is better than nothing.

### 4.2.1 Static code analysis tools

*Static code analysis is a method of debugging by examining source code before a program is run. It's done by analyzing a set of code against a set (or multiple sets) of coding rules. [8]* Static analysis tools are not interactive and such, but a good static analysis tool can prevent many vulnerabilities. Every tool has OWASP's top 10 vulnerabilities and can recognise them. There is a problem with analysing authentication and access. Also, these tools can not handle every security flaw, so we can not rely on them.

The basic concept of a static analysis tool is checking code for errors and vulnerabilities even before the code is compiled, interpreted or whatever technique uses used language. There are more problems than benefits with static analysis tools (at least for Python). For lower languages, such as is C language, this tool is handy because errors in C are not that complex. These tools can only handle a certain number of security flaws, and there are reports that this tool does not work correctly with the modular program. Also, there are problems identifying security flaws when the problem is not present in the code. That is quite an obvious one, but there are often problems in semantics instead then syntactic

errors. In conclusion, static analysis tools are unreliable. Some examples of good static analysis tools for Python: SonarQube, Codacy, Pylint and Pyflakes [38].

### 4.2.2 Guidelines for other languages

Even if Python does not have official guidelines, some languages already have one. These guidelines can be helpful even when developing in Python. Not in a syntactic way, because every programming language has its unique syntax, but the ways of dealing with some problems are more or less the same. For example, Java has guidelines directly on the Oracle website. It is an enormous file that covers many Java problems [44]. Python developers could want to look into these because both languages are object-oriented, and some problems could have similar solutions. This thesis (Chapter 5) should be used as guidelines for Python.

## 4.3 Summary

The tools are insufficient. Most of these tools are for web application development. They can help prevent OWASP top 10 vulnerabilities, but they can not teach much about generic Python vulnerabilities such as input validation. This work aims to securely teach the user to code web applications and generic Python code. A learning programmer does not know what he will be coding during his professional life. Python development is not only about OWASP. Also, most of the tools are only code analysis tools. Programmers should know what to do and how to code securely and then look for analysis tools.

The other problem is that no guidelines are there. There are only online learning tools, which are even behind the paywall or only for company use. This thesis should fill this space and create a guideline that will teach the programmer about the most dangerous flaws and should be free and available everywhere.

## Chapter 5

# Secure Coding Guidelines for Python

This chapter aims to teach the reader the secure ways of coding in Python with the guidelines. Guidelines begin with basics like using the latest version of Python. The more reader goes into this chapter, the more complex vulnerabilities are introduced. Every section of guidelines includes examples of bad or good practices with an explanation. The guidelines themselves are divided into three different parts. The first one is called Basics and contains basics that every programmer in Python should know. The second part is called the Standard library and contains vulnerabilities, which are part of the Python standard library (no third-party module), plus a couple of efficient and secure coding techniques. The last part is all about web programming and OWASP Top 10 because web programming is very popular these days. The section contains the rest of the OWASP Top 10, which was not mentioned in Chapter 3.

### 5.1 Basics

Some essential basics need to be shown to the programmer. They consist more of the correct Python settings than some coding examples. Setting Python right can prevent many possible vulnerabilities. Such as vulnerabilities that have been patched, for example, in versions 3.0+ such as `input()` [24] vulnerability.

#### 5.1.1 Version of Python

The first thing that a programmer needs to have in mind is the correct version of Python. Many of the vulnerabilities have been repaired in the latest Python versions. For example, in Python version 2.5 exists exploit, where can attacker cause a buffer overflow via string in `socketmodule.c` [43]. Preventing is simple by using the latest stable version of Python. If your company uses Python < 3.0, consider updating the Python version and its dependencies. It is not only more secure, but it is also offering a better user experience. You can check your Python version by typing this command to command line:

```
python3 --version
```

Figure 5.1: Checking the version of Python.

### 5.1.2 Virtual environment

The safest programming in Python should be in the virtual environment. It is recommended to use it because `venv` (virtual environment) can assure that Python projects are divided from the folder with Python itself. If the project has some malicious modules or other security problems, it is limited only to this project and does not affect other Python projects on this device.

When a programmer needs to create requirements for a project, it is essential to know which modules need to be downloaded. Another good practice is using a virtual environment to create files with requirements. When working in the virtual environment, the programmer can be sure that only the modules used for the project will be set as required.

These two commands can set the virtual environment in desired project, and the third connects the user to the virtual environment:

1. `$python3 -m pip install virtualenv`
2. `$python3 -m venv env`
3. `$source venv/bin/activate`

### 5.1.3 Importing modules

There are several methods how to import a module in Python [12]. In particular absolute and relative import and implicit and explicit import. The best way to use import is to use absolute import. Figure 5.2 shows what absolute import looks like. Absolute import

```
from sys import stdin
```

Figure 5.2: Absolute import.

assures that the programmer imports precisely what he wants. Relative import is from Python 3 onwards only explicit. Implicit import has been removed.

It is recommended to use only absolute import or relative explicit import for secure module import. Relative import is used only when the programmer is importing from his local Python files. It is not recommended to use relative import when importing third-party modules as it could import the wrong module because the path to the module is not clear. It also could potentially create problems when changing the project structure.

The second problem with modules is that we do not need everything from the module. We usually need only some of the functions or classes. In Python, exists many ways how to import particular parts of modules. The best way to import a module is to specify class and method by absolute import.

Another way of importing that exists is *from module import \**, but that is an insecure way of importing. The first problem is that we are importing methods that we do not need, and the second problem is that we usually do not know what is in the module. There is a rising probability with the number of modules that we download modules with either malicious methods or overwrite the previous module. In Figure 5.3, there is an example of wrong importing, where importing `time` overwrites module `asyncio`. It is a basic example from the standard library, but it is important to show some basic mistakes. The module `asyncio` is there only for the showing of no overwriting. Both of the modules contain method `sleep()`.

```

from asyncio import *
from time import *

print("Hello " + str(time()))
sleep(1)
print("After " + str(time()))
sleep(2)
print("End " + str(time()))

```

Figure 5.3: Wrong import with overriding.

In the bad example, there is no proof that we are calling `sleep` from `time` or `asyncio`. In this case, it depends on the latest import, which is `time`. Figure 5.4 is a better example, where only what is needed is selected from module `time`.

The last warning on this topic is typosquatting [59]. Typosquatting is kind of an attack

```

from time import time, sleep
import asyncio

print("Hello " + str(time()))
sleep(1)
print("After " + str(time()))
sleep(2)
print("End " + str(time()))

```

Figure 5.4: Importing only what is needed.

that targets heedless programmers. It works like this: the programmer writes an import but makes a typo. Import is still not underlined as a mistake; why? The programmer has just imported a malicious package. Be careful with packages and check after yourself. There are many malicious packages in PyPI [56]. There is an example where the programmer typed `modulee_test` instead of `module_test`. In this example (Figure 5.5), there is called only `print`, but it could be, for example, a malicious module with a system command deleting files instead of `print`.

To highlight the severity of this attack, the programmer names the malicious module as `mod`. If the programmer did not use the abbreviation, he would probably spot the mistake. Later in the code programmer uses only the `mod`, and with this abbreviation, there is a higher possibility of executing code with the malicious module. In all cases, this attack should not be underestimated.

From a usable point of view is the best practice to divide imports into three categories. The first category is standard library imports, the second is third party imports, and the last is local imports. Each category should be sorted in alphabetical order. All things are here in this video [61].



```

import modulee_test as mod
import module_test

#malicious module
mod.test_print()
#secure module
module_test.test_print()

```

Figure 5.5: One typo changes everything.

```

honza$ python3 modules.py
Malicious module!
Safe module!

```

Figure 5.6: Output from the code above.

## 5.2 Standard Python

In this section, I will be dealing with things related to the coding itself. I will show some validation techniques and compare two main Python validating constructions. They are still basic, but their impact on security is much more significant. Then there are vulnerabilities from the standard Python library, from which I have chosen the ones that seemed to be quite important to know about, or they are severe.

### 5.2.1 Input validation

The right way of validating user input can assure that the program will be secure. However, there are many practices on how to check user input. Some of them are better than others. This section shows the right and most secure way of input validating. [45].

#### Blacklisting vs. whitelisting

There are two main ways how to validate input. The first method is blacklisting. Blacklisting means marking some inputs as forbidden. The main problem with this technique is that it is practically impossible to catch every wrong input because there are infinite wrong inputs that users can create. Therefore, it is advisable to use whitelisting.

Whitelisting is a practice when the programmer knows the correct input and checks only the correct input. It is a more secure and usable technique because it theoretically costs fewer conditions and fewer lines to create. It seems this statement is wrong, but blacklisting costs a theoretically infinite number of lines. The most suitable technique for whitelisting is to create a regular expression for the correct input. Regular expressions (regexes) are patterns used to match character combinations in strings [40]. Programmers should use regexes because one right regex can cover all the wrong inputs. There is also an online regex tool that can be helpful [2]. Module *re* assures regular expressions for Python [27]. Python also contains many built-in functions like *isdigit()* that can check if the string is a digit or not.

Now, some examples should be shown. The goal of these examples is to explain the main dif-

ferences between these techniques and the importance of good regular expressions. Regexes are sometimes very hard to code right, so I recommend using the online tool mentioned above [2].

In this example, the input must be the “Hello“ string. Figure 5.7 uses blacklisting and Figure 5.8 uses whitelisting with some regular expression. Which one looks safer? The example is elementary, but it proves the point because using whitelisting, there is only one condition and using blacklisting, there are four conditions, and still, it is not sufficient because the word „Hallo“ can be written on the output.

```
#blacklisting, trying to catch all mistakes
if len(input_text) == 5:
    if not input_text.isdigit():
        if input_text.startswith("H"):
            if input_text.endswith("o"):
                #insufficient...can print Hallo
                print(input_text)
            else:
                print("Wrong ending.")
        else:
            print("Wrong beginning.")
    else:
        print("Error, includes digits.")
else:
    print("Wrong length.")
```

Figure 5.7: Bad practise using blacklisting.

```
import re
#whitelisting with regex
if re.search("^Hello$", input_text):
    print(argv[1])
else:
    print("Wrong input.")
```

Figure 5.8: Good practice using whitelisting with regex.

### Try-except vs. if-else

In Python and other languages, exist two main constructions for conditions. Try-except and if-else are used for flow and condition control. They work with similar logic, but the benefits of using one can be pretty significant. Try-except is ideal when the programmer does not know about all possible errors that can occur. With one conditional block programmer can cover many errors. However, it is not without a cost. If-else blocks are more efficient than try-except blocks, it is recommended to use them when we know possible errors. The try-except block is better to use when there is less possibility of error, but the if-else block

becomes more efficient when the possibility arises [33, 7].

When the programmer is working with files, it is recommended to use the try-except block because when some error happens when opening or closing a file and the try-except block is present, no information is lost. Try-except block works with the system itself. In conclusion, it is safer to use the try-except block when we are unsure about possible errors or when there is little possibility of some error/exception. In the other cases, it is recommended to use the if-else block.

## 5.2.2 Standard library vulnerabilities

Vulnerabilities are in every aspect of programming languages. The first area which I would like to mention is the standard library. In the standard library, there are many modules, classes and functions. Among them are some methods which require caution. There is a possible injection and path changing; also, we have to be cautious when working with files and logs. There are many topics for this subsection [60, 54].

### Temporary files - *mktemp()*

Sometimes it is necessary to create temporary files when coding a program. In Python, a special module exists for this problem called *tempfile* [28]. It is a part of the standard library. This module seems secure except for one function *mktemp()*. Function *mktemp()* has been deprecated since Python 2.3, but it is still in the module. It works like this: Function creates a link with an absolute path to a non-existent file. Before your program starts working with this path and creates the actual file, somebody else can steal this path and do whatever he wants with it, more precisely with the created file. The attacker can corrupt the file, which can cause unpredictable behaviour of the application. The advice is not to use the method. For example, old versions of module NumPy used *mktemp()*. It is recommended to use NumPy 1.21.6 and newer [72].

Instead of *mktemp()* use *mkstemp()* which is more secure or you can use function *tempfile.TemporaryFile()* like on Figure 5.9 [28].

```
import tempfile

file = tempfile.TemporaryFile()
file.write(b"This is temporary file.")
file.seek(0)
print(file.read())
file.close()
```

Figure 5.9: Proper creation of temporary file.

### Pickle module

Python module pickle is used for serializing or de-serializing data from other sources [26]. It can be used for sending or receiving object states. Pickle is converting byte streams to object structure and vice-versa. The vulnerability lies in the byte string, which is not formatted and prone to outside code execution. It means that the attacker can, for example, read the

contents of the server, including configuration files or sensitive information, as shown later in the Chapter 8.

The best solution for this vulnerability is:

- do not use pickle module, use JSON
- do not pickle or unpickle data from untrusted sources
- use HMAC or other algorithms for ensuring integrity of data

### Command injection (exec, eval, input, os)

These built-in functions are vulnerable to the same problem. Command injection is a problem when the attacker can execute code through other functions or codes. The problem occurs when the programmer does not sanitize the user's input. These selected functions are the most used ones, so they will be described and given a solution [57].

Eval is a function that evaluates a given expression through Python evaluating rules. It can be used for evaluating number expressions (*eval('x+1')*). The return value is a value from the evaluated expression [24]. Exec is a built-in function for executing Python code [24]. Both of these functions face the same problem with command injection. In the worst cases, the command injection can give the attacker control over the target's system. The solution for the proper use of these methods is validating user input and not giving the input right into these methods. The third method is a little bit different.

The method input reads user input and puts it into a variable [24]. In Python 2.x was possible to create an exploit like this 5.10:

This vulnerability is fixed now, but it could still be a problem for someone using an older

```
password = "something"
input_password = input("Please enter your password: ")
if password == input_password:
    print("Logged!")
else:
    print("Incorrect password!")
```

Figure 5.10: Input exploit.

```
honza$ python command_inj.py
Please enter your password: password
Logged!

honza$ python3 command_inj.py
Please enter your password: password
Incorrect password!
```

Figure 5.11: Python 3 has patched this exploit.

Python version (not recommended). Method input from version 3.0 converts input into the

string. That is why in Python 3 is, the condition false. However, in Python 2.x is true because the method evaluated given input as a variable. It means that the condition is for Python 2.x something like `var password == var password` which is true. Figure 5.11 shows breaking the logging process by naming the password the same as the variable for the password.

Other methods which are frequently used are from the `os` module [25]. The most dangerous method is the `os.system()`, which uses only a single string argument executed as a command. That opens code for command injection vulnerabilities; therefore, it is not recommended to use this function at all. Furthermore, if necessary, check that the argument is valid and be sure that the input is what you want to execute. Recommended function to execute commands is `subprocess.function()` (Figure 5.12). However, there is also vulnerability if the programmer sets the argument `shell=True`. This setting means that the argument can execute its shell, which differs based on the system. It could cause undefined behaviour of the application or, if used right, delete the user's files, so it is recommended to use `shell=False`.

```
subprocess.Popen(['nslookup', hostname], ... , shell=False)
```

Figure 5.12: The right way of executing commands.

## Regular expressions

Even if regular expressions are an efficient tool when checking user input, they are vulnerable to ReDoS attacks [73]. This attack uses algorithmic complexity against the program. The complexity of regular expression can reach up to  $2^m$ , where  $m$  is the length of the regular expression. This means some expressions would take exponential time to evaluate. Thus programmer has to be careful when using regexes. Some regexes are forbidden due to their evaluation complexity against inputs like `aaaaaaaaaaaaaaaaaaaaaaaaaaaaa!`. A list of evil regexes can be found here [73] or later in this thesis in chapter 8.

## String formatting

String formatting in Python also has its problems. They are not problems like in C language, which will be described in Chapter 6.1, where working with strings is tricky, but bad string formatting can cause a data leak — detailed explanation on this website [51]. Through `str.format()`, an attacker can see internal parts of objects or sensitive data.

The best solution is to check user input using whitelisting or not having user input. Other solutions include using “old,” style formatting. That type of formatting is similar to C string formatting, and it is relatively secure. Its cons are smaller clarity and longer code. Other solutions are to use f-strings (Python 3.6+) or template strings from the standard library [5].

## XML

Sometimes is needed XML input to create a properly working program. Several Python modules can help with XML. The problem is that every XML Python module is vulnerable to two types of attacks. The first one is called Billion laughs attack, and the second one is

The Quadratic blowup [31].

Billion laughs attack, as in Figure 5.13, is constructed from ten entities. Every entity consists of “lol,” strings, which call other defined lol elements. With this calling, one file size of about one kB needs up to several GBs of memory when executed. The Quadratic blowup attack uses the same formula of calling entities and creating DoS using too much memory.

These attacks have been known for decades, and there is only one solution. The best practice here is using the package defusedxml. Defusedxml is created to prevent especially these two attacks [48]. Other XML Python modules do not cover these two vulnerabilities.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Figure 5.13: Example of million laughs attacks XML.

## Random

In Python, a module called random is used to generate pseudorandom numbers. This module is not very secure because its algorithm is quite predictable, and there are crackers and guessers available on the internet for free. Its primary purpose is to be used for general purposes, not encrypting and security. Please, do not use it for these purposes [60].

For secure pseudorandom number generation use module secrets instead [29]. This module can ensure a secure token generator and many other number generators.

## Assert

Assert comes very handy in testing because the programmer can assume with this statement that the assertion is true, and if it is not, Python raises an assertion error. However, assert should be used only in testing because having assertions in production code is vulnerable. Assert can be carried out only if Python’s variable `__debug__` is set to true. This variable should be set to true only in development. Even if the variable is set to true in production does not mean that every environment on every device is the same.

Furthermore, the variable should be set in production to false. This situation can raise problems where the Python interpret skips part of the code or executes part of the code

that should not be executed. It can create undefined behaviour [54, 46].

Figure 5.14 shows the wrong usage of `assert`. In this example, `assert` tests if the user has superuser privileges. In testing, everything runs smoothly, but in production Python interpreter removes assertion, and the `assert` condition will be skipped, giving all users superuser privileges. The best practice is to use `assert` only in a testing environment. When going to production, be sure that all of the assertions are removed or do not affect the functionality of the code or its security (authorisation, authentication, and others).

```
def check_permission(super_user):
    try:
        assert(super_user)
        print("\nYou are a super user\n")
    except AssertionError:
        print(f"\nNot a Super User!!!\n")
```

Figure 5.14: This assertion could break the authorization control [46].

## Tarfile and zipfile

Tarfile is a module for extracting archives [30]. Programmers should avoid using tarfile when extracting from untrusted sources [54]. The archive could be a file with an absolute path containing `..` or `/`, which can cause problems because functions in tarfile can not handle that. Use module `tarsafe` instead.

The second problem with archive files is the so-called “zip bomb,” [19]. This bomb has one goal. Crush the host’s system or program by decompressing a small file containing more data than the host’s memory. Programmers can defend by setting and checking the maximum size of decompressed data and the maximum number of files. This link [19] shows `.zip` files having a few kilobytes compressed, but they need terabytes of free space after decompression. This exploit of extracting files has been solved, but it is good to mention that programmers should be cautious even nowadays when extracting `.zip` files.

## 5.3 Web programming

This part of coding is popular these days. Most of the world depends on the internet, and web applications are the best way how to build user-friendly access to the internet. However, every sun creates a shade. The problem with web applications is that they store sensitive data about users, their passwords and others. Web applications are vulnerable to many types of attacks. This section should introduce the worst web programming vulnerabilities and their solution.

### 5.3.1 Python frameworks for web programming

Web development was long about languages like PHP and HTML, CSS and JavaScript. This rule does not apply these days when there are several good frameworks (light-weighted or full-stack), even for Python. HTML, CSS and JavaScript are still essential for web development, but web development is more accessible to the public overall. Django is the framework that I used for examples and educational tools. For other frameworks, visit this

website [47].

The best practice is to use one of these frameworks because they have already implemented security features like Django forms for registering users. These forms are probably better than user-created, so it is recommended to use them. They are also user-friendly and easy to use. If the programmer is cautious about its security, he can visit [21] to ensure that these forms should ensure the secure handling of data.

The recommended framework is the Django framework because it is one of the most popular frameworks due to its simplicity and usability. Also, I have been using Django for some time now, and its simplicity is breathtaking [20]. Developing web applications on your own could be hazardous and create many vulnerabilities, the worst described below. It is better to use any framework.

### 5.3.2 OWASP Top 10

OWASP Top 10, as it was said in Chapter 3, is the list of web applications' most severe vulnerabilities. These vulnerabilities below are a problem even for the Python frameworks mentioned above. This section is focused on the rest of the vulnerabilities that have not been described yet.

#### Broken access control

The most common exploit is called Broken access control. This exploit appeared in 318 thousand web apps that were tested. This number shows how many vulnerable applications are there, whilst it is known that this exploit can be very dangerous. The attacker can bypass access control by modifying the HTML page or the site's URL or modifying the internal application state. This attack can cause elevation of privilege, which is acting as the user when not logged or as admin when not logged as admin. Other problems that Broken access control can cause are metadata manipulation, cookie manipulation or force browsing to authenticated pages as an unauthorized user. [62].

Figure 5 is a URL with visible request parameters. There is one thing wrong and one

```
https://example.com/accounts/details?id=123&password=abcdefg
```

Figure 5.15: URL with visible request parameters.

that should be secured. The wrong thing is a non-encrypted password. The password must be encrypted in requests or the URL address. The second one is that the attacker can modify parameters to malicious values and send the request. So the first advice is that every request has to be validated. Values and authorization rights need to be correct. The prevention is to encrypt requests in the browser and validate the parameters. It is also good practice to limit the number of requests for users as it helps defend against automatic tools.

#### Cryptographic failures

The second-largest exploit is not a root cause but more of a symptom. The main problem with cryptography is that it used to be broken, and data are visible in the transmission. That is why we have to check if we transmit data in the right format and encrypted and



check what kind of encryption we use. Personal information has to be encrypted due to EU privacy laws (GDPR) [63].

The solution is simple. For private data transmission, use HTTPS instead of HTTP, which is not encrypted. For other types of encrypted communication, such as SMTP, use TLS for encryption. Check if the server is trusted, check for needed certificates, and update encryption methods for more powerful and secure methods. If using generated cryptographic keys, be sure that they are properly managed [6]. Also, if not needed, do not store sensitive data. Data that are not stored can not be stolen.

## Injection

This category was composed of several types of injection, including SQL injection, cross-site scripting and others. The most common injection is SQL injection which occurs when a programmer, for example, uses credentials directly from an authentication form to perform a SQL query. This bad practice can even lead to the deletion or leaking of the whole user database. SQL injection is exploited in Chapter 8.

Cross-site scripting (XSS) is a form of attack which executes the script on the server-side of the application. There are three kinds of XSS attacks:

- reflected XSS - from HTTP request (from other website)
- stored XSS - malicious input into the database
- DOM-based XSS - on the client-side of the application

```
<script>function(something)</script>
```

Figure 5.16: HTML script example.

Preventing this attack is simple. Do not trust user input. Validate user input and restrict the length of input or characters that users can use. When creating a SQL query, use only user input that has gone through validation [64]. In Django, use the secure Django forms for registration and other tasks. Against XSS attack applies the CSP (Content Security Policy), an HTTP request header that restricts sources from which the server can execute scripts. This header is also available for Python Django [22], so use it for the security of your application.

## Identification and authentication failures

This category contains every possible way to attack authentication. The easiest attack to defend from is a brute force attack. It combines random or not random strings from the database or some other source and tries to authenticate. This type of attack can be defended by putting the number of tries on the form and then blocking the attacker from accessing the form for some time [68].

Another vulnerability is a weak password. The form should not accept passwords that do not contain upper-case and lower-case characters, numbers and special characters like '\*' or '?'. It is also not safe to have a password similar to the username. The more complicated password, the lesser the chance of breaching of account. Fortunately, Django has this concern covered with built-in forms and password validator.

Weak passwords: 123456789, abcdefghi, fitvutbrno, password  
Strong passwords: 94K%-aN8, ]BZ-4b6hZ, crP5Fsg\$5, @d@rR5S7b

Figure 5.17: Passwords comparison.

Password has to be transmitted and stored in encrypted form. So as the session ID. Weakness is also password recovery that contains private safety questions. This recovery is also not user-friendly, so it is not usable because users tend to forget their answers. Safety questions are also vulnerable to brute force attacks. Sending notifications through e-mail is safer.

### Server side request forgery (SSRF)

Server-side request forgery is rising with new modern web applications that use user-provided URLs. Server-side request forgery happens when a web application fetches a remote resource without validating the user-supplied URL. This vulnerability can cause a redirecting to a malicious website or other places even when protected by a firewall or VPN. [71].

The prevention against SSRF is about the request and input validation. Like in most

```
https://example.com/page?url=http://127.0.0.1/admin  
https://example.com/page?url=http://127.0.0.1/phpmyadmin
```

Figure 5.18: Examples of redirecting to a admin page on the localhost.

vulnerabilities, never trust the user input. A good practice is to whitelist allowed hosts for a particular application and URLs. In Django urls.py file, there is a list of allowed URL paths and in the file settings.py is a list of allowed users.

## 5.4 Others

These guidelines could have had dozens of pages, but only the three most concerning sections were mentioned. Other vulnerabilities can be found here [16]. Some vulnerabilities have been already patched versions ago, but others have not been patched yet. All the info is available on the link mentioned before.

- executing code from comments (mentioned here 8)
- urllib.parse exploits
- ipaddress zero characters in the octets exploit

## Chapter 6

# Well-known vulnerabilities in other languages

Even if there are uncountable vulnerabilities in Python, some have already been dealt with. For example, the biggest of them all, buffer overflow, is almost impossible to do in Python. This chapter shows the reader that programmers bother with even more problems that could cause even bigger damage in other languages. The language used in the examples is C language, where these three vulnerabilities are a huge concern. Figures and content used in this chapter can be found in the book *Secure Programming with static analysis* [14].

### 6.1 Buffer overflow

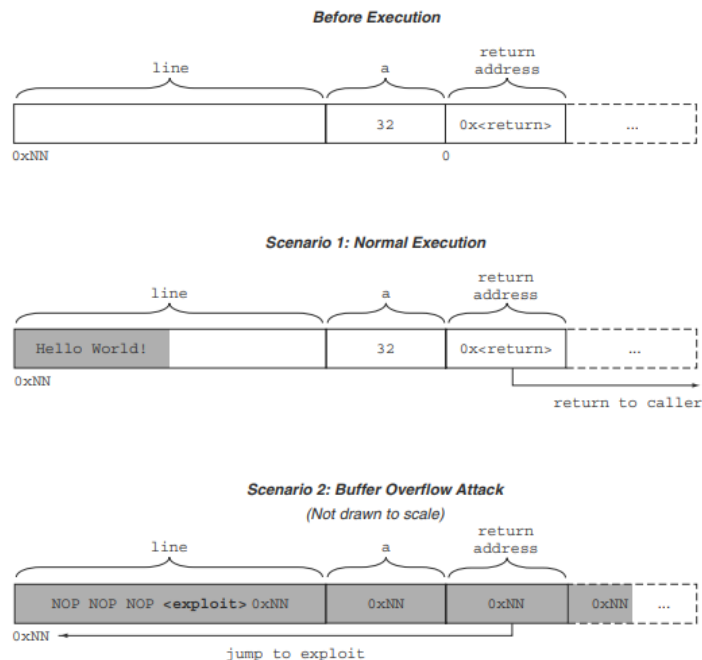


Figure 6.1: Buffer overflow illustrated [14].

This problem is the biggest problem that C language programmers have to face. Buffers are an essential part of C programming, and getting a buffer overflow is easy. C does not include string data type, so we have to use buffers when working with strings.

There are two ways to allocate memory for buffer in C. Static and dynamic. Static allocation allocates one block of memory with size depending on the data type of buffer. For example, a buffer for ten characters will have a size of ten bytes because the data type char has a size of one byte. This type of allocation is acceptable unless we are working with the buffer during the program's execution. In this case, the static allocation becomes insecure because when we concatenate a string in this buffer with another string, we can easily get buffer overflow which causes a segmentation fault error.

Segmentation error finally leads us to figure 6.1, where the buffer overflow attack is shown. The scenario before execution shows that we have allocated part of the memory of a specific size. Behind the buffer is some character, and behind, there is a return code with some address. In scenario one, where we are using the buffer as we are supposed to use, everything is fine. The problem arrives in scenario two, where we have saved a string bigger than the buffer size, for example, with some function from the string library. The memory is corrupted with data that changes the return address in memory to the address at the beginning of the buffer. There is a malicious code in the buffer itself that can be executed now. This code can do anything. It could delete some data or rewrite data or cause other problems.

Nevertheless, we can defend. When working with a buffer after its allocation, it is better to allocate the buffer dynamically. With dynamic allocation, we can ensure, unless there is no space in memory, that buffer overflow does not occur. Moreover, we can resize the buffer when the string is bigger than the buffer. The most effective way to defend is to check every input and change in the buffer. It does not cost much time to check the size of a string, and it costs more not to check the size.

## 6.2 String

In conclusion, from subsection 6.1, strings in C languages are buffers of characters. What can cause a buffer overflow or how can it be prevented was shown, but the programmer has to be warned that in the C language, there are existing functions that are not safe, and we must not use them if we want to achieve safe code. In Python, there can not occur buffer overflow, but there are ways how to improve work with strings as well.

There are functions in C that are so unsafe that even the compiler warns about them. The worst one has to be function `gets()`. This function reads input from stdin and stores it into a given buffer. Buffer overflow appears whenever the buffer is smaller than the stdin input, which often occurs.

This was the worst example. Working with string in C languages is very unsafe. Programmers should minimize the risk with righteous using of function, working with string or with input. Do not use `gets()`.

### 6.3 Integer overflow

Another thing that does not appear in Python is an integer overflow. In Python, only floats have a hard limit on value, so it is good to know what could happen if Python was like C or why Python developers chose to have a non-limit on value in integer data type.

In Figure 6.2, you can see an overflow of a 4-bit integer. In standard math, it would be

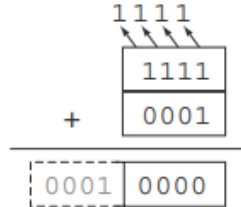


Figure 6.2: Integer overflow on 4-bit value [14].

$15 + 1 = 16$ , but here it is 0. This result shows that there are problems with overflowing. Another issue is with unsigned and signed integers. An unsigned integer can store values starting with zero; a signed integer starts with a minimal integer value if necessary.

It is good to be aware of this problem because this can also cause severe damage. An

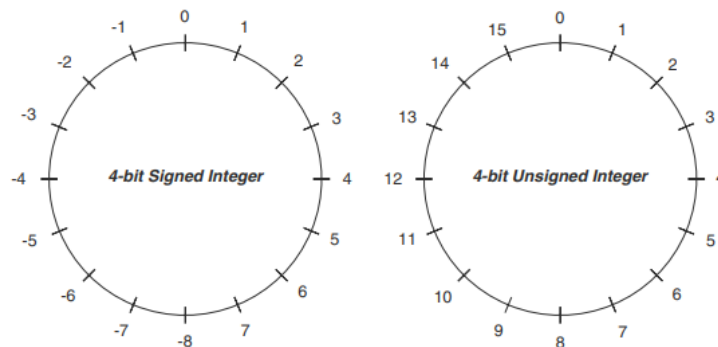


Figure 6.3: Values in signed and unsigned integers [14].

integer overflow has been used for exploiting buffer overflows. The best practice in this area would be using an unsigned integer on indexing and size allocation because these values can not be lesser than zero. Be aware of what C operations can cause an integer overflow, and check the input numbers. Even if we are using a program for counting numbers lesser than 100, it does not mean that some attacker or user would not try to write millions instead of tens.

## Chapter 7

# SeCoPy (Secure Coding in Python)

Every theoretical approach is good, but the practical approach makes learning better. SeCoPy takes this idea and makes it a reality. This chapter is focused on the development process and designing process of SeCoPy. SeCoPy is a web application that is created for educational purposes. It is developed in Python Django, which is now probably the most popular Python web development framework. It is also secure enough to create secure web applications. Front-end was created with Bootstrap's help, a web framework with built-in CSS classes and Javascript methods. This help should have ensured that SeCoPy would be good-looking, which also helps the usability of this product. The content of the SeCoPy is taken from Chapter 3, Chapter 5 and Chapter 8 of this thesis.

### 7.1 Design

The first section of this chapter is focused on design. Two different methods were used in the development, and the more usable method won. The most important rule was to make it simple because SeCoPy is not something that is meant to be complicated.

#### 7.1.1 Requirements analysis

There is a need for an application that would be usable, secure and educational. First, there was a need for use cases, which users should demand from the app for the best learning experience. These use cases in Figure 7.1 were evaluated as the most important.

From now on, it was crystal clear how the SeCoPy should look. The most usable way of implementation would be to divide the app into three independent parts. Each part should provide one of the use cases. The user could learn about secure coding in some parts, including all the theoretical matter plus the examples and existing methods.

Taking the test should be separated from the rest of the theory because the user should not look into the guidelines when taking the test. Downloading is better on a separate page because the user could only come to the app for downloads. From these thoughts, it is clear that every part of the app should be separate in some ways.

These items were the main concerns when implementing the SeCoPy. The main emphasis was on the test part, which has been developed only for this part of the work, and the user should have seen the results of his learning progress here. After submitting, the user would see the number of his correct answers and can retake the test.

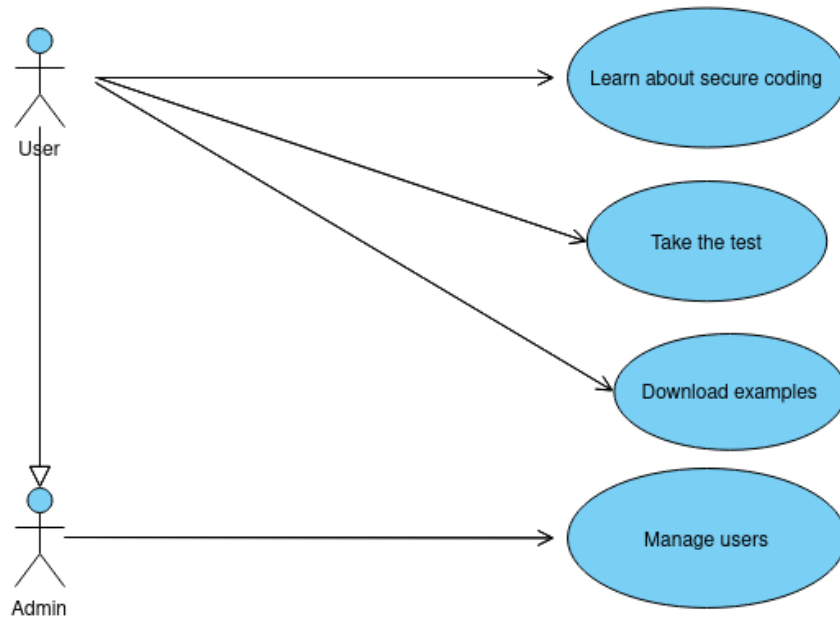


Figure 7.1: Use cases of SeCoPy.

Table of requirements	
Functional	Non-functional
Learning from theory	Application should be secure
Taking the test	Application should be usable
Re-taking the test	Implemented in Python
Download the examples	Mobile version
See the results of the test	

Table 7.1: Table of requirements.

There is also a need for a responsive mobile version of the app because millions of mobile users exist, and SeCoPy would be even more usable. After these thoughts and requirements on the SeCoPy, there is time for creating the app's first version.

### 7.1.2 Beta version

The first design idea was to design one web page with one navigation menu and all the content. In Figure 7.2, there is a navigation menu from SeCoPy beta. The main page consisted of different parts, which were shown according to the button in the menu that was clicked.

The next idea was to allow only registered users to get the test results. It should draw some line between unregistered and registered users. It was necessary to create a login and register page, which are two separate pages. It is required to insert a username, e-mail, password and several other details to register.

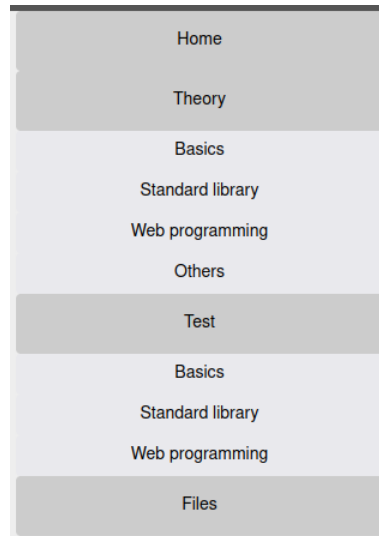


Figure 7.2: Side navigation menu (Beta version).

### 7.1.3 Final version

After consultation with my supervisor, changes have been made, which should make the application more user-friendly and usable. The most noticeable one would be marking the correct answers with green and giving user feedback on why is the one answer correct. This marking should help the user learn because he would have another source to learn from, and it is the most powerful one, which is from his mistakes.

The second change which radically improves usability is getting rid of logging in if the user wants to see his result. This learning tool should be free and completely anonymous because the user does not want to store his sensitive data.

Another change made is the floating navigation menu, which had been fixed so far in the top left corner. The SeCoPy was also made responsive for mobile devices. This change brings the sticky hamburger menu, which should be the most usable technique. Moreover, the last significant change is the change of the page logic. So far was, all content on one page. The final version has every part of the content on its own page. This change should add to the product's usability and create a more user-friendly approach.

## 7.2 Implementation of SeCoPy

This section will break down the implementation of the application. There was pressure on the secure development initially, so the first main thing was setting the Python and Django version. SeCoPy is developed in Python 3.8.10, which is not the newest version available, but it should be enough from a security point of view. Django is on version 4.0.2, which should be the latest one. After setting up the Django project, there was created virtual environment, so the requirements for this project could have been set, and the project has been divided from other local Python projects.

The SeCoPy project hierarchy looks like this: The most relevant folders and files are secopy, secopy\_app, manage.py and requirements.txt. The folder secopy is the main project folder and consists of config files for the project. The folder secopy\_app contains all of the vital files for the app itself, like views.py and static files (HTML files, Bootstrap and JavaScript



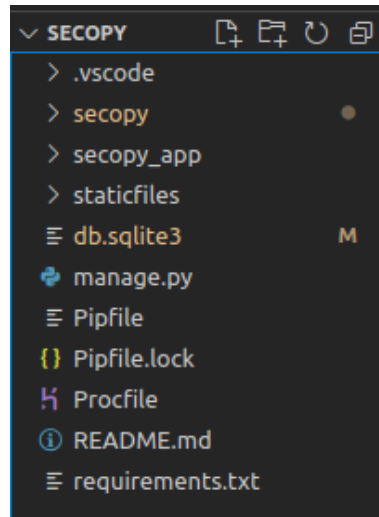


Figure 7.3: Project hierarchy of the SeCoPy.

files). File `manage.py` is very important for project management because it can set admin user or even runs the local server. It can also migrate changes in the database. The last file, `requirements.txt`, is nothing new for developers. It consists of every package needed for that project.

There are several files worth mentioning. The file handling requests are called `views.py`. This project does not require much request handling, but it is worth mentioning the handling of logging in. This method is simple yet secure because Django has built-in methods for form

```
def auth_login(request):
    if request.method == "POST":
        form = AuthenticationForm(data=request.POST)
        if form.is_valid():
            user = form.get_user()
            login(request, user)
            return redirect("../")
    else:
        form = AuthenticationForm()
    return render(request, 'login.html', {'form': form})
```

Figure 7.4: Authorization method of SeCoPy.

checking, which can be trusted. They even have some password requirements, such as not having only numerical characters or passwords the same or similar to a username.

The second important file is `urls.py`, where all of the available URLs of the project are stored. This file is not only in the `secopy_app` folder but also in the `secopy` folder with config settings. The config folder includes the admin URL, debug URL, and all of the project's URLs. To every URL pattern, there is assigned one method from `views.py`, which handles the requests on these URLs after, for example, clicking on a button.

The last thing probably worth mentioning from the implementation point of view is handling test results. There was a minor issue when implementing handling where empty question

answers caused unwanted behaviour, so the fifth option, “I do not know”, was added. The handling itself checks if the correct value in the radio is checked and then increments the variable points. The number of points is returned after the user clicks on submit, and it prints into the template. The project was developed on a local Django server 127.0.0.1:8000.

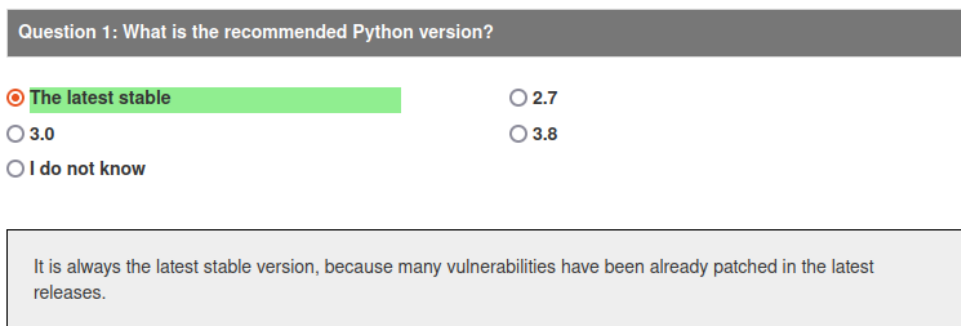
```
//question x
if(document.querySelector('input[name="venv"]:checked').value == "b"){
    points++;
}
```

Figure 7.5: Selecting the right answer in JavaScript.

After finishing basic development, there was a need for some hosting. Heroku hosting was chosen because its free version is enough for this thesis’s purposes. The SeCoPy app can be found here <sup>1</sup>.

### 7.3 Content of the application

One part of developing an actual learning tool is implementation, but there is even one more important part. Its actual content. This application was created to teach not beginners but programmers who want to code more securely. The content itself had to be more specific and professional than just some basic programming guide. It is more complex and technical. The content itself can be divided into three parts. The first part would be the guidelines themselves. In the SeCoPy are the complete guidelines, which in this thesis are chapter 5. It also contains standards and methods such as OWASP, etc. The user should have the full theory available to him without searching the internet. The second part consists of a test. The test is made up of every part of guidelines and its examples. There are five questions on every topic, which have five answers. Why five is mentioned in the above section. The four “real”, answers have been made like that because it was necessary to have answers hard enough to distinguish someone who knows the topic or read the guidelines from someone who clicks random answers and hopes for the best. The figure 7.6 shows one of the questions. After submitting the test, the correct answers and the number of points



Question 1: What is the recommended Python version?

The latest stable  2.7

3.0  3.8

I do not know

It is always the latest stable version, because many vulnerabilities have been already patched in the latest releases.

Figure 7.6: Question no. 1.

gained are shown. Also, the small explanation is shown for the user because some of the

<sup>1</sup><https://secopy.herokuapp.com/>

answers do not have to be clear for some users.

The last part of the SeCoPy is the data bank of files, where all of the exploits, examples, and code used in the development of this thesis and tool except the code of SeCoPy itself are stored.

## 7.4 Security

The secure point of view is a big part of this application. As was mentioned before, OWASP has even created The top 10 web application vulnerabilities, and the list is updated every few years. So is SeCoPy secure? To prove it is, let us compare to OWASP's top 10. Not every vulnerability can be exploited in SeCoPy, because it simply does not have as much content, so there will be a focus only on ones that can be created.

**Broken Access Control** – SeCoPy does not have too many requests. The question is, can the user enter admin restricted parts of the app when not admin?

`http://127.0.0.1:8000/admin/auth/user/` is addressing the user section in the admin part. Suppose the user does not have the right to go there. The application rightly redirects to the admin login page. So it should be secure due to the Django authorisation module.

**Injection** – This vulnerability is also impossible to create due to Django. Django forms are used when creating a new account or logging in. These forms should ensure that SQL injection or XSS will not happen.

Overall the application should be secure enough for my requirements, but there is little room for error in this small-scale application.

## 7.5 Usability

One of the topics for SeCoPy was user usability. This application supposes to be usable as much as possible, but not at the expense of security. One of the exciting things is that text aligning matters [3]. This application uses text-align left, which should be the best for this type of work. Other steps to user usability were taken after designing the final version of SeCoPy.

For example, seeing results as unregistered could have huge consequences for returning users. Also, marking correct answers could raise the percentage of users getting better results in the test and even learning better, but it is only an assumption of a developer. The reality can not be found out until testing.

## 7.6 Testing

First user testing was made by my supervisor, which gave me ideas on final design. The second one, the bigger one, was made on sample of four BUT FIT students. The important metrics for testing were:

- the number of points received before reading the guidelines
- reading time of the guidelines
- the number of points received after reading the guidelines
- improvement rate (in %)

- is the home page understandable?

The improvement rate and the number of points were necessary because of the learning efficiency of SeCoPy. Reading time is essential because too short reading time probably means that guidelines are not complete or too basic. On the other hand, a long reading time could scare new readers. It is important to have content divided into reasonable long parts.

### 7.6.1 Results of testing

Users reported small design issues like footer overlapping navigation menu and navigation menu bugs or non-functional link. Overall I have got important feedback. Other problem that users reported was incomprehensible text in the guidelines in some parts or unclear question.

From the results, we can assume that the guidelines and the test are doing what they

Testing results	
Metric	Result (average)
Number of points before reading	11.25
Reading time	about 1 hour
Number of points after reading	15.25
Improvement rate	35.6 %
Understandable home page	mostly yes

Table 7.2: Results of the testing.

should be. The test is not easy even for somebody with programming experience but gets better results after reading the guidelines. After this feedback, some bugs were fixed in the application, and it is even more usable than before. The reading time is only an approximation because every person has their reading methods. Some have skipped some parts, and others read some parts two times. The overall result is pleasing for me because there is a sign that these guidelines could be useful.

# Chapter 8

## Real-life exploits

This chapter is about exploits again but from a more detailed view. Some examples have been already shown in Chapter 5, which focuses on the theoretical approach with some examples. Here, in this chapter, examples will be described and explained. Precisely what is happening in the background and why it is happening.

### 8.1 SQL Injection

The first exploit I would like to cover is SQL injection. This exploit is widespread and was introduced theoretically in the previous chapters. I created another web application for showing SQL injection to the user what it could do.

The application itself was again developed in Python Django with some Bootstrap. This application simulates a database of users with their details. Here is the database 8.1. It

#### Table of users:

Username  Password

Username	Name	Email	Country	Password	Age
admin	Jan	admin@admin.com	CZ	sq351ay.4	35
joedoe	Joe	joedoe@email.com	USA	joe123doe	25
johnny	John	john.smith@email.com	UK	q456ws..54za]	30
bozka_123	Božena	bozkanova1@seznam.cz	CZ	bobik123	65
fit_destroyer	Martin	marta45@email.sk	SK	fitisthebest420	21

Figure 8.1: Database of users.

is fully functional; the user can add users as he pleases, and it will be shown in the table. Adding is done with the help of Django form, so no vulnerabilities should be there. What is important is that the “login,” on top of the table uses the method `raw()` on executing SQL queries. When the user wants to search for the added user (login), it must enter the username and password, which is then not tested and executes SQL query. The method for searching users looks like this: The function is a little bit edited because of the

```

def results(request):
    searched_username = request.GET.get('search_text')
    searched_password = request.GET.get('search_pass')
    if searched_username == "" and searched_password == "":
        query = MyUser.objects.raw("SELECT * FROM sql_exploit_myuser")
    else:
        query_string = "SELECT * FROM sql_exploit_myuser
        WHERE username = '%s'
        AND password = '%s'" % (searched_username, searched_password)

        query = MyUser.objects.raw(query_string)
    return render(request, 'index.html', {'data': query})

```

Figure 8.2: SQL handling function.

LaTeX formatting. As was mentioned in the guidelines, input has to be validated, and here it is not. When the user/attacker finds out, he can try to exploit the method. In this case, it is straightforward. Typing one of these inputs to the username field breaks the app, and the user can become any user he wants or choose to print all users.

- admin' –
- ' or 1=1–
- [any username]' –

The app works as follows. It simulates the login form, and the user that is shown is the one user would log in. The SQL query which executes after clicking on the search/reset button is shown below the user table. When the user enters the first item from the list, the SQL query would be 8.3:

As you can see, the user is now logged as admin because typing ' after admin means the

```

SELECT * FROM user WHERE username = 'admin' --' AND password = ''

```

Figure 8.3: SQL injection, user is the admin now.

end of the string (username) and - - means the start of the comment. Everything after these characters does not execute, so the user does not have to enter the password.

### 8.1.1 Solution

It must be said that cause SQL injection is not as easy as it seems in modern Django, but it is possible. It showcases terrible programming techniques that someone could have used somewhere. Even if the Django is relatively safe, there is some need for prevention.

How to prevent it? It has been mentioned a few times in these guidelines, but in this case, simply validating the input with the help of regular expressions, for example, would stop the SQL injection. Never execute input right from the user. Do not allow characters like '

or - in the username. For the best result, use Django forms. Its login form has already a built-in password and username validator. Moreover, its login method can create a session. Please do not reinvent the wheel when it is not necessary.

Also, the string formatting is very crucial here. If the right string formatting was used, the exploit would not be possible. Figure 8.4 shows sting formatting which not allows SQL injection. This formatting creates a server error, but nothing more. With this string formatting, input validation or Django can prevent SQL injection.

```
query_string = "SELECT * FROM sql_exploit_myuser WHERE username={username}
AND password = {password} ".format(username=searched_username,
password=searched_password)
```

Figure 8.4: Better string format.

## 8.2 ReDoS

Regular expression denial of services is an interesting attack. I have been using regular expressions for several years, and I would not believe it if somebody told me that regexes can user crash the whole application or slow it down drastically. As was said before, there are regular expressions that can cause harm. These expressions are called “evil regexes”, and some could be in many applications because they were used to check usernames and e-mails. In this chapter, it is appropriate to list them.

- (a+)+
- ([a-zA-Z]+)\*
- (a|aa)+
- (a|a?)+
- (. \*a)x for x  $\setminus$ >10
- $\hat{[a-zA-Z0-9]}([.][\_]+)?([a-zA-Z0-9]+)^*$   
 $(@)1[a-z0-9]+[.]1([a-z]2,3)([a-z]2,3[.]1[a-z]2,3)$$

The last regex was used for e-mail validation [73]. The one thing these regexes have in common is their vulnerability. This input *aaaaaaaaaaaaaaaaaaaaaaaaaaaaa!* causes timeouts and crashes. For example, I added input validation in my web application which simulates the login form and user database. Of course, I am using a regular expression because, as I have already written, regular expressions are an excellent helper. The condition is simple yet “effective”. When the user types the input *aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!* the request is taking forever. When the user wants to create a new request, the previous request must be completed. It means that the server has to be restarted for desired functionality. For example request where username was *6a518sd98a14s98fd198as1d98as1d9!* lasted for 296.7 seconds. That is quite a lot. From this point of view, it is quite a powerful attack. Regular expressions are a favourite technique for input validation, and they are very effective when performing a whitelisting validation. So what is the prevention of ReDoS?

```

#testing redos - yes it works:
#aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa! should be enough
if re.match(r'^(([a-zA-Z0-9])+)+$', str(searched_username)):
    query_string = "SELECT * FROM sql_exploit_myuser WHERE username = '%s'
    AND password = '%s'" % (searched_username, searched_password)

    query = MyUser.objects.raw(query_string)
else:
    return render(request, 'index.html', {'data': query})

```

Figure 8.5: Input validation with evil regex.

### 8.2.1 Prevention

The only defence is to avoid using evil regular expressions. Use of regular expressions is still recommended, but the advice is to be careful and thoroughly test the condition if there is no evil regular expression. It could be tested simply, as shown and explained in the examples above. On this site [73], the programmer can find further information about other exploits.

## 8.3 Comment code execution - 'Trojan source'

This exploit, which has been called in related work Trojan source [10], is one of the latest examples that there are still vulnerabilities for many programming languages out there. It was discovered in late 2021, and its impact was quite drastic.

Its problem is not in exploiting from remote sources but from the inside. It is not a big concern for programs where the insides of the application are unknown to users. To exploit this vulnerability, the attacker has to access the source code. However, it is a different story for open-source applications and online sites such as GitHub, where are these open-source projects usually stored or even public developed.

Because this vulnerability is relatively new, it was intentionally put in this chapter for a more detailed introduction. For example, I prepared simple registration and login script in Python. We could pretend that this form or background is used somewhere, even if it is a really simple example. It is an example of an early return due to a return in a comment. In Figure 8.6, we can see that return is misplaced outside the multi-line comment. This should mean one thing. If the condition is evaluated true method returns, and the code continues. What is the big deal when it can be seen at first sight? The big deal is that is the way how the interpreter sees the code. The basic text editor or the GitHub repository with invisible symbols turned off sees it like can be seen in Figure 8.7. All the problems create the Unicode character U+2067 (Right-to-Left-Isolation). This character is used when there is a need for the right to left reading. This feature is used, for example, in Arabic or Hebrew language. Why might it be a problem, and what it can cause?

The biggest problem is that this vulnerability can be exploited in Python and also in languages like C, Java, C++, and many others. It is an issue that occurs when a programmer downloads code from the internet. There are many open-source codes out there or pages with some code examples. Because this character could be invisible in code editors, the programmer does not have to see the problem and takes malicious code to production. On the other hand, the programmer can add this line of code into some open-source project



```

'''user wants to login to system'''
def login(self):
    print("Attempt to login: ")
    check_user = input("Type your username: ")
    check_pass = input("Type your password: ")

    print(self.users)

    for user in Form.users:
        if check_user == user["username"] and
        check_pass == user["password"]:
            '''If it is user in database then login and RLI'''return
            print("Login successful")
            '''Login method or something'''
            return
        else:
            pass

    print("Unknown credetials!")

```

Figure 8.6: The return is weirdly placed.

```

'''If it is user in database then login and return;'''

```

Figure 8.7: How the text editor sees it.

where thousands of lines of code are written, and nobody even notices it, and when they notice, it could be taken only as a simple comment.

In Figure 8.6, early returns cause DoS by not enabling the user to log in. This early return comment can be put anywhere in the code, so this Figure is only one of the many examples. This Figure 8.8 shows that even if the user tries to log in to an already created account, the script does not print the message 'Login successful!' as it should. Even if you tried to log in as a non-existing user, the program does not output anything. Everything is caused by the comment.

### 8.3.1 Solution

The prevention is quite simple. Use environments where are invisible characters shown, such as Visual studio code, where the default setting is set to show invisible characters. GitHub has already managed to add a warning to all projects where invisible characters are included. The third piece of advice directly for Python is to use only one-line comments (starts with #). These comments are not vulnerable to Trojan source.

```

Attempt to login:
Type your username: H0nza
Type your password: 123

Users:
[{'username': 'Admin', 'password': 'admin'},
{'username': 'H0nza', 'password': '123'}]
Program ended.

```

Figure 8.8: Denial of services because of blocked login.

## 8.4 Pickle module

As mentioned in Chapter 5, pickle is a Python module used for deserializing and serializing objects from or to a byte stream [26]. The problem with pickle is that it does not check the data sent through the byte stream. The object is usually encrypted by base64 encryption, so it is hard to check for malicious content.

This exploit will be shown how does the pickle works. Then using pickle, we create an attack where the attacker steals data from the server. This application is created using Flask, another Python framework for web applications. For this example is more suitable Flask because it does not require such preparations as Django. Pickle has many methods, but for this exploit are important these two and also override of method `__reduce__()`:

- `dumps()` - this method returns a byte string of a given object
- `loads()` - this method deserializes byte data to object form
- `__reduce__()` - this method creates the initial object, which is then modified

These methods make it possible to create an object where the command can be hidden. The problem is that communication through request is encoded by base64 on the server. So using module base64, we can encrypt the byte string. The goal of this attack is to steal user data from the server. Figure 8.9 shows how the executed command, its byte string and then pickled string look like in the program. The pickled string should be sent directly

```

command = ('ls -a')
return os.system, (command,)

\x80\x04\x95 \x00\x00\x00\x00\x00\x00\x00\x00\x8c\x05posix\x94\x8c\x06system\x
\x94\x93\x94\x8c\x05ls -a\x94\x85\x94R\x94.'

b'gASVIAAAAAAAAAACMBXBvc2l41IwGc3lzdGVt1JOUjAVscyAtYZSF1FKULg=='

```

Figure 8.9: Pickling and encrypting command.

to the server, which executes the code and shows its content. It could be done by sending it over a cookie, as shown here [1].

This exploit is created with the help of a blog post by David Hamann [34]. The server looks

in the request form and unpickles everything sent in item 'pickle='. When the attacker runs this command 8.10, the server responds with a list of files and folders 8.11.

Now it is easy to get the data from the server. If the root rights are needed, there is no problem executing the command line interface and gaining those rights. This example only shows how to get data. If it is rerun with this command `'ls -a && cd data && ls -a && cat *.txt'`, the attacker tries to open files in folder data with a .txt ending. Fortunately for him, folder data contains file users.txt with all of the users logged. The attacker has all the usernames now and can continue with other types of attacks. SQL injection on the login form, for example.

```
curl -d "pickled=gASVIAAAAAAAAAACMBXBvc2l41lwGc3lzdGVt1J
OUjAVscyAtYZSF1FKULg==" http://127.0.0.1:5000/pickle
```

Figure 8.10: Sending server the command.

```
. . . app.py data env __pycache__ venv
127.0.0.1 - - [31/Mar/2022 10:56:37] "POST /pickle HTTP/1.1" 200
```

Figure 8.11: Response from the server.

### 8.4.1 Solution

Due to encryption and difficult data validation in the pickle module is very hard to distinguish between normal and malicious data. The only suitable solution or prevention for this problem is not to use pickle at all. If there is no other possibility than using a pickle, try to use some encrypting algorithm such as HMAC for assuring data integrity. Standard base64 encoding is publicly known, and as shown in the example, it can be misused quite easily.

## Chapter 9

# Conclusion

This thesis explained the need for secure coding and the existing solutions and materials. The most important is implementing the new educational tool SeCoPy and new guidelines for coding in Python, which should ensure that programmers will have learning materials. This thesis points out that it is important to code securely, and programmers should be cautious. There are too many vulnerabilities even in Python, which, as the thesis suggests, does not have as significant security concerns as the C language. There are no official Python guidelines, and Python developers should consider writing them. These guidelines are only the fundamental contribution to what could be massive official guidelines just as good as Java has. Existing tools are insufficient for general public usage because they are too focused on one topic, mainly web development. Also, they are behind a paywall, and I think that even if it costs many hours to develop this kind of application and even these guidelines, it should be free as it prevents costly mistakes.

Programmers should use this tool to code more securely or students if they want to know something about secure coding. This thesis was written when Python 3.10.0 was the latest stable release. It is important to keep up even these guidelines up to date. Everyone reading this thesis should take the final thought: The programming will never be completely secure.

# Bibliography

- [1] OXBRO. *Pickle Insecure Deserialization* [online]. 2021 [cit. 2022-03-30]. Available at: <https://maoutis.github.io/writeups/Web%20Hacking/Pickle%20Insecure%20Deserialization/>.
- [2] 101 regular expression. *Regex101: Regex Library* [online]. 2022 [cit. 2021-02-15]. Available at: <https://regex101.com/library>.
- [3] ARCHITECT, T. W. *Does Text Alignment Matter for Accessibility and Usability?* [online]. 2020 [cit. 2022-03-22]. Available at: <https://thewebsitearchitect.com/does-center-aligned-text-matter-for-accessibility/>.
- [4] AVATAO. *Avatao* [online]. 2021 [cit. 2021-01-17]. Available at: <https://avatao.com/>.
- [5] BADER, D. *Python String Formatting Best Practices* [online]. 2016 [cit. 2022-02-22]. Available at: <https://realpython.com/python-string-formatting/>.
- [6] BARKER, E. *Recommendation for Key Management* [online]. 2020 [cit. 2022-04-28]. Available at: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf>.
- [7] BEGINNERS, P. for. *When to Use try/catch Instead of if/else* [online]. 2022 [cit. 2022-02-15]. Available at: <https://www.pythonforbeginners.com/control-flow-2/when-to-use-try-catch-instead-of-if-else>.
- [8] BELLAIRS, R. *What Is Static Analysis? Static Code Analysis Overview* [online]. 2020 [cit. 2022-04-28]. Available at: <https://www.perforce.com/blog/sca/what-static-analysis>.
- [9] BHATTACHARYYA, S. *Cybersecurity Breaches Of 2021 Worth Taking A Look* [online]. analyticsindiamag.com, november 2021 [cit. 2021-12-21]. Available at: <https://analyticsindiamag.com/cybersecurity-breaches-of-2021-worth-taking-a-look/>.
- [10] BOUCHER, N. and ANDERSON, R. *Trojan Source: Invisible Vulnerabilities* [online]. 2021 [cit. 2022-03-23]. Available at: <https://trojansource.codes/trojan-source.pdf>.
- [11] BROOK, C. *What's the Cost of a Data Breach in 2019?* [online]. 2020 [cit. 2022-03-23]. Available at: <https://digitalguardian.com/blog/whats-cost-data-breach-2019>.
- [12] CARATTINO, A. *Complete Guide to Imports in Python: Absolute, Relative, and More* [online]. 2022 [cit. 2022-02-10]. Available at: <https://www.pythonforthelab.com/blog/complete-guide-to-imports-in-python-absolute-relative-and-more/>.

- [13] CHECKMARX. *Codebashing* [online]. 2021 [cit. 2021-12-21]. Available at: <https://free.codebashing.com/>.
- [14] CHESS, B. and WEST, J. *Secure programming with static analysis*. 1st ed. Addison-Wesley Professional, 2007 [cit. 2021-12-21]. ISBN 978-0-321-42477-8.
- [15] CORPORATION, T. M. *CVE Program Mission* [online]. 2022 [cit. 2022-03-24]. Available at: <https://www.cve.org/>.
- [16] CVE. *Python : Security Vulnerabilities* [online]. 2022 [cit. 2022-03-22]. Available at: [https://www.cvedetails.com/vulnerability-list/vendor\\_id-10210/product\\_id-18230/Python-Python.html](https://www.cvedetails.com/vulnerability-list/vendor_id-10210/product_id-18230/Python-Python.html).
- [17] CYBERTALK.ORG. *Alarming cyber security facts to know for 2021 and beyond* [online]. 2021 [cit. 2022-01-15]. Available at: <https://www.cybertalk.org/2021/12/02/alarming-cyber-security-facts-to-know-for-2021-and-beyond/>.
- [18] DELOITTE. *Impact of COVID-19 on Cybersecurity* [online]. Deloitte, may 2021 [cit. 2021-12-21]. Available at: <https://www2.deloitte.com/ch/en/pages/risk/articles/impact-covid-cybersecurity.html>.
- [19] FIFIELD, D. *A better zip bomb* [online]. 2019 [cit. 2022-02-22]. Available at: <https://www.bamssoftware.com/hacks/zipbomb/>.
- [20] FOUNDATION, D. S. *Meet Django* [online]. 2022 [cit. 2022-04-13]. Available at: <https://www.djangoproject.com/>.
- [21] FOUNDATION, D. S. *Working with forms* [online]. 2022 [cit. 2022-04-28]. Available at: <https://docs.djangoproject.com/en/4.0/topics/forms/>.
- [22] FOUNDATION, M. *Installing django-csp* [online]. 2016 [cit. 2022-04-13]. Available at: <https://django-csp.readthedocs.io/en/latest/installation.html>.
- [23] FOUNDATION, O. *OWASP* [online]. 2021 [cit. 2021-12-21]. Available at: <https://owasp.org/>.
- [24] FOUNDATION, P. S. *Built-in Functions* [online]. 2022 [cit. 2022-02-16]. Available at: <https://docs.python.org/3/library/functions.html>.
- [25] FOUNDATION, P. S. *Os — Miscellaneous operating system interfaces* [online]. 2022 [cit. 2022-02-16]. Available at: <https://docs.python.org/3/library/os.html>.
- [26] FOUNDATION, P. S. *Pickle — Python object serialization* [online]. 2022 [cit. 2022-02-16]. Available at: <https://docs.python.org/3/library/pickle.html>.
- [27] FOUNDATION, P. S. *Re — Regular expression operations* [online]. 2022 [cit. 2022-02-15]. Available at: <https://docs.python.org/3/library/re.html>.
- [28] FOUNDATION, P. S. *Re — Regular expression operations* [online]. 2022 [cit. 2022-02-16]. Available at: <https://docs.python.org/3/library/tempfile.html>.
- [29] FOUNDATION, P. S. *Secrets — Generate secure random numbers for managing secrets* [online]. 2022 [cit. 2022-04-12]. Available at: <https://docs.python.org/3/library/secrets.html>.

- [30] FOUNDATION, P. S. *Tarfile — Read and write tar archive files* [online]. 2022 [cit. 2022-02-16]. Available at: <https://docs.python.org/3/library/tarfile.html>.
- [31] FOUNDATION, T. P. S. *XML Processing Modules* [online]. 2022 [cit. 2022-02-22]. Available at: <https://docs.python.org/3/library/xml.html#xml-vulnerabilities>.
- [32] GALANSKÁ, K. *Usability of Usable Security Guidelines from IT Professional Point of View*. Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: <https://www.fit.vut.cz/study/thesis-file/24001/24001.pdf>.
- [33] GEEKS, G. for. *Try-except vs If in Python* [online]. 2021 [cit. 2022-02-15]. Available at: <https://www.geeksforgeeks.org/try-except-vs-if-in-python/>.
- [34] HAMANN, D. *Exploiting Python pickles* [online]. 2020 [cit. 2022-03-30]. Available at: <https://davidhamann.de/2020/04/05/exploiting-python-pickle/>.
- [35] INC., N. S. A. S. *Secure Coding* [online]. NTT Security AppSec Solutions Inc., march 2022 [cit. 2022-03-20]. Available at: <https://www.whitehatsec.com/glossary/content/secure-coding>.
- [36] ISO. *ISO/IEC 27001* [online]. ISO, march 2022 [cit. 2022-03-24]. Available at: <https://www.iso.org/isoiec-27001-information-security.html>.
- [37] LIMITED, S. C. W. *Secure Code Warrior* [online]. 2021 [cit. 2021-01-17]. Available at: <https://www.securecodewarrior.com/>.
- [38] LUMINOUSMEN. *Python Static Analysis Tools* [online]. 2021 [cit. 2022-04-12]. Available at: <https://luminousmen.com/post/python-static-analysis-tools>.
- [39] MAGALHAES, M. *Security vs. usability: Does there have to be a compromise?* [blogpost (english)]. Dec 2018. <https://techgenix.com/security-vs-usability/>. Available at: <https://techgenix.com/security-vs-usability/>.
- [40] MDN. *Regular expressions* [online]. 2022 [cit. 2022-04-12]. Available at: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\\_Expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions).
- [41] MELTZER, R. *How and Why Companies Use Python* [online]. 2022 [cit. 2022-04-14]. Available at: <https://www.lighthouselabs.ca/en/blog/how-and-why-companies-use-python>.
- [42] NIST. *About NIST* [online]. 2022 [cit. 2022-04-14]. Available at: <https://www.nist.gov/about-nist>.
- [43] NIST. *CVE-2014-1912 Detail* [online]. 2022 [cit. 2022-02-09]. Available at: <https://nvd.nist.gov/vuln/detail/CVE-2014-1912>.
- [44] ORACLE. *Secure Coding Guidelines for Java SE* [online]. Oracle, september 2020 [cit. 2021-12-21]. Available at: <https://www.oracle.com/java/technologies/javase/seccodeguide.html>.
- [45] OWASP. *C5: Validate All Inputs* [online]. 2022 [cit. 2022-02-15]. Available at: <https://owasp-top-10-proactive-controls-2018.readthedocs.io/en/latest/c5-validate-all-inputs.html>.

- [46] PANDEY, M. *How “assertions” can get you Hacked !!* [online]. 2021 [cit. 2022-04-12]. Available at: <https://infosecwriteups.com/how-assertions-can-get-you-hacked-da22c84fb8f6>.
- [47] PATEL, J. *7 Top Python Frameworks You Should Consider For Your Web Project* [online]. 2021 [cit. 2022-04-13]. Available at: <https://www.monocubed.com/blog/top-python-frameworks/>.
- [48] PYPI. *Defusedxml 0.7.1* [online]. 2021 [cit. 2022-02-22]. Available at: <https://pypi.org/project/defusedxml/>.
- [49] PYVEC. *Python.cz* [online]. 2022 [cit. 2022-01-15]. Available at: <https://python.cz/>.
- [50] RESEARCH, T. *Global Online On-Demand Home Services Market 2017-2021 to grow at a CAGR of 49%: Technavio* [online]. 2017 [cit. 2022-02-23]. Available at: <https://www.businesswire.com/news/home/20170503006089/en/Global-Online-On-Demand-Home-Services-Market-2017-2021-to-grow-at-a-CAGR-of-49-Technavio>.
- [51] RONACHER, A. *Be Careful with Python’s New-Style String Format* [online]. 2016 [cit. 2022-02-22]. Available at: <https://lucumr.pocoo.org/2016/12/29/careful-with-str-format/>.
- [52] ROSS, R., MCEVILLEY, M. and OREN, J. C. *Systems Security Engineering* [online]. 2017 [cit. 2022-01-15]. Available at: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-160v1.pdf>.
- [53] SCHLOTHAUER, S. *Which programming language is the most secure? High security vulnerabilities for Java have declined since 2015* [online]. JAXenter.com, march 2019 [cit. 2021-12-21]. Available at: <https://jaxenter.com/security-vulnerabilities-languages-157038.html>.
- [54] SCOTT, A. *Security Pitfalls in the Python Standard Library* [online]. 2021 [cit. 2022-02-16]. Available at: <https://medium.com/ochrona/security-pitfalls-in-the-python-standard-library-ee4692723946>.
- [55] SECURITY, K. *‘Trojan Source’ Bug Threatens the Security of All Code* [online]. Krebson Security, november 2021 [cit. 2021-12-21]. Available at: <https://krebsonsecurity.com/2021/11/trojan-source-bug-threatens-the-security-of-all-code/>.
- [56] SHARMA, A. *Malicious PyPI packages with over 10,000 downloads taken down* [online]. 2021 [cit. 2022-04-12]. Available at: <https://www.bleepingcomputer.com/news/security/malicious-pypi-packages-with-over-10-000-downloads-taken-down/>.
- [57] STACKHAWK. *Command Injection in Python: Examples and Prevention* [online]. 2021 [cit. 2022-02-16]. Available at: <https://www.stackhawk.com/blog/command-injection-python/>.
- [58] STANDARDS, N. I. of and TECHNOLOGY. *CYBERSECURITY* [online]. 2022 [cit. 2022-01-15]. Available at: <https://www.nist.gov/cybersecurity>.



- [59] STINNER, V. [*Security-announce*] *Typo squatting and malicious packages on PyPI* [online]. 2022 [cit. 2022-02-10]. Available at:  
<https://mail.python.org/pipermail/security-announce/2017-September/000000.html>.
- [60] SUPAKEEN. *Dangers in Python's standard library* [online]. 2022 [cit. 2022-02-16]. Available at:  
<https://supakeen.com/weblog/dangers-in-pythons-standard-library.html>.
- [61] TATUSKO, J. *Absolute vs Relative Imports in Python: Overview* [online]. 2022 [cit. 2022-02-10]. Available at:  
<https://realpython.com/lessons/absolute-vs-relative-imports-python-overview/>.
- [62] TEAM, O. T. . *A01:2021 – Broken Access Control* [online]. 2021 [cit. 2022-02-24]. Available at: [https://owasp.org/Top10/A01\\_2021-Broken\\_Access\\_Control/](https://owasp.org/Top10/A01_2021-Broken_Access_Control/).
- [63] TEAM, O. T. . *A02:2021 – Cryptographic Failures* [online]. 2021 [cit. 2022-02-24]. Available at: [https://owasp.org/Top10/A02\\_2021-Cryptographic\\_Failures/](https://owasp.org/Top10/A02_2021-Cryptographic_Failures/).
- [64] TEAM, O. T. . *A03:2021 – Injection* [online]. 2021 [cit. 2022-02-24]. Available at:  
[https://owasp.org/Top10/A03\\_2021-Injection/](https://owasp.org/Top10/A03_2021-Injection/).
- [65] TEAM, O. T. . *A04:2021 – Insecure Design* [online]. 2021 [cit. 2022-04-12]. Available at: [https://owasp.org/Top10/A04\\_2021-Insecure\\_Design/](https://owasp.org/Top10/A04_2021-Insecure_Design/).
- [66] TEAM, O. T. . *A05:2021 – Security Misconfiguration* [online]. 2021 [cit. 2022-04-12]. Available at: [https://owasp.org/Top10/A05\\_2021-Security\\_Misconfiguration/](https://owasp.org/Top10/A05_2021-Security_Misconfiguration/).
- [67] TEAM, O. T. . *A06:2021 – Vulnerable and Outdated Components* [online]. 2021 [cit. 2022-04-12]. Available at:  
[https://owasp.org/Top10/A06\\_2021-Vulnerable\\_and\\_Outdated\\_Components/](https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/).
- [68] TEAM, O. T. . *A07:2021 – Identification and Authentication Failures* [online]. 2021 [cit. 2022-02-24]. Available at:  
[https://owasp.org/Top10/A07\\_2021-Identification\\_and\\_Authentication\\_Failures/](https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/).
- [69] TEAM, O. T. . *A08:2021 – Software and Data Integrity Failures* [online]. 2021 [cit. 2022-04-12]. Available at:  
[https://owasp.org/Top10/A08\\_2021-Software\\_and\\_Data\\_Integrity\\_Failures/](https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/).
- [70] TEAM, O. T. . *A09:2021 – Security Logging and Monitoring Failures* [online]. 2021 [cit. 2022-02-24]. Available at:  
[https://owasp.org/Top10/A09\\_2021-Security\\_Logging\\_and\\_Monitoring\\_Failures/](https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/).
- [71] TEAM, O. T. . *A10:2021 – Server-Side Request Forgery (SSRF)* [online]. 2021 [cit. 2022-02-24]. Available at:  
[https://owasp.org/Top10/A10\\_2021-Server-Side\\_Request\\_Forgery\\_%28SSRF%29/](https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/).
- [72] VERACODE. *Insecure Usage Of Tempfile.mktemp() Vulnerability in the numpy library* [online]. 2014 [cit. 2022-04-28]. Available at:  
<https://www.sourceclear.com/vulnerability-database/security/insecure-usage-of-tempfilemktemp/python/sid-2224>.

- [73] WEIDMAN, A. *Regular expression Denial of Service - ReDoS* [online]. 2022 [cit. 2022-02-22]. Available at: [https://owasp.org/www-community/attacks/Regular\\_expression\\_Denial\\_of\\_Service\\_-\\_ReDoS](https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS).