UCLouvain

epl

École polytechnique de Louvain

Strengthening software security of Odoo

An integrated approach using Semgrep with rule-based modification and optimization

Authors: **Charles Lohest, Romain Tourpe**
Supervisor: **Axel Legay**
Readers: **Charles Pecheur, Florian Vranckx, Martin Trigaux**
Academic year 2022–2023
Master [120] in Computer Science and Engineering

Abstract

This thesis presents an integrated approach to strengthen software security of Odoo, a popular open-source ERP and CRM system. The approach employs a comprehensive methodology that integrates risk analysis based on the CIA triad, static code analysis using the open-source tool Semgrep, rule modification, and an efficient algorithm for duplicate rule detection based on M. Legast's thesis. The research aims to enhance the identification and mitigation of security flaws within the Odoo software and contribute to the field of software security. The insights gained from this research have the potential to apply not only to Odoo but also to other software systems, facilitating the development of more secure and resilient applications by improving rules creation and security flaws detection.

Acknowledgement

First and foremost, we would like to thank each and every person who accompanied us throughout the process that led to the writing of this thesis.

We would like to express our sincere gratitude to Mr. Axel Legay, our supervisor, for his valuable advice, regular and constructive feedback, his trust in us, and his optimism throughout the supervision of our thesis.

We would also like to thank Florian, Martin, Xavier, and the rest of the Odoo security team who supported and helped us throughout the year by answering our numerous questions.

We want to thank especially Olivier Lohest, Emmanuel Tourpe and Herve Tourpe for their precious advice after spending a lot of their time reading our work. This guidance helped us to improve our thesis and produce this final result.

Finally, these acknowledgments would not be complete without a warm thought for our parents, siblings, friends, and roommates for their unwavering support and encouragement.

Contents

Introduction

With the increasing reliance on software systems in various domains, ensuring the security of computer systems has become a paramount concern. Vulnerabilities and flaws in software applications can expose sensitive data, compromise user privacy, and lead to financial losses or reputational damage.

Consequently, robust security measures and comprehensive security assessments are essential to identify and mitigate potential risks. This thesis focuses on software security within the context of Odoo[5], an open-source enterprise resource planning (ERP) platform widely used for business management.

We will introduce in Chapter 1 the developer tutorial provided by the Odoo community, which serves as a valuable resource, guiding us through the setup and customization process, and applying the groundwork for the subsequent phases of the research.

In Chapter 2 we will present the methodology we used during our thesis. This methodology will explain the functioning of the three main chapters of the thesis which are risk analysis, the use of Semgrep and rule modification, and the development of our algorithm.

Specifically, our research investigates the detection and mitigation of security flaws by employing a comprehensive methodology that includes a risk analysis in Chapter 3, static code analysis using Semgrep in Chapter 5, rule modification in Chapter 4, and developing an efficient algorithm for detecting duplicate rules in Chapter 5. By addressing these key aspects, this thesis aims to enhance the security assessment process for Odoo, contributing to the broader field of software security.

The first step of our methodology involves conducting a risk analysis to identify potential vulnerabilities and threats within the Odoo software. By applying the CIA triad (Confidentiality, Integrity, and Availability) principles, we assess the

security posture of the system and prioritize the potential risks. This analysis provides a solid foundation for subsequent stages of the research, enabling us to focus on the most critical areas of concern.

Next, we leverage the power of static code analysis using Semgrep[7], an open-source tool designed for detecting code flaws and security vulnerabilities. By configuring and customizing Semgrep rules, we systematically analyze the Odoo codebase to identify potential security weaknesses. The effectiveness of Semgrep lies in its ability to perform static analysis, providing insights into potential vulnerabilities without the need for code execution. Through close collaboration with the Odoo security team and utilizing their expertise, we further refine the Semgrep rules to ensure their relevance and accuracy for the Odoo software.

Building upon the Semgrep analysis, we address the issue of duplicate rules, which can impede the efficiency of the security assessment process. To tackle this challenge, we based ourselves on the work of Magali Legast [15] who worked on an algorithm to detect overlapping rules in energy consumption. From her work, we developed an algorithm specifically designed to identify and eliminate duplicate rules within the given set. The algorithm significantly reduces computational overhead by streamlining the rule evaluation process, enabling a more effective and expedient analysis.

In summary, this thesis aims to contribute to the field of software security by improving the security assessment process of Odoo. By employing a comprehensive methodology that integrates risk analysis, Semgrep static code analysis, rule modification, and an efficient algorithm for duplicate rule detection, we strive to enhance the identification and mitigation of security flaws within the Odoo software. The findings and insights gained from this research have the potential to apply not only to Odoo but also to other software systems, facilitating the development of more secure and resilient applications.

Chapter 1

A journey in Odoo's software

1.1 What is Odoo?

According to Wikipedia[3], Odoo is an integrated management software including numerous modules aimed at meeting the various needs of their clients, who are primarily small and medium-sized enterprises (SMEs). What sets this company apart is its commitment to being open-source, which means that the majority of its code is available online and accessible to anyone, with only 20% of its functionalities being paid for, allowing them to sustain its business needs. As of today, Odoo has over 2000 employees and aims to expand its operations to numerous countries outside of Belgium. All the source code can be found, modified, and used on github[4].
Odoo has different usages and ways to be deployed. It is in a first way deployed on Odoo.com to be used by the employee of Odoo company. They use it to manage the bill features, subscriptions, and everything related to the company services. Another utilization is software as a service. It means that clients create an instance on the Odoo servers and run the code provided by Odoo without any modification and the Odoo company takes charge of all operations, e.g., the hosting, database management, backup, security... The other utilization is a platform as a service. The customers use Odoo.sh infrastructure to run some custom code however, they still manage the backup and security. However, those clients can have security leaks due to their custom code and then can crash some parts of the application such as the database. Finally, on-premise, some clients that deploy the Odoo software locally and manage everything by themselves (database, backup, security...). Regarding the "on-premise", Odoo is not responsible anymore for the application and the client has to update correctly its application to maintain security.

1.2 Odoo developper tutorial

As we were going to work on the source code of Odoo, it was important to familiarize ourselves with the coding techniques as well as the interface and software. That's why we had to complete a tutorial provided on Odoo's official website, which is mandatory for developers to follow when starting at the company[10].
The tutorial teaches how to set up the necessary environment for the functioning of Odoo. We prepare the database using the pgAdmin software and install the source code directly through Odoo. Then, the server is launched through the odoo-bin file on Windows and is accessible on port 8069. We now have access to Odoo and we can install some Apps. The next part of the tutorial teaches us how to add a custom application to the software how to configure its permissions and how to create some basic views for this application. The following chapters of the tutorial help us to create and define fields to use in our model and how they are related to the database through the ORM (object-relational mapping) designed by Odoo. When the module is initiated all the fields are directly set in the database with their type and a potential default value if defined. Here is an example of what a basic module definition looks like :

```
class Property(models.Model):
    _name = "estate.property"
    _description = "estate property"

    name = fields.Char(required=True,default='Unknown')
    description = fields.Text(compute="_compute_description")
    postcode = fields.Char()
    address = fields.Char()
    date_availability = fields.Date(copy=False,
    default=fields.Datetime.add(fields.Datetime.today(),
    months=+3))


    asked_price = fields.Float(required=True)
    expected_price = fields.Float(required=True)
    selling_price = fields.Float(readonly=True,copy=False)
    bedrooms = fields.Integer(required=True,default=2)
    amount = fields.Float(compute="_compute_total",
    inverse="_inverse_total")
    garden_orientation = fields.Selection(
        [
            ("north", "North"),
            ("south", "South"),
```

```
            ("east", "East"),
            ("west", "West"),
        ]
    )
```

Listing 1.1: Initiating the model estate property with initial field and value directly integrated into the database.

The rest of the tutorials focus on how to connect modules and views, improve certain views, and create sub-modules within them to create a final version of a module that can be included in the default Odoo application. The tutorial also teaches how to use evaluation functions to perform calculations and certain actions directly from the web page in a secure manner.

The goal of this tutorial is to understand how the code and application work. It also provides developers with a clearer vision of the expected code structure and how to implement it. Although the tutorial takes a lot of time to fully immerse oneself in, it is advantageous in the long term for both the company and developers as it saves time and enhances understanding for both parties involved. Furthermore, we discovered that it was preferable to avoid running Odoo on a Windows operating system since the application is made to deploy on a Linux OS.

Chapter 2

Methodology

This section presents the methodology we used to investigate and enhance the software security of Odoo. The research methodology we have adopted for this study involves a combination of risk analysis, static code analysis using the Semgrep open-source tool, rule integration, rule modification, and the development of an algorithm to detect duplicate or overlapping rules.

Our research approach is based on multiple layers. First, we started by conducting a detailed analysis of the Odoo framework, using the CIA triad. The CIA triad is one of the best model crafted to provide direction for information security policies within an organization. This part helped us to detect the potential risks and gave us insight into the area that required further investigation. Then, we used the open-source tool Semgrep for static code analysis. We decided to use a static code analysis tool since this way we can perform the analysis without the actual execution of the program[**?**], which makes it faster to analyze. We will analyze in Chapter 4.1 the different tools we have taken into consideration for our thesis. By applying Semgrep[7] to the Odoo code base, we aimed to identify security issues such as SQL injection, improper identification, or unsafe data handling. After analyzing the different flags provided by Semgrep and Odoo's rules, we decided to analyze and modify them specifically for Odoo so that they would give fewer false positives while keeping the integrity of the rules for their code base. And finally, we modified M. Legast[15] algorithm that helps identify duplicate and overlapping energy-consumption rules into an algorithm that identifies duplicate and overlapping YAML syntax-based rules.

We will use a variety of data in this thesis. We use Odoo's open-source code base[4] together with the "enterprise" code base for the code analysis. We will also use the open-source Semgrep tool. The goal is to run the static analyzer[7] on Odoo's code base, to gather more information about potential vulnerabilities in the

framework. We also received a lot of guidance from our collaboration with Odoo's security team. They have provided us with expert knowledge and invaluable information about the Odoo framework, such as best practices and recommendations specific to Odoo, as we will see in 3.5, 3.4. Moreover, we had the support of our teacher, Mr. Legay, who gave us lots of valuable information and advice for our work. We also found valuable information online by researching academic papers, official documentation, or security standard and framework. The combination of all these sources formed the basis for conducting our thesis.

The risk analysis we conducted aimed to assess and evaluate potential security risks and threats within the Odoo software. For this analysis, we used the well-established principles of the CIA triad, which encompasses Confidentiality, Integrity, and Availability. For this purpose, we followed a few steps. First, we have identified Odoo's different critical assets. Then, we conducted a threat assessment, followed by a vulnerability analysis. And finally, have evaluated the risks and provided different solutions to these problems. From this risk analysis, we gained insight into the software security in Odoo.

Static code analysis played a crucial role in our evaluation of the security of the Odoo software. It involved the utilization of the Semgrep tool to make a static analysis of the code base. We started by applying Semgrep's rules on the entire Odoo code base. Then, we analyzed the results obtained by this static analysis and discussed some cases with the security team of Odoo to have a better understanding of the behavior of their code in different situations. Afterward, we understood and analyzed Odoo's regular expressions given by the security team and designed it to be run at each push to flag potential security flaws. We adjusted them so that we could translate those rules into the Semgrep syntax. By systematically scanning the code, applying customized rules, and analyzing the results, we gained a comprehensive understanding of the software's security landscape. During the static code analysis phase, we recognized the need to tailor the security rules of the Semgrep tool to meet the specific requirements and characteristics of the Odoo software.

The rule modification process involved sorting, adapting, and refining the existing Semgrep rules and Odoo's rules to enhance their effectiveness in identifying vulnerabilities within the Odoo code base. To improve the rules, we had to examine numerous flags to understand and establish which flagged code patterns could be considered harmless and which ones should continue to be monitored. Therefore, we analyzed many flags that allowed us to modify the rules and significantly reduce the number of false positives. Additionally, we sorted the rules provided by Semgrep beforehand to keep only the ones we consider most relevant and interesting to fit

with Odoo's web application. By customizing and enhancing the rules, we improved the tool's capability to quickly detect vulnerabilities within the code base.

During discussions with the security team, it became evident that they expressed concern and frustration regarding the processing time required for their rules to be applied throughout the code base. Therefore, reducing the number of duplicates in a rule set is crucial. Consequently, the development of an algorithm to check for duplicate rules became obvious. The algorithm we worked on aims to optimize the rule evaluation process by identifying and eliminating duplicate or overlapping rules, thereby reducing computational overhead. We did so by modifying Magali Legast's work[15] on WeSmart[18]. We had to modify the parser, comparison function, and duplicate rule identification for the algorithm to be successful. The algorithm's design and implementation were guided by considerations of computational efficiency, accuracy, and applicability to the specific requirements of the research.

Chapter 3

Risk Analysis

3.1 CIA

To conduct a risk analysis for Odoo and its source code, the CIA triad model has been chosen as the guiding framework. As a reminder, CIA stands for Confidentiality, Integrity, and Availability. This model has been extensively discussed and improved over the years to come to what it actually is. The CIA triad is an approach that helps identify and mitigate risks associated with protecting sensitive information and ensuring its proper usage by analyzing those three different parts of security. We chose this well-defined framework for two main reasons. Firstly, this is a very familiar concept for us since it has been widely explained in several of our classes. Secondly, the CIA triad is a fundamental concept in the information security field.

To effectively apply the CIA model, information will be harvested from different sources. First, we will use publicly available information on Odoo's official website[5]. This includes their documentation[6], available security-related disclosures, or best practices provided by Odoo. Understanding the security measures and protocols recommended by the platform can help in determining the level of confidentiality, integrity, and availability provided by Odoo.

Additionally, we will also get information for the risk analysis by searching in the code itself. Thanks to reviewing the source code and understanding the implemented security measures, we can have a clearer picture of the risks encountered by the company. Therefore, we have examined some access control mechanisms, encryption practices, error handling procedures, and other security-related aspects relevant to Odoo.

Moreover, we discussed a lot with Odoo's security team. These discussions with the employees were without doubt a valuable source of information. Engaging with individuals responsible for the development, maintenance, and security of the code base has provided information about risk mitigation measures, incident response protocols, and some observed security incidents or weaknesses.

By combining information from these different sources, a comprehensive assessment of the risks to confidentiality, integrity, and availability has been conducted. The analysis involves identifying potential threats, assessing their likelihood and potential impact, and proposing appropriate risk mitigation strategies. The objective is to ensure that the company's code is robustly protected, to minimize the risks of unauthorized access, data manipulation, or service disruptions. Note that the security depends a lot on the utilization of the Odoo software made by the client. The security will be different if the client uses it as a SaaS, PaaS, or on-premise as explained in Chapter 1. Furthermore, some security features are made for the employee of the company who uses the application in a different way than their clients. As a consequence, they have a different utilization of the web application.

3.1.1 Confidentiality

Confidentiality involves the efforts of an organization to make sure data is kept private. For this purpose, access to information must be controlled to prevent the unauthorized sharing of data intentionally or accidentally. A key component of maintaining confidentiality is making sure that people without proper authorization are prevented from accessing assets important to the business. Conversely, an effective system also ensures that those who need to have access have the necessary privileges.

For example, those who work with an organization's finances should be able to access spreadsheets, bank accounts, and other information related to the flow of money. However, the vast majority of other employees and perhaps even certain executives may not be granted access. To ensure these policies are followed, strict restrictions have to be set in place to restrict access.

There are several ways confidentiality can be compromised. This may involve direct attacks aimed at gaining access to systems the attacker does not have the right to see. It can also involve an attacker making a direct attempt to infiltrate an application or database so they can take data.

These direct attacks may use techniques such as man-in-the-middle (MITM) attacks, where an attacker positions themselves in the stream of information to

intercept data and then either steal or alter it. Some attackers engage in other types of network spying to gain access to credentials. In some cases, the attacker will try to gain more system privileges to obtain the next level of clearance.

However, not all violations of confidentiality are intentional. Human error or insufficient security controls may be to blame as well. For example, someone may fail to protect their password, either to a workstation or to log in to a restricted area. Users may share their credentials with someone else, or they may allow someone to see their login while they enter it. In other situations, an application may not properly encrypt communications, allowing an attacker to intercept some information. Also, an attacker may steal hardware, whether an entire computer or a device used in the login process and use it to access confidential information.

To fight against confidentiality breaches, one can classify and label restricted data, enable access control policies, encrypt data, and use multi-factor authentication (MFA) systems. It is also advisable to ensure that all staff has the training and knowledge they need to recognize the risks and avoid them.

To analyze confidentiality concerning Odoo's design and security measures, we will examine specific subsections of the confidentiality part to understand how they safeguard against security breaches such as SQL injection or XSS (cross-site scripting). The following subsections will focus specifically on Odoo's confidentiality measures. These sections will cover the measures implemented by Odoo to enhance confidentiality, as well as any observed vulnerabilities or unusual findings discovered during the analysis.

Cryptography security

A cryptography security policy is implemented to protect sensitive data such as passwords in the web app. These passwords are transformed into hashes using the PBKDF2 + SHA512 function. The principle of this algorithm is that it applies a hash function, encryption, or HMAC to a secret phrase, which will serve as the initial vector for the creation of the cryptography key. Salt (= a random number) is then added to this key, and the operation is repeated many times to generate a key that will be used to encrypt any sensitive content that needs to be kept out of reach of unauthorized access.

With this method, it will be difficult, if not extremely difficult, to decrypt encrypted content. The only way to verify a password is to compare the hashes. Therefore, it is extremely difficult to recover a password if it is forgotten, and the user will be constrained to reset it.

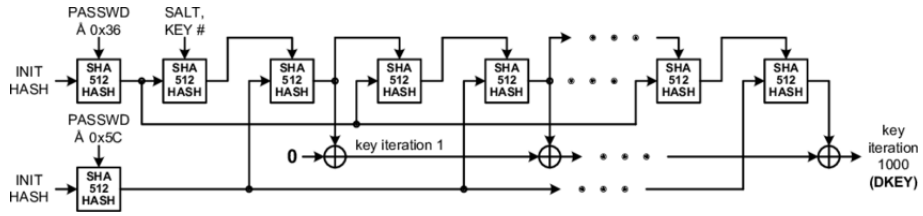

Figure 3.1: Schema of PBKDF2 + SHA512
[8]

While this method provides good confidentiality, it requires particular attention to phishing attacks that aim to steal Odoo credentials and reset passwords by posing as someone else. To prevent this, the principle of two-factor authentication should be more widespread. Currently, it is mandatory only for Odoo administrators, but the company is allowing this option for its clients. It is currently only an optional feature. Additionally, Odoo has an internal phishing prevention policy which we will cover later in the thesis.

Authentication by phone is also preferable to email because a phone is a more reliable physical security method. Returning to the password hash, one could consider the idea of simply comparing hashes by observing the response of the server provided in the logs or using tools such as John the Ripper or a rainbow table, which are the most popular and effective tools for cracking hashes. However, the use of the current cryptography algorithm and the significant number of iterations combined with the use of salt and a secret phrase make this kind of attack extremely difficult. It is preferable to limit the number of attempts available during an authentication attempt, which can be set up from version 12 of Odoo but is not mandatory or implemented by default.

Finally, during data exchanges, particularly sensitive data, are sent in encrypted form using the HTTPS protocol. This security measure makes it challenging for any interception or unauthorized modification of traffic to occur through a proxy. This security is combined with an OpenSSL web server which checks the validity of TLS (Transport Layer Security) certificates provided with the data transferred. We will cover the analysis of the servers using Open SSL (Secure Sockets Layer) Lab [19] later in the integrity section.

User access

The Odoo software allows to create different access profiles to reflect the organization's roles and responsibilities. It encompasses various groups that correspond to specific roles and functionalities, including internal users, salespersons, and HR officers. These groups not only cover business-related tasks but also encompass technical aspects such as multi-currencies, access to the export feature, and displaying the lead menu. Each group can be associated with models, views, and access rights rules following the Create, Read, Update, and Delete (CRUD) paradigm. For instance, an internal user may have read-only access to all sales orders, while a salesman can create and modify sales orders but only those assigned to them. A sales manager, on the other hand, may have full access to create and modify all sales orders. The Administrator group, known as base.group_system, possesses extensive access rights and may automatically inherit access from other groups (e.g., being added as a sales manager). However, it operates within the same rights framework as other groups, following the established system of access rules. Note that Odoo's internal organization group is also defined by the Odoo software.

We learned that customer support from Odoo has full access to their client's accounts to directly reproduce the manipulations that can cause problems. However, they do not have direct access to sensitive data such as passwords, since they connect with an admin account. Although they are only supposed to access the necessary files, they nevertheless have full access and administrative permissions when connecting to client accounts, which can raise questions about confidentiality and also jeopardize integrity, another point of the CIA triad.

To ensure a minimum privacy policy, the user is still notified of the administrator's visit and can therefore always check on their own what is happening. Furthermore, two-factor authentication is mandatory for all Odoo's employees' accounts, whether by email or phone, although the latter remains more secure as mentioned previously. When connecting to the client's database, modifications are impersonated, meaning that they appear to be made by the client and not the admin. However, this method is used because the goal is to see the actions that a user would perform on their database, which avoids repetitive exchanges and makes investigations more efficient.

Additionally, adding a separate user to the database can be burdensome in terms of billing, database usage, and server memory usage, so it is preferable to use the same user as the client. In addition to the previous information, it should be noted that Odoo's goal is to eliminate this impersonation access method and instead combine logs with admin access to client accounts, to have a clear overview

of what actions were taken during the connection to the client's account to ensure the confidentiality and have a track of the different modification made on the client's account.

3.1.2 Integrity

Data integrity is a critically important aspect of security, as it ensures the accuracy and reliability of information within an organization's systems and processes. We will begin by defining the critical assets and processes of Odoo that must maintain their integrity to ensure the accuracy and reliability of the information. Next, we will evaluate the threats and vulnerabilities that could lead to unauthorized alteration of data or systems, such as data manipulation attacks, programming errors, or human errors. We will then assess the potential impact on the company if data integrity were compromised, such as financial losses or business decisions based on incorrect information.

Odoo's critical assets

When analyzing the integrity of Odoo's vital assets and processes, several considerations come to mind. Taking a global perspective, we can divide the assessment of their integrity into two main aspects. We will also divide the assessment between odoo.com (for the company) and the Paas/Saas services (for clients).

The first aspect regards the integrity of all databases (odoo.com, Paas, Saas) that are made available through the Odoo platform. Ensuring the integrity of these databases involves verifying the accuracy, consistency, and reliability of the stored data. Therefore, we have to verify the data validation, data encryption, and access controls to prevent unauthorized modifications or tampering.

Then there is Odoo's responsibility to keep the integrity of the connection between a user and the server. This means protecting against any kind of attack that could compromise the integrity of the information, by having a secure and trustworthy communication channel between the user and the server.

Threats and vulnerabilities

There can be multiple types of threat actors and vulnerabilities for such a big company with such a large code base. It is important to define the different potential threat actors to know their possible goals or motivations, and therefore know how to protect against them. Threat actors include cyber-criminals who seek financial gain, hacktivists who seek visibility, insider threats, and competitors. But it also

could be an employee victim of social engineering. Now that we know about the threat actors, we can discuss the different known vulnerabilities for integrity.

First, we will analyze the security of the communication channel between the server and the user. This channel can be a vector for such attacks as MITM attacks. Therefore, we have used Qualys SSL lab[1], which assesses the security of a website's SSL/TLS implementation. It evaluates various aspects such as the SSL certificate, protocol support, cipher strength, and other security-related parameters. We used it on odoo.com to gather more information about the communication certificate when communicating with Odoo's server.

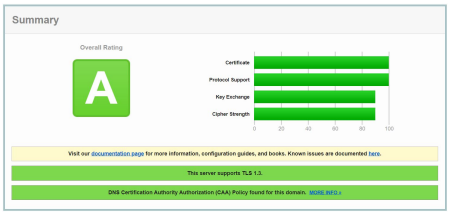

Figure 3.2: Qualy SSL analysis summary

Figure 3.2 shows that Odoo's server got an A grade. Yet, the best grade a server can obtain is an A+ grade. After a discussion with the security team, this should not be an issue. They do have a maximal compatibility with the browser and therefore a secure connection can be established with the servers. The reason they do not have an A+ grade is because they still not have phased out TLS 1.0, and TLS 1.0 no longer meets the security standards. However, they already have phased out this unsafe version for their customer hosting service (Saas, Paas). [2]

Another threat concerning integrity that cannot be overlooked is losing information due to faulty hardware. Hardware failures can lead to the permanent loss or corruption of critical information. On the Odoo website, we can read that in the worst case, their RPO (Recovery Point Objective) is 24 hours. The worst case happens if an entire data center is unreachable for a long time. Odoo does keep 14 full backups of each database for 3 months. They will keep these backups on 3 different data centers and at least two different continents. The RPO is 24 hours because they update 2 other copies of the databases every 24 hours.

Another aspect is the physical access to the hardware. The integrity of the hardware can be compromised if an attacker gains physical access to Odoo's cloud

[1]https://www.ssllabs.com/ssltest/
[2]https://www.odoo.com/fr_FR/security-report

infrastructure. Physical access represents a severe threat to integrity because it enables unauthorized server access and potential tampering with server connections. However, Odoo does not directly manage its cloud infrastructure; instead, it relies on the services provided by OVHcloud for odoo.com, and Saas and Paas rely on Google Cloud. As a result, Odoo considers the security of this aspect to be the responsibility of OVH and Google and trusts it concerning the security since they meet the standards (PCI-DSS certification [31], ISO/IEC 27001 certification [30], SOC 1 TYPE II and SOC 2 TYPE II certificate[32][33]).

Finally, we will discuss the database access. As we discussed before, secure and limited database access are primordial for the integrity of the data. The most popular attack to tamper a database, according to OWASP top 10 [17] would be an injection attack (XSS, SQL, ...). According to Odoo security web page[12], Odoo protects his data from both of them. "Odoo relies on an object-relational mapping (ORM) framework that abstracts the construction of queries and prevents SQL injections by default. In particular, developers do not create SQL queries manually, they are generated by the ORM and the parameters are always correctly escaped. The framework also escapes all expressions rendered in views and pages, which prevents XSS. Developers must specifically mark expressions as "safe" to include in rendered pages." Therefore, we can consider Odoo safe from injection attacks as long as the ORM is not compromised.

Impact on Odoo

If an attacker discovers a vulnerability in one of the integrity aspects, this could have a huge impact on the company. The impact could either be a direct financial loss if the attacker can compromise the connection between the user and the server, if databases are compromised or data becomes corrupted, essential business operations may be disrupted and finally they could suffer from a bad reputation. It is therefore vital for such a company with around 5.000.000 users to ensure the integrity of its data.

3.1.3 Availability

Availability is the last pillar of the CIA triad. Availability is crucial to the daily operations of a business like Odoo. If data or servers were not to be available anymore, the enterprise would be stopped. Therefore we have to analyze the implemented techniques to guarantee availability in the Odoo framework. According to Odoo security web page [12], the worst case scenario for them in terms of availability is when a data center crashes. If a data server crashes, they have an RTO (Recovery Time Objective) of 24 hours for a paid subscription and 48 hours

pour free trials. An RTO is the maximum fixed objective in terms of time to recover in case of disruption.

The most common attack in terms of availability Odoo could suffer from is a DDoS (Distributed Denial of Service). A DDoS is a type of attack that renders a service unavailable. This kind of attack does not concern Odoo directly since they are handled by the server providers (OVH and Google Cloud).

3.1.4 Defending Against Cyber Threats

Regarding cybersecurity in the web application, Odoo's designs and modules are built in a way that prevents SQL queries from being done manually. Instead, they pass through an ORM directly designed by Odoo, which constructs SQL queries itself in a secure way. This helps to prevent SQL injections if the ORM is properly used. With the ORM, items of a record can be accessed in a secure way

```
1 self.env['product.pricelist'].browse(product.env.context['pricelist']
```

Similarly, the framework provided by Odoo prevents XSS vulnerabilities by blocking the rendering of expressions that have not been validated by developers. Care should be taken to ensure that ordinary users do not obtain rights to modify qweb. This could grant them access to many rights that could lead to injections and other security issues if it is modified by the wrong person. POST requests provided without an authentication token are also blocked for security reasons, and URL manipulation is extremely difficult because it passes through a validation layer before redirection.

We can observe that the security of the web application is fairly well managed in cases where developers use good practices, and attacks come from the outside. However, if a malicious person is within the company, there are only a few methods in place to prevent an internal attack. An Odoo employee can have access to more information than they might need. However, a stricter client using Odoo as a software or as a platform could restrict as much as he wants the access of his employees to his data. Additionally, we found some concerning data left public in the source code, such as private keys.

To further improve overall security, Odoo has decided to launch campaigns focused on raising awareness about social engineering attacks like phishing, as well as providing proper training for developers to adopt best practices that can significantly reduce potential security vulnerabilities in the code. Moreover, they run a rule set focused on potential known security flaws every day to detect several pre-established and selected vulnerabilities.

Fishing Campaign

As previously mentioned, campaigns against phishing attacks are usually carried out on a large or small scale inside Odoo. They are organized as follows: the large-scale campaign targets employees randomly by sending them phishing emails. These emails consist of previously received emails that have been modified to include the name or position of the targeted individuals to make the phishing attack realistic and credible. It is not uncommon for employees to fall for these attacks, so they receive training to learn how to detect and avoid phishing attacks. These campaigns are conducted approximately twice a month. Additionally, less frequently, phishing attacks are targeted toward employees in more sensitive positions who are at a higher risk of being targeted due to the sensitive information or actions that an attacker could obtain.

To provide an example of real numbers obtained from our contacts at Odoo, out of **13946** phishing emails sent, **14209** emails were opened, and **1215** of them had the link clicked. Among those, **18** individuals entered their login/password information. Additionally, **94** attachments were downloaded. It is important to note that a single email can be opened multiple times, for example, on a computer and a mobile phone.

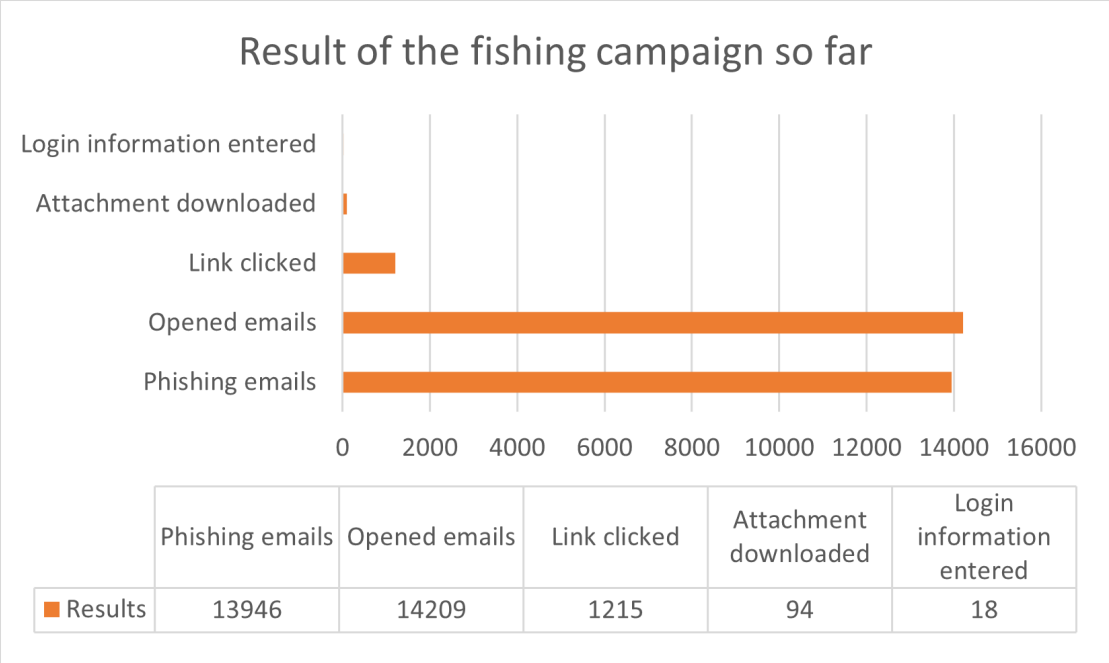

Figure 3.3: Result of the fishing campaign

Regarding the 18 individuals who fell victim to the phishing attack, **15** of them received a phishing email that was lookalike to the login page of Odoo for employees, as the simulation aimed to replicate an internal Odoo email. However, having 15 employees affected is still a significant number that can pose a substantial risk to the company. Furthermore, the remaining 3 individuals were even more trapped, as they fell for an obvious and indiscreet phishing attempt. This highlights the importance of informing and training employees about such practices, as they can prove to be highly dangerous to the company. This is even more important in the context where Odoo does not have a strict policy in terms of data access and put a lot of trust in his employees.

Good practices training

Odoo also provides training on best practices for all its developers. In this training, the company discusses the major vulnerabilities that were previously detected and how to avoid them. The training includes the following topics:

- Unsafe public method: Unsafe public methods are bad practices that should be avoided. Some methods should be private to prevent them from being accessible anywhere in the code. This is especially true for methods that handle sensitive data such as passwords. These methods should only be called when the password is entered (login and register), and not in a part of the application that doesn't necessarily require it. This helps prevent things like resetting passwords from a location that shouldn't have access to it.

- Unsafe sudo: In many parts of the code, it is sometimes necessary to make sudo calls, which stands for "superuser do". These calls that provide superuser rights can be problematic when the call is made, for example, on a write call combined with parameters provided by the application user. This could be diverted from its main purpose and be used to act as a bash command with potentially dangerous rights. To minimize problems, the OS used by Odoo is a highly limited Linux OS to avoid as many problems as possible, but it is not immune to many problems.

- Injections: Developers are also warned about bad practices that can lead to various types of injections, including cross-site scripting or SQL injection. The audit provides them with examples of code that can lead to such vulnerabilities. However, even though the queries are supposed to be pre-built by the ORM to prevent SQL injections, such attacks still occur. Certain parts of the code are particularly sensitive, such as accounting sections, where programmers often need to bypass the ORM and construct their queries. Therefore, it is important to sufficiently educate developers on using the ORM whenever

possible to prevent such vulnerabilities. Similarly, for cross-site scripting (XSS), it is advised not to link HTML content with parameters that can be modified by users or notifications with such parameters. User inputs should not be displayed before being properly processed.

```
18  class HelpdeskTeam(models.Model):
19      _name = "helpdesk.team"
20      _inherit = ['mail.alias.mixin', 'mail.thread', 'ir.needaction_mixin']
21      _description = "Helpdesk Team"
22      _order = 'sequence,name'
23
24      name = fields.Char(string='Helpdesk Team', required=True, translate=True)
25      description = fields.Text(string='About Team', translate=True)
26      company_id = fields.Many2one('res.company', string='Company')
27      sequence = fields.Integer(default=10)
28      color = fields.Integer('Color Index')
284     @api.model
285     def modify_target_helpdesk_team_dashboard(self, target_name, target_value):
286         if target_name:
287             self.env.user.sudo().write({target_name: target_value})
288         else:
289             raise UserError(_('This target does not exist.'))
290
```

Unsafe sudo!

Figure 3.4: Unsafe sudo

```
25  class res_users(osv.osv):
26      _inherit = "res.users"

36      def set_pw(self, cr, uid, id, name, value, args, context):
37          if value:
38              self._set_password(cr, uid, id, value, context=context)
39              self.invalidate_cache(cr, uid, context=context)

72      def _set_password(self, cr, uid, id, password, context=None):
76          encrypted = self._crypt_context(cr, uid, id, context=context).encrypt(password)
77          self._set_encrypted_password(cr, uid, id, encrypted, context=context)
78
79      def _set_encrypted_password(self, cr, uid, id, encrypted, context=None):
86          cr.execute(
87              "UPDATE res_users SET password='', password_crypt=%s WHERE id=%s",
88              (encrypted, id))
89
```

Figure 3.5: Unsafe public method

All of these good practices may seem obvious, but not all developers have the same training or the same best practices. Therefore, it is important to train each new developer on what is good and what is dangerous for the application to minimize the risk of error and vulnerability. However, training is often not enough

for complete security. Vulnerabilities can come from forgetfulness, misunderstanding during training, or even new functionalities added over time. Each developer is not immune to making a mistake that can ultimately slip through the cracks. This is also why Odoo has implemented a set of rules and runs them each time code is added. Each code part put on the master GitHub branch is indeed checked several times at different levels of authority before accepting the merge. However, we also know that many pull requests are processed each day and that a small problem could easily go through this step without being detected. That is why an automated detection method is an interesting option for the company.

Actual set of rules

To automate the detection of security flaws in the code, the security team has implemented a set of rules based on regular expressions. This file is run every day on the code and matches as many parts of the code as possible that correspond to the rules considered dangerous. Currently, the file matches JavaScript and Python code.

For JavaScript, the aim is to block XSS attacks using opt-in exceptions. This means that the HTML wrapper that can contain inserts is blocked, and if the insert does not contain anything, the rule is ignored. For Python, a large number of calls to modules considered dangerous are flagged, such as those related to file management, archives, and the OS. Requests are also considered dangerous, especially the use of the Request module from the HTTP library for the Python programming language. They also aim to flag sudo calls, but these are too frequent to yield useful results. Finally, the use of web templating (qweb [11]) is also considered sensitive as an XSS attack vector. The use of unescape, qcontext, and the t-raw keyword are flagged, as t-esc or t-out should be used instead to prevent the execution of JavaScript in the template data.

These rules provide significant additional security. However, due to the use of regular expressions and the fact that the rules are too broad, Odoo encounters a high rate of false positives, making it difficult to manage this tool properly. The question is how to reduce the false positive rate to obtain useful results and find a tool that may be better suited for detecting cases and creating different security rules for the code.

Chapter 4

Enhancing and Expanding Rule Set

4.1 Tool selection

To improve the detection methods implemented by the company, as discussed in the previous chapter, we have decided to use a free and open-source detection tool, that enables static analysis of source code on GitHub, GitLab, or using a local code base. This tool supports several languages, such as C#, Go, Java, JavaScript, JSON, Python, PHP, Ruby, and Scala. Additionally, it offers experimental support for 19 other programming languages. The tool also includes a large number of pre-implemented rules for these languages, providing an initial overview of potential issues upon the first launch. This tool seems suitable for working with Odoo's code as it allows us to combine the rules already implemented by Odoo using regular expressions with rules available in the tool, while also enabling easy modification of these rules.

To select this tool, we had to choose from a list of reputable code analysis tools.

- Semgrep : Semgrep is a static analysis tool based on a pattern approach to identify issues in the code. It provides a large amount of pre-defined rules based on common security vulnerabilities, coding mistakes, and best practices.
- CodeQL[16]: CodeQL is a code quality and security research tool developed by GitHub. It uses a special query syntax to describe the code patterns it searches for, and it can be used to detect security vulnerabilities in code.
- Code Climate: Code Climate is a cloud-based code-checking tool that can be used to detect security vulnerabilities, as well as code quality and performance

issues.

- DeepScan: DeepScan is a cloud-based code-checking tool that uses machine learning to detect security vulnerabilities in JavaScript and TypeScript code.
- Veracode: Veracode is a cloud-based security scanning service that can be used to detect security vulnerabilities in web applications and other types of applications.

Among all these tools, we made a pre-selection of two, CodeQL and, of course, Semgrep. We made this pre-selection based on the supported languages by these open source tools, since we were looking for a tool that supports Python and javascript. After conducting further research, we found that writing rules in CodeQL are more difficult than in Semgrep. As mentioned earlier, Semgrep is user-friendly both in terms of its functionality and rule management, which is an important aspect. On the other hand, CodeQL provides some advanced analysis capabilities, including its data flow analysis, to identify complex security vulnerabilities. Additionally, the existing rules in Semgrep are based on well-known Linters such as Bandit[22] or ESLint[23]. Furthermore, Semgrep proves to be a high-performing[24], fast [2], and efficient tool compared to other static analysis tools like Pylint[21].

The tool also offers a premium version that includes access to DeepSemgrep. However, for this thesis, we will not be using this version. DeepSemgrep, still in beta, is only available for Java and JavaScript, and unfortunately not yet for Python. It allows for a departure from static analysis, as the tool attempts to understand the entire program rather than just a single file.

4.2 Semgrep

4.2.1 Tutorial

Semgrep provides a series of tutorials for beginners, which allow them to learn the syntax of the tool and how to create rules of varying complexity.

The introduction consists of writing a basic pattern that precisely matches the desired pattern we want to flag. For instance, if we want to mark the sentence *print("hello world")*, we simply select the simple pattern and write down *print("hello world")*. This will select any sentence in a file that exactly matches the pattern. Additionally, it can also flag instances like *print("hello \World")* even if they span multiple lines. However, it does not match comments or strings, for example:

```
#print('Hello, World!')
x = "print('Hello, World!')"
```

The next part is the ellipsis operator, which is an important and practical way to exclude certain patterns that are not relevant. It is equivalent to ".*" in regular expressions, meaning that it avoids flagging the 0 to * characters that follow. For example, replacing the "1" in "logging.info(get_user(1) + "logged in")" with "logging.info(get_user(...) + "logged in")" will match any "logging.info(get_user())" and not just the ones with a "1" inside. It is also possible to match blocks of patterns by starting with an expression, followed by an ellipsis expression, and ending with the end of the block. For example:

```
print("hello")
...
print("end")
```

will flag all the code between the two prints.

Metavariables

Metavariables are useful when desiring to match something but we are not certain about the exact content we want to match. They are similar to capture groups in regular expressions. Metavariables begin with a dollar sign followed by uppercase characters. For example, $def \$FUNC(...)$: will match any function definition in the code. Metavariables can also be used to create variables. For instance, $\$X = print(”hello”)$ followed by $\$X.write()$ would match the code where a variable named X is assigned the value of "print("hello")" and then accessed using the write() method.

Pattern composition

A Semgrep file is a YAML file composed of different types of patterns depending on the rule we want to write. We use these different patterns to construct the rule.

- **Pattern-either :**
  This parameter is used to match either pattern1 or pattern2. It is useful to match variations of a certain pattern.
- **Pattern-not :**
  This parameter is used to filter out the patterns we do not want to match. This pattern-not needs to be used in combination with a pattern.

- **Pattern-inside :**
  This pattern lets us search for patterns inside a specified pattern. This "pattern-inside" is useful for searching inside function definitions, searching before or after function calls, or verifying some imports. The opposite pattern is called pattern-not-inside and filters out patterns to avoid.

- **metavariable-regex :**
  It is complementary to metavariable which allows to specify which kind of regular expression expression the metavariable should match. It is very useful if we want to match a metavariable that should contain a specific regular expression.

- **Pattern-regex :**
  It is possible to define some patterns only composed of regular expressions. Sometimes using a regular expression is simpler to define a pattern for example if we want to match $baseURL$ variables that do not start with $http$: // or $https$: // we will use pattern-not-regex: https?://.* which match http then accept 1 or 0 's' then match :// followed by any string (.*) after that.

- **metavariables-comparison :**
  This pattern is useful to restrict metavariable based on mathematical comparison. It is useful for example to match patterns for a certain length and is typically helpful in the detection of keys (false positive of Amazon key detected in the code while it is only an image URL).

- **Fix :**
  The parameter fix allows us to propose a fix to the pattern that we just flagged.

4.2.2 Presentation of the tool

The next step is to write complete rules, adapt existing ones and sort useful Semgrep rules. First, we will attempt to write a simple rule that should be able to flag a t-raw field. To do this, we will examine an example of an already implemented rule in Semgrep, such as the **Findings 662 Open Findings Group by Rule explicit-unescape-with-markup**, we can see that the rule is written like that:

```
1 rules:
2   - id: explicit-unescape-with-markup
3     message: Detected explicitly unescaped content using
4     'Markup()'.
5     This permits the unescaped data to include unescaped HTML
```

```
    which could result in cross-site scripting.
    Ensure this data is not externally controlled, or
     consider rewriting to not use 'Markup()'.
    metadata:
      cwe:
        - "CWE-79: Improper Neutralization of Input During
        Web Page Generation
          ('Cross-site Scripting')"
      owasp:
        - A07:2017 - Cross-Site Scripting (XSS)
        - A03:2021 - Injection
      references:
        - https://tedboy.github.io/flask/generated/generated/
              flask.Markup.html
      category: security
      technology:
        - flask
      cwe2022-top25: true
      cwe2021-top25: true
      subcategory:
        - audit
      likelihood: LOW
      impact: MEDIUM
      confidence: LOW
      license: Commons Clause License Condition v1.0
           [LGPL-2.1-only]
    languages:
      - python
    severity: WARNING
    pattern-either:
      - pattern: flask.Markup(...)
      - pattern: flask.Markup.unescape(...)
      - pattern: markupsafe.Markup(...)
      - pattern: $MARKUPOBJ.unescape()
```

In the rule creation and editing section of the tool, we also have a second window where it is possible to write test code. This test code is immediately related to the rule being edited, allowing us to quickly verify if some cases are working correctly. In the same example, it is presented as follows:

```
    from flask import render_template, Markup, request
from markupsafe import Markup as mkup

```

```
4  from application import app
5
6  @app.route('/markup')
7  def markup_test():
8      search_query = request.args.get('q')
9      if search_query:
10         search_query = '"{0}"'.format(
11             search_query.replace(
12             '\"', '\\\"').strip()
13             )
14     else:
15         search_query = '""'
16
17     playlist = request.args.get('p')
18     if playlist:
19         playlist = '"{0}"'.format(
20         playlist.replace('\"', '\\\"').strip()
21         )
22     else:
23         playlist = '""'
24     # ruleid: explicit-unescape-with-markup
25     return render_template('/markup.html',
26     query=Markup(search_query), playlist=Markup(playlist))
27
28 @app.route('/markup_unescape')
29 def markup_unescape_test():
30     search_query = request.args.get('q')
31     # ruleid: explicit-unescape-with-markup
32     return render_template('/markup-unescape.html',
33     query=Markup.unescape(search_query))
34
35 @app.route('/markupsafe')
36 def markupsafe_test():
37     search_query = request.args.get('q')
38     # ruleid: explicit-unescape-with-markup
39     return render_template('/markup-unescape.html',
40     query=mkup(search_query))
41
42 @app.route('/good')
43 def good_test():
44     search_query = request.args.get('q')
45     # ok: explicit-unescape-with-markup
46     return render_template('/markup-unescape.html',
```

```
47      query=Markup.escape(search_query))
```

Now that we have seen what a legitimate rule looks like and with the help of tutorials, we can proceed with writing our own rule, which, as a reminder, aims to flag the usage of t-raw. To do this, we create a new YAML file and write the rule as follows:

```
1     rules:
2   - id: odoo-qweb-t-raw
3     languages:
4     - generic
5     message: |
6       Using 't-raw' or '<t t-raw' in Odoo QWeb templates can lead
7       to XSS attacks. Make sure to validate and sanitize user
8       input before marking it as safe.
9     patterns:
10    - pattern-regex: t-raw\s*=\s*"[^"]+"
11    severity: WARNING
```

Finally, to verify our rule, it is important to write a small code snippet that includes the functions we want to flag, in this case, the t-raw functions. So, we write a test file like this one:

```
1     <div t-name="insecure_template">
2     <div id="information-bar"><t class= "hey"
3     t-raw="info_message" /></div>
4     <t class="bonjour"/>
5     </div>
```

When we run the test, Semgrep immediately detects a single flag on line two, which corresponds to the usage of a t-raw function. The test is validated, and our rule is working correctly. Of course, this is a relatively simple example, but it is still important, in more complex cases, to create corner cases to ensure that we have tested all possibilities of the rule. This helps prevent false positives or, even worse, missing true positives.

4.3 Working on Rule Set

We will now see how to classify the different rules we will be working on and the important vulnerabilities we will address. Nowadays, there are numerous vulnerabilities identified in web applications, so we decided to focus on a specific subset for this work. To accomplish this, we have decided to rely on OWASP

Top 10 [17], a widely recognized and influential list of the top ten most critical web application security risks. We will be working with the latest version of this ranking, which was released in 2021. Each element in this top 10 represents a type of risk and consists of several CWEs[25] (Common Weakness Enumerations), which are standard identifiers for software weaknesses.

By following this approach, we can narrow down the scope and focus on the most relevant cases within the OWASP Top 10 that align with our work related to the Odoo web application.

4.3.1 OWASP TOP 10

Here is the list of selected top categories for the work, along with the specific detailed CWEs chosen for each top category. The next objective will be to write/modify rules and classify them based on the different OWASP categories as well as CWEs.

A.01 Broken access control

Broken Access Control is a security issue that arises when the access control mechanisms of a web application are poorly implemented. This allows malicious users to gain unauthorized access to certain functionalities and resources that they should not have access to. These resources may include sensitive data that should remain protected. It is important to address this vulnerability to prevent unauthorized access and safeguard sensitive information.

- **CWE-200:**
  The Information Exposure weakness, as its name suggests, refers to the inadvertent exposure of sensitive data, such as passwords or user credentials. It occurs when this confidential information is unintentionally made accessible or obtained by unauthorized individuals. This vulnerability poses a significant risk as it allows malicious actors to gain unauthorized access to sensitive systems or compromise the privacy and security of users. We will search for different patterns such as:
  - Use of plain-text protocols or non-encrypted connections: http, FTP, or socket. We have to be sure that the data goes through protocols such as https and sftp. Concerning Odoo, we are mostly speaking about https.
  - Improper use of encryption, as we did earlier.
  - Incorrect access control. There, we have to look for authentication handlers such as in the werkzeug library.
  - Exposure of sensitive data in logs or error messages, such as Syslog.

- **CWE-201:**
  The insertion of sensitive information into sent data is a situation where sensitive data is unintentionally sent by the application over untrusted networks or to unauthorized individuals. For the insertion of sensitive information into sent data, we have to look for POST methods. We can check what is sent over the network for example if the POST message contains a password or a user-id, user token. For that purpose, we have to check the URL in the POST method and search for a match with these sensitive keywords.

- **CWE-352:**
  This CWE is about CSRF (Cross-Site Request Forgery). It occurs when an attacker manages to manipulate a victim into secretly performing unwanted actions on the web application on which the victim is authenticated. We need to ensure that a CSRF token is activated we have to check if the authentication is none/public or not. If it is public we allow the fact that the CSRF token is set to false. On the other hand, if there is an authentication a CSRF token should be present and used with https.

A.02 Cryptographic failures

In applications like Odoo, cryptography is commonly used to protect sensitive data, as we have seen in Chapter 3 with password encryption. Cryptography failure is a vulnerability that occurs when cryptography algorithms are used or implemented incorrectly.

- **CWE-259 :**
  This weakness is related to the use of "hard-coded" passwords. Developers directly provide passwords in plain text, without any protection or encryption. Therefore, it is important to search for functions and keywords such as "password" and replace them with appropriate encryption methods.

- **CWE-327 :**
  The Use of a Broken or Risky Cryptography Algorithm occurs when developers have used cryptography functions known to be outdated or vulnerable. For example, an algorithm that is using insufficient key lengths. For this CWE, we need to search for different cryptography algorithms and their signatures. Here is a list of the most well-known:

  - DES is a symmetric encryption algorithm that was once very popular but is now vulnerable to brute-force attacks.
  - RC4 is a stream cipher also used in the past but vulnerable to known plain-text attacks.

  - MD5 is a popular hash algorithm used for digital signatures and password hashing. It is no longer considered secure.
  - SHA-1 is also a hash algorithm used for digital signatures but is no longer considered secure.
  - RSA is a public-key encryption algorithm widely used for securing communications and digital signatures. However, it is now vulnerable to attacks from quantum computers. It is still secure for now, but not for much longer, so it is advisable to flag its usage as well.

- **CWE 331 :**
  This vulnerability is related to insufficient key sizes and poor entropy. Generating keys with insufficient length can make it easy for attackers to guess and recover the keys used in cryptography algorithms using techniques like brute force. Unfortunately, Python libraries for cryptography operations can generate keys with inadequate entropy. Therefore, it is important to monitor how these keys are generated.

A.03 Injection

Injection vulnerabilities occur when malicious data is improperly handled by an application and is able to execute hidden code within that data. These vulnerabilities can allow unauthorized access to some data, and in some cases, compromise an entire system.

- **CWE-89 :**
  This vulnerability concerns the famous SQL injection. We have tested the sqlmap tool[20] on the Odoo website to see if we could find some data through entry points with a command similar to "python3 sqlmap -u <url> –dbs", but it yielded no results. However, it is still important to verify that database calls are properly made through the ORM (Object-Relational Mapping) designed for Odoo and to check places where database access occurs, such as execute calls.

- **CWE-79 :**
  This vulnerability is also well-known and refers to cross-site scripting (XSS), which aims to make victims unknowingly execute certain actions by modifying URLs or injecting scripts into HTML. This occurs when user-supplied data is not properly handled. In the application, we can prevent an entry point for this attack by replacing "t-raw" with "t-esc" or "t-out". It is also important to be cautious when using unescaped strings, handling requests, and making calls to web templating (QWeb).

- **CWE-73 :**
  This vulnerability refers to attacks where an unauthorized person gains access to files or parts of a system they are not supposed to access. The attacker can then modify, delete, or read confidential files. This can lead to unauthorized access to sensitive information and even the execution of malicious code on the system or web application.

  To protect against this vulnerability, Odoo should carefully monitor and flag the usage of files, particularly the "open" methods in their code, as they provide access to files. Additionally, it is important to detect sensitive paths, especially when they are hard coded, such as "../etc/passwd". By identifying and addressing these issues, Odoo can enhance the security of its system and prevent unauthorized access to sensitive files.

A.04 Unsecured desing

The Unsecured Design vulnerability is a security problem that arises during the development and implementation of an application's architecture. It occurs when the risk assessment has not been adequately taken into account, and the application, as it is constructed and designed, contains vulnerabilities.

- **CWE-209 :**
  The vulnerability known as "Information Exposure Through an Error Message" occurs when developers neglect to modify default error messages that are typically generated by databases. These default messages often divulge excessive information about the underlying database, which can potentially reveal sensitive or confidential details to malicious individuals. In the context of the Odoo app, it is crucial to be cautious regarding error messages generated by the psycopg2 library, which is commonly used for database operations. These error messages should be carefully monitored to ensure that they do not inadvertently expose sensitive information.

- **CWE-256 :**
  The vulnerability known as "Unprotected Storage of Credentials" refers to the insecure storage of sensitive information such as usernames, passwords, and private keys. This vulnerability arises when these credentials are stored without adequate protection, such as storing them in plain text within code or using weak or ineffective encryption methods. Such insecure storage practices can lead to unauthorized access by malicious actors.

  By storing credentials in an unprotected manner, malicious individuals can easily gain access to these sensitive pieces of information. This can result in severe consequences, including unauthorized system access, impersonation,

and potential data breaches. To focus on this point it is important to check the credential calls and the actions made on those sensitive data to check whether they are not accessed as plain text.

- **CWE-522 :**
  The vulnerability known as "Insufficiently Protected Credentials" highlights the inadequate security measures taken to protect important and confidential information. This vulnerability shares similarities with the previous CWE discussed and encompasses situations where credentials are stored in plain text, transmitted over insecure networks, or even hard-coded in configuration files.

  The mitigation strategies for this vulnerability align closely with those of the previous CWE. It is crucial to carefully examine how credentials are used within the application and ensure that appropriate security measures are in place for their storage and transmission.

  By addressing the "Insufficiently Protected Credentials" vulnerability, organizations can strengthen the security of their systems by implementing robust encryption, secure network protocols, and secure storage practices for sensitive credentials.

- **CWE-501 :**
  The "Trust Boundary Violation" vulnerability occurs when an attacker supplies malicious input to a web application that fails to handle it correctly. This vulnerability is akin to an injection attack, as the attacker aims to breach a trusted context and exploit the application's improper handling of external input.

  To mitigate CWE-501 vulnerabilities, it is crucial to maintain well-defined trust boundaries within the application. These boundaries help establish secure contexts and enable proper validation of external input. Specifically, when input is used to interact with sensitive components like databases, robust validation, and sanitizing techniques should be implemented.

  The mitigation strategies for CWE-501 align closely with those outlined in the OWASP A03 category, which addresses injection attacks. By adopting secure coding practices, conducting thorough input validation, and employing techniques such as parameterized queries and output encoding, developers can fortify their applications against trust boundary violations and related injection vulnerabilities.

A.05 Bad security configuration

The vulnerability of Bad Security Configuration arises when the security parameters and system configuration of an application are improperly set up and implemented. For instance, leaving certain parameters at their default values or keeping an operating system configured with default settings can lead to this type of vulnerability, allowing for data harvesting and unauthorized access to the system.

- **CWE-16 :**
  The "Configuration" vulnerability encompasses weaknesses related to the incorrect or insecure configuration of system software during the setup process. This vulnerability can occur when default configurations are left enabled, granting access to potentially dangerous functions, such as operating system (OS) calls. Additionally, the use of default or weak credentials can make systems susceptible to brute-force attacks or credential theft.

  Mitigating CWE-16 vulnerabilities requires a comprehensive approach to system configuration:

  - Modify default configurations: Disable or reconfigure default settings to ensure that the system is not left in an insecure state. This includes changing default passwords, disabling unnecessary services, and closing unused ports.
  - Limit functionality to necessary components: Configure the system to only enable essential functionalities, disabling or removing unnecessary features or services that may introduce security risks.
  - Perform secure code reviews: Pay special attention to code sections that interact with critical system resources, such as OS calls, and ensure that appropriate security measures are set in place to mitigate potential vulnerabilities.

- **CWE-611 :**
  The "Improper Restriction of XML External Entity Reference" vulnerability involves the processing of XML inputs, which can result in unauthorized access to sensitive data or the execution of malicious code. This vulnerability is similar to file-related vulnerabilities but specifically pertains to XML usage. The objective is to identify the presence of XML in the code and ensure that data within XML is handled securely to prevent vulnerabilities from arising.

- **CWE-489 :**
  The presence of leftover debug code refers to the existence of debug statements that were used during the development and testing phases but were not properly disabled in the final code. This oversight can inadvertently expose

sensitive information and provide attackers with valuable insights about the system's environment.

When debug code is present in the production environment, it can reveal internal details, logging information, or other sensitive data that was displayed during the debugging process. Attackers can leverage this information to gain unauthorized access to sensitive resources or exploit vulnerabilities.

To mitigate CWE-489 vulnerabilities, it is crucial to review the code base and identify any remnants of debug code that were not disabled. Special attention should be given to sections of code that pertain to debugging, ensuring that sensitive data is not inadvertently exposed. By flagging and addressing such instances, organizations can enhance the security posture of their software applications and prevent potential information disclosures.

4.3.2 Semgrep useful rules

Since Semgrep is a powerful open-source static code analysis tool, he offered a wide range of pre-configured rules that proved instrumental in identifying security issues within the Odoo software. There are a total of **2142** rules in Semgrep, covering most existing CVE (Common Vulnerabilities and Exposures) and CWE. Some of these rules are inspired by the rules of many popular linters and checkers, including ESLint, RuboCop, Bandit, and FindSecBugs[26]. Not all of them are useful to Odoo. Several specific rules stood out as particularly useful in our analysis.

We tried to determine which rules were useful or not to Odoo. To do so, we took the library of rules and look how the rules were sorted. What is, in our case very helpful, is that most rules are sorted by language (java, python, HTML, ...). The first step was then to sort all of them and decide whether they were useful to Odoo or not. Since Odoo is mostly written in Python and JavaScript (97.1%), we could first decide to keep going with those both types of rules.

After being divided by language, the rules are divided by type of vulnerability. To decide whether a rule set (such as Flask, for instance) would be useful, we used visual studio code. In the IDE, we use the search tool and we looked for import or common function calls.

We will start with Python rules since Python is the most important language for the Odoo framework. Python represents **42.9%** of the code base of Odoo. There is **22** type of rules for a total of **397** rules in the python category. Therefore, we must only choose the useful ones, otherwise, it might become time-consuming and very slow to run all these rules. Out of the 22 categories, only 6 of them seemed to be

useful for Odoo (aws-lambda, correctness, jinja2, requests, sqlalchemy, language). Again, these rules are sometimes sorted inside these folders, and can be labeled as "best practice", "correctness", "maintainability" or "security". We focused on the first and last one, security and best practice. In total, out of the 397 rules, we only selected **83** rules. The way we selected the rules is mainly based on library import and common function calls. Most Semgrep rules are based on patterns that require a specific library. Once a library has been used in the code base of Odoo, we decide to retain the rule. By keeping only 83 rules concerning Python out of 397 (**20,9%**) in Semgrep, we can be faster to run the open-source engine, which is non-negligible for Odoo and their 1500+ contributors.

We repeated the same principle with the JavaScript rule set. Since JavaScript is a big part of the Odoo code base (**54.2%**), it makes sense to optimize the number of rules we will use. There are initially **205** rules in Semgrep concerning javascript, all sorted by type of vulnerability. Once again, we only kept the categories that interest us by systematically analyzing the code base using our IDE and rules. Out of **35** categories, we only kept some rules from 7 categories (browser, aws-lambda, audit, jquery, lang, vue, wk_html_to_pdf). Like for the Python rules, they are sometimes also sorted as "best practice", "correctness" or "security". We focused on the first and last one, security and best practice. Finally, out of the 205 rules in the JavaScript folder, we only kept **49** rules (**23.9%**) that we think were useful to Odoo. This makes a reduction of around **75%**, which is a great improvement and might be time-saving.
Finally, we can say that out of the **2142** rules in Semgrep, we only kept **132** of them. This is around **6.2%** of the rules, which is a huge improvement in terms of performance and speed of execution for the Semgrep engine.

4.3.3 Modified rules

Among all the useful rules inside the Semgrep tool, we will discuss 2 important rules for Odoo.

The first rule that caught our attention was the "detected hard-coded key" rule. This rule is about finding a hard-coded key pattern in the code, such as RSA, DSA, OpenSSH, or ENCRYPTED private key. In the case of Odoo, we can find two of them. One in the public Odoo repository, and one in the enterprise repository. In the public repository, we can read that this should have been a temporary fix for an IoT Box image, but the temporary fix is 5 years old. We asked the security team and they told us that it should not be important but we did not get more information about it.

The second rule that triggered our attention is the "sqlalchemy execute raw

query" rule. This rule is based on CWE-89: "Improper Neutralization of special elements used in a SQL". This rule has flagged **316** cases and we went through each one to make sure none of them was a threat. This rule is based on the "execute" function call and is looking for a pattern inside the execute call where there is an unsafe string formatting, such as "%s"%user_input. This rule is the rule that has flagged the most cases, which makes sense since this rule flags every single SQL request in Odoo. Yet, our goal was to reduce this number of true positives to make the rule more "correct" for Odoo. We have brought the number of flags to **147**, almost less than half of the initial flags, which is a huge improvement.

We were also triggered by the "unescape-markup" rule inside Semgrep because this rule has the second-highest number of flags. But we saw later that Odoo had a similar rule so we combined both of them before fixing them. We will explain more about it later.

Another rule that triggered our attention is the "detected AWS access key id value". An image in the Odoo code base is flagged 11 times as an AWS key value. We looked into it and the problem with this rule is that there is no length limit to the key. So we fixed this problem by adjusting the limit on both sizes of the pattern and the problem has been fixed, there is no more flag for this rule.

Odoo's rules

We have taken over the file that contained the rules written using regular expressions by Odoo. They use these rules for as a basis for code security checks on each push. We started by translating these rules into the syntax used by the Semgrep tool and executed them on Odoo's code base. As expected, we encountered a large number of alerts, with approximately **6600** new flags. However, among these, there were **3464** flags detected by a rule targeting sudo's calls. This rule is no longer used by Odoo since it is too general and triggers too many alerts. Consequently, we have decided to discard this rule as well.

Our objective is to reduce the number of flags to improve the relevance of our search and, consequently, our effectiveness. To achieve this, as we explained in our methodology, we analyzed the different flags that could be found based on the rules, aiming to identify what could be considered non-threatening and what should remain flagged as elements to be cautious about. Using these results, we were able to modify and create new rules that are consistent with Odoo's established rules, to reduce false positives.

- **Risky modules :**

  This is the most flagged rule with **(970)** alerts. It is not surprising since this target a lot of basic library from Python such as :

  - os
  - sys
  - zipfile
  - tempfile
  - tarfile
  - pathlib
  - ctypes
  - urllib

  To optimize this rule there is no way to automatize that since that depends on which functions are used. The best way to improve the rule is to run the rule on Semgrep, detect the flags, and then analyze flags one by one to see if the function flagged is safe or not. If it is safe, since it does not require input from outside and does not interact with sensitive data or does not allow to display any data, we can consider this function safe and add an exception for that method in the rule. By doing that it is possible to reduce considerably the amount of flags while ensuring the detection of dangerous functions. The analysis is not restricted to Odoo so that rule can be re-used for other applications. After spending time on analysis of the different parts of code flagged by this rule we could reduce the amount by **50%** since we now have only **420** detected flag for this rule.

- **handle requests separately from other risky modules :**

  This rule is made to detect the use of request. Request is a method implemented in Python which is used to send HTTP requests. Of course, using http is dangerous since we used to communicate with the outside. There are then good practices such as using tokens and sanitizing requests to respect when using these methods. The rule was targeting a lot of alerts by flagging import or error handling related to requests. We managed to reduce again the number of alerts by almost half going from **474** flags to **252**.

- **Odoo attribute (kn64kbnw) :**

  This rule concerns the value of attributes, especially by the use of built-in methods getattr() and setattr(). Those methods can be dangerous if used with unsanitized input or if they are used with sensitive data. Furthermore if after a getattr we want to display its value by using a print, we have to do something like print(f"attribute name: value") and not print("attribute : value") to avoid formatting leaks. For this one, we assert that the methods from where we get the attributes are safe if they are provided by the developer. So, we made a rule to allow the use of getattr when used with an attribute defined previously in a dictionary by a developer. With that method we can go from **397** to **79** flags

- **Sanitize attribute is false :**

  This rule concern the case in HTML and Qweb templating where the parameter to allow the sanitizing is false and has been set as false intentionally by the developers. That could be either a sanitize general or for attributes or even a form. This rule flags **61** cases.

- **Crypto :**

  This rule is similar to the first one since it flags every time a cipher method is called. However, we decided to avoid studying this rule because it requires knowledge of Python's cryptography methods that we don't have time to learn. However, there are **95** flags detected whit the rule made to detect dangerous cryptography algorithms explained in the Risk analysis from Chapter 3.

- **Module open :**

  This rule is made to flag every use of open. We get initially **88** flags with that rule. with external files is dangerous so this rule is very important and can not easily be reduced or precise since we can not statically see if a file is safe or not. However, we can still check inside the different flags to verify if there are safe parts of the code that is flagged and then put it as an exception in the rule.

- **T-raw :**

  T-raw is a parameter used to inject some HTML in web pages it should be avoided in any case since it allows the execution of some scripts and is

not secured any t-raw should be replaced by t-out which blocks scripting execution. There is **20** use of t-raw and they should all be flagged so that rule is just about matching t-raw expression.

- **Qweb Eval :**

  QWeb eval is used to evaluate expressions. However, using eval can pose significant risks as it executes arbitrary code, making it susceptible to severe security vulnerabilities if user input is not properly validated and sanitized.

  To mitigate these risks, it is crucial to validate and sanitize user input before incorporating it into QWeb expressions or any code that undergoes evaluation through eval. Whenever possible, it is advisable to avoid using eval and explore safer alternatives like ast.literal_eval for evaluating simple expressions or implementing a secure evaluation mechanism.

  In this rule, the eval context or qcontext is flagged as it represents the sensitive part of this process. Since authorizations are granted through the context, ensuring its safety becomes crucial. However, certifying what is safe or not within the context becomes challenging as it depends on the specific user and its contents. Nevertheless, by modifying the syntax of the regex to avoid taking the wrong part of a code of simply an import that would not be used we try to flag the real use of a q_eval_context, and we managed to reduce the number of flags from **180** to **70**, improving the rule's efficiency.

- **Copy() :**

  The copy method of this object creates a shallow copy of the parameters dictionary, which can be modified without affecting the original object. Using request.params.copy() in Odoo can potentially create security vulnerabilities if the copied dictionary is not properly sanitized or validated before being used in further processing. Concerning the rule we will just keep what was flagged initially which is a request to params.copy call.

- **cr.execute :**

  The ‘cr.execute‘ method is used in Odoo to execute SQL queries on the database. This method uses Odoo's ORM (Object-Relational Mapping) to ensure safe requests. However, it is only safe to use ‘cr.execute()‘ if the queries are correctly formatted and the method is used properly. Working

The ‘cr.execute‘ method is provided by the Cursor object, typically accessible as ‘self.env.cr‘ or ‘self.cr‘ within an Odoo model. Improper use of this method can lead to security vulnerabilities, such as the well-known SQL injection. If non-validated or unsanitized user input is incorporated into the SQL query, an attacker could inject malicious SQL code, resulting in data leaks, data corruption, or unauthorized access. This can occur when formatting is applied in queries or when the second part of the method is not used correctly. The method should be called as ‘cr.execute(query, arguments)‘, but sometimes developers mistakenly (or do not have the choice) use ‘cr.execute(query % arguments)‘, which fails to use the ORM correctly and might introduce vulnerabilities.

To mitigate these risks, the rule can be expanded to check the correctness of argument formatting and verify that the query does not contain any forbidden or unallowed formatting, such as the common expression ‘self._tables‘ often used in such ‘cr.execute‘ scenarios. The challenge lies in creating a rule that can accurately evaluate these conditions to differentiate between what is considered safe and what is considered dangerous.

To minimize risks when using ‘cr.execute‘, use parameterized queries. Instead of concatenating user input directly into the SQL query, utilize placeholders and pass variables separately. This helps prevent SQL injection attacks.

Currently, there are **313** detected flags, but by implementing these measures, the number can be reduced to **199**, which is approximately one-third of the original flags. Here is a comparison between the original rule in Semgrep and then the modified one :

Listing 4.1: cr.execute rule from odoo

```
  rules:
- id: odoo-cr_execute
  metadata:
    OWASP: A03 INJECTION
    CWE: CWE-89 sql injection
  patterns:
    - pattern-either:
        - pattern: |
            $CONNECTION.execute( $SQL + $VAL , ... )
        - pattern: |
            $CONNECTION.execute( $SQL % $VAL , ...)
        - pattern: |
            $CONNECTION.execute( $SQL.format($VAL), ... )
        - pattern: |
            cr.execute( $SQL.format($VAL), ... )
```

```
    message: EXECUTE IS SENSITIVE TO SQL INJECTION WHEN USED WITH
        INPUTS
    languages:
      - python
    severity: WARNING
```

Listing 4.2: cr.execute rule modified

```
    rules:
  - id: odoo-cr_execute
    metadata:
      OWASP: A03 INJECTION
      CWE: CWE-89 sql injection
    patterns:
      - pattern-either:
          - pattern: |
              $CONNECTION.execute( $SQL + $VAL , ... )
          - pattern: |
              $CONNECTION.execute( $SQL % $VAL , ...)
          - pattern: |
              $CONNECTION.execute( $SQL.format($VAL), ... )
          - pattern: |
              cr.execute( $SQL.format($VAL), ... )
          - patterns:
              - pattern-inside: |
                  $QUERY = $SQL + $VAL
                  ...
              - pattern: |
                  $CONNECTION.execute($QUERY, ...)
          - patterns:
              - pattern-inside: |
                  $QUERY = $SQL % $VAL
                  ...
              - pattern: |
                  $CONNECTION.execute($QUERY, ...)
          - patterns:
              - pattern-inside: |
                  $QUERY = $SQL.format($VAL)
                  ...
              - pattern: |
                  $CONNECTION.execute($QUERY, ...)
          - patterns:
              - pattern-inside: |
                  $QUERY = f"...{...}..."
```

```
37                 ...
38             - pattern: |
39                 $CONNECTION.execute($QUERY, ...)
40     message: EXECUTE IS SENSITIVE TO SQL INJECTION WHEN USED WITH
           INPUTS
41     languages:
42       - python
43     severity: WARNING
```

- **Safe evaluation :**

  The safe evaluation function is a method defined by Odoo that allows for code evaluation. However, it can be dangerous as it can execute malicious code within the application by evaluating hidden code. It takes a dictionary as an argument to define the environment and restrict access to certain methods, such as OS and print. The default local dictionary blocks many of these dangerous functions, but vulnerabilities can still be found. For instance, the "time" library allows access to OS calls which was an old vulnerability found with this method.

  The evaluation function has different modes: eval (default), exec, and single. The eval mode only evaluates Python expressions, excluding statements, assignments, and definitions. The exec mode allows for the execution of Python code, including statements like function and class definitions. The single mode is a combination of the other two, allowing for the evaluation of both statements and expressions.

  Understanding and studying this function was challenging due to its contextual nature. Modifications were made, and calls to this method were flagged. However, cases, where the local dictionary is correctly defined by the developer using ORM functions, are supposed to be safe and then set as an exception. Nevertheless, this approach should be reviewed as it does not guarantee complete safety. Although some true positives may be missed, developing a comprehensive rule to determine the safe and dangerous parts of the environment would be very difficult. Therefore, the assumption is made that the developer is knowledgeable and uses it properly. With this rule, we successfully brought down to **61** alerts instead of **72**.

- **Markup :**

  The markup function in Odoo is part of the odoo.tools module and is used to sanitize HTML content. The function helps prevent Cross-Site Scripting

(XSS) attacks by removing potentially malicious code from the inserted HTML. The markup function uses the lxml.html.clean module internally to perform sanitizing. With our modification, we went from **120** alerts down to **0**. We were able to do this because Odoo always uses the safe version of markup (markupsafe). Although the markup function is designed to improve security by sanitizing HTML content, there are still potential risks if it is not properly used. Disabling sanitizing: If the sanitize parameter is set to False, the function will not perform any sanitizing, leaving an application vulnerable to XSS attacks.

- **Unescape :**

  This rule is flagging only **4** code snippets. In Odoo, the unescape function is part of the odoo.tools module and is used to convert HTML entities in a string back to their original characters. This function can be useful when working with HTML content that contains escaped characters, such as text retrieved from a database or user input that has been sanitized. The unescape function is built on top of the html.unescape function from the Python standard library. While the unescape function is generally safe to use, there are potential risks if not used carefully: Double unescaping: If an already unescaped string is accidentally unescaped or a string is unescaped multiple times, it could lead to unexpected results, potentially introducing security vulnerabilities or causing rendering issues. Unescaping untrusted input: If untrusted input is unescaped or user-generated content is not properly sanitized, it might inadvertently introduce security vulnerabilities, such as Cross-Site Scripting (XSS) attacks. Since it does not flag a lot of cases it was not necessary to make this rule more precise.

- **Sqli linter :**

  This rule regards some comments in the code base that concern code modification. It is not directly related to security leaks but it is an easy way to find part of the code that has to be changed. Therefore, we have decided to not modify the initial regular expression which allows us to detect this kind of comment.

Complementary rules

We also have written additional rules, independent of the Odoo rule set, which include the detection of hard-coded credentials, requirements for authorization tokens when accessing APIs, and rules to identify if error messages from databases

are logged. These rules, named "odoo-(rule name)", can be found on our GitHub repository[28]. They consist of modified Odoo rules as well as rules we created ourselves. These rules are believed to be advantageous for Odoo, providing benefits. Those additional rules are directly related to the different CWE we discussed earlier and which we think were not enough developed in Odoo's rule set.

- **External control of filename or path :**
  This rule is designed to verify whether a path is correctly sanitized or not. It will flag every instance where a path is used with an "open" operation, and using regular expressions, it will allow exceptions when the path is a normal string. In other words, it rejects paths that contain special characters like ' or character sequences like ../../, which are typically paths that should not be accessed, such as "../../../../../etc/passwd". This example will be flagged accordingly. Here is the YAML representation of this rule :

```yaml
      rules:
  - id: odoo-external-control-of-file-name-or-path
    metadata:
      OWASP A03: cwe-73
    patterns:
      - pattern: open($INPUT,...)
      - metavariable-pattern:
          metavariable: $INPUT
          patterns:
            - pattern-not-regex: ^[a-zA-Z0-9_\-\."]+$
    message: THE PATH IS NOT SANITIZED CORRECTLY
    languages:
      - python
    severity: WARNING
```

  And its test file :

```python
      import os
import re

# Get user input for file name or path
user_input = input("Enter file name or path: ")

# Define a regular expression pattern to match valid file names
pattern = r'^[a-zA-Z0-9_\-\.]+$'

# Check if user input matches the pattern
if re.match(pattern, user_input):
    # Sanitize the input by removing any path separators
```

```
    sanitized_input = user_input.replace(os.path.sep, '')

    # Use the sanitized input to access the file
    with open(sanitized_input, 'r') as f:
        # Do something with the file
        pass
else:
    print("Invalid file name or path.")
    with open("../../../../../etc/passwd", 'r') as f:
        # Do something with the file
        pass
else:
    print("Invalid file name or path.")
    with open("doesn't", 'r') as f:
        # Do something with the file
        pass
else:
    print("Invalid file name or path.")
    with open("doesnt", 'r') as f:
        # Do something with the file
        pass
```

- **Sensitive data in Logs :**

  The purpose of this rule is to simply detect calls to database modules, specifically psycopg2, which is the Python module that interacts with the PgAdmin database. It will identify cases where errors from this module are displayed and flag them to prevent the disclosure of sensitive information to unauthorized individuals. The rule is :

```
      rules:
  - id: odoo-sensitive-data-in-logs
    metadata:
      OWASP:
        - A04-2021 Unsecured design
      CWE:
        - cwe-209 and cwe-256
    pattern: |
      try:
        ...
      except psycopg2.$ERRORMSG as $ERROR:
        ...
        print(...,$ERROR)
```

```
    message: INFORMATION ABOUT DB PROVIDED IN ERROR LOGS
    languages:
      - python
    severity: WARNING
```

- **Unsafe connection :**

  This rule is designed to detect the insecure use of HTTP routes, specifically when a CSRF token is not used to validate user identity. However, we do allow redirection when the route is considered public or when the authentication parameter is intentionally set to "none," as we consider it a conscious choice by the developers. However, if the parameters require authentication but the tokens are not valid or not provided, an alert will be raised.

- **Untrusted console output :**
  This rule is designed to flag console calls (error, debug, log, etc.). The purpose of this rule is to allow these calls when they consist only of a hard-coded string. However, when they are used with values using formatting, such as ‘console.error("hello there! " + content);‘, there might be data that should not be disclosed, and an error will be raised. Here is the code of this rule:

```
      rules:
  - id: odoo-unstrusted-console-output
    patterns:
      - pattern-either:
          - patterns:
              - pattern-either:
                  - pattern: $CONNECTION.$PRINT("..." + $VALUE)
                  - pattern: $CONNECTION.$PRINT("..." % $VALUE)
              - metavariable-pattern:
                  metavariable: $PRINT
                  patterns:
                    - pattern-regex: error|log|debug|warn|warning
          - patterns:
              - pattern-either:
                  - pattern: $CONNECTION.error($CONTENT)
                  - pattern: $CONNECTION.log($CONTENT)
                  - pattern: $CONNECTION.debug($CONTENT)
                  - pattern: $CONNECTION.warn($CONTENT)
                  - pattern: $CONNECTION.warning($CONTENT)
              - metavariable-pattern:
                  metavariable: $CONTENT
```

```
22                  patterns:
23                    - pattern-regex:
                          \berror\b|\blog\b|\bdebug\b|\bwarn\b|
24                    \bwarning\b|\be\b|\berr\b|\bcontent\b
25      message: The output might contain sensitive information
26      languages:
27        - javascript
28      severity: WARNING
```

Those rules are not especially flagging a lot of cases but we think that they still are useful to keeping the code base as safe as possible. We do believe that the number of flags does not always reflect the efficiency of a rule. There might be other parameters to take into account, like the fact that most of Odoo's developers might be aware of this kind of weakness.

4.3.4 Results and comparison

Following extensive work on the rules within the Semgrep tool, we successfully assessed its effectiveness by applying the tool to the source code of both Odoo and its paid extension, Odoo Enterprise. We will now compare the results between the old and the new version of the rules. For the rules that were already implemented by Odoo and which we adapted, we achieved a reduction of about **30%** in flags. Working on specific rules allowed us to significantly decrease false positives, such as those related to risky modules, requests, and database calls, as shown in the graph4.1. However, the number of flags is still not at its minimum, and it would still be possible to further reduce some rules, such as risky modules by doing some deeper investigation about the different flags and their impact on the code. On the other hand, we believe that some rules cannot be further modified to decrease the number of alerts without losing efficiency. However, this often requires a case-by-case analysis and we would have to further write down exceptions to exclude them from the rules. This kind of work is time-consuming because, for each flag, many code files often need to be further analyzed to understand the dependencies.

In total, we worked on approximately **25** rules, including those from Odoo indicated in the graph, as well as others that were not included in the graph because the results compared to Odoo did not show significant numbers. However, these rules are still relevant for the future, such as rules concerning injections, XSS, or error messages that may leak sensitive information. As previously explained, all these rules were developed and categorized based on the OWASP Top 10 and CWE, with the ranking of each rule provided.

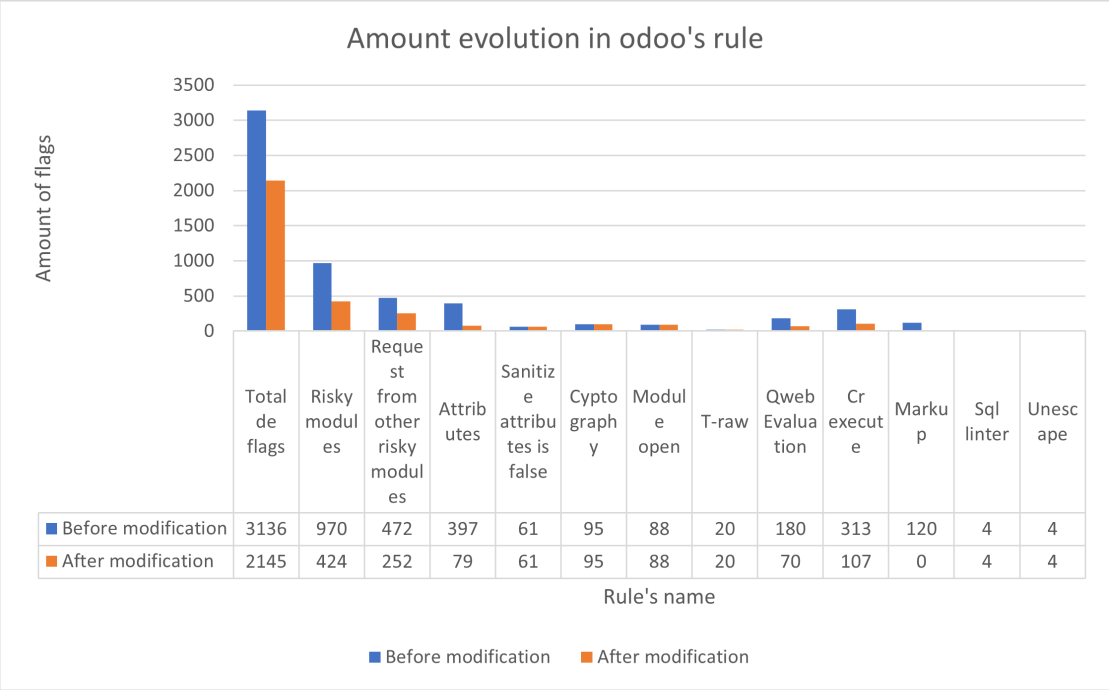

Figure 4.1: Final result after modifying the rules

Additionally, we also reviewed over a hundred rules from Semgrep to further improve efficiency for Odoo. We selected only the most relevant rules, reducing the rule base and enhancing the tool's performance. The **30%** decrease in flags achieved by these results makes this research more relevant and efficient for Odoo. It would enable them to have better code verification and administration, reducing the risk of code vulnerabilities. Indeed, a difference of approximately **1000** flags is significant and should not be overlooked.

As mentioned before we worked on many rules and it was extremely difficult to write them in this report so they can be accessed on GitHub[28].

Chapter 5

Comparison algorithm

The use of rule-based expert systems, which are applications that implement certain knowledge, in our case rules, and help to solve specific problems, such as detecting anomalies in code, proves to be very useful. However, since their knowledge is provided by humans, it is not inevitable that errors creep into the rules, especially when they become complex and numerous. In the case of this thesis, it can be seen that the implementation of rules can quickly become complex. An important aspect in addition to the effectiveness of these rules is to avoid contradictions and redundancies. Therefore, it is useful to have tools to detect such problems to facilitate the writing of new rules and improve the available rule base.

Based on the advice of our supervisor, we decided to build upon the work of a former student, Magali Legast, who developed an algorithm for detecting errors in rule-based systems, such as energy consumption rules. We had to understand her research and adapt her algorithm, which was available on her open-source GitLab page[27]. We had to modify a syntax designed for number intervals to a syntax specialized in regular expressions to suit our types of rules and correspond to the Semgrep syntax. This will help us to obtain results that improve our existing rules and future ones that will be added. Therefore, this tool is named the Relationship Identification Tool (RIT).

5.1 Role of the algorithm

The principle and role of the algorithm are well-detailed in the paper published by A. Legay and M. Legast[9]. We will summarize the main points in this paragraph. Our algorithm will be built based on a list of rules forming a matrix in which the rows are the different rules and the columns are the attributes as well as the name of the rule (name of the problem). Here is an example of what the rule matrix

should look like in our case with a few simple rules as well as how the patterns lists are constructed and how they are delimited :

Id	pattern	pattern-not	metavariable
odoo-test1	"[import os;import md5;import sha1]"	*	*
odoo-test3	"[import sha1]"	*	*
odoo-test5	"[pattern.test(...)]"	"[import os;import md5;import sha1]"	*

Table 5.1: Example table

The goal of this algorithm will be to compare the rules to each other to define whether they are completely different or if similarities can be found. We will establish a certain number of values called Inter Difference Coding (IDC). These values can be

- -1 Difference between recommendations
- 0 Disjunction between attributes
- 1 Equality
- 2 Inclusion of attribute j in attribute i
- 3 Inclusion of attribute i in attribute j
- 6 Overlap

The values chosen in the IDC are selected to facilitate the expression of relationships between rules. The difference values must be absorbent in multiplication, so we chose 0. Equality must be a prime number to enable identification via modulo, and the choice of identity 1 simplifies the identification of this condition. The inclusion values must be different from the previously defined values for ease, so we simply chose the sequence with the numbers 2 and 3. Finally, the overlap, which can be roughly considered as a product of inclusion ij and ji, is the product of the two inclusion numbers. Therefore, we get the number 6.

After that, the Inter-difference matrices (IDM) are constructed. These are matrices for each attribute that will be of size nxn, where n represents the number of rules. Those matrices are upper-triangular in which the relationships between the different rules are noted by the IDCs. Therefore, we obtain m+1 IDMs of size nxn.
For example, if we have a rule matrix that is different than the previous one but simpler as an example :

Rules	Attribute 1	Attribute 2	Attribute 3
1	[0.2,0.8]	[0.2,0.8]	[0.5,0.9]
2	[0.2,0.8]	[0.6,1.0]	[0.9,1.0]
3	[0.6,1.0]	[0.4,0.8]	[0.7,1.0]

The IDM for Attribute 1 will be :

Rules	R1	R2	R3
1	0	1	6
2	0	0	6
3	0	0	0

As we can see in the example rule number 3 for the first attribute is overlapping both the rule number 1 and 2 which are equals. Since the IDM is upper triangular we get the result of this comparison in the right upper corner.

The next step will be to construct the product matrix (PM) generated via the m+1 IDMs (m attributes + id). For this, we define a value

$$p_{ij} = \prod_{x=0}^{m} R_{ij}(x) \tag{5.1}$$

Where x is equivalent to every m attribute.

In other terms $R_{ij}(x)$ is the relation found in the IDM corresponding to the attribute x for the rules i and j. In the previous example the $R_{1,2}(1) = 1$ and $R_{1,3}(1) = 6$ such as $R_{2,3} = 6$. We can then construct the PM such that P[i,j] = p_{ij}.

Finally, to detect potential equivalences between rules, mathematical operations based on modulo are performed on the values stored in the product matrix. This allows not only the detection of connections but also the identification of the type of relationships thanks to the previously constructed IDCs and IDMs. The mathematical operations used are detailed in the article[9] and are defined as follows:

- **Disjunction :**
  - $p_{ij} = 0$
  - Condition : $P[i, j] = 0$
- **Inclusion of R_i in R_j, same recommendation:**
  - $p_{ij} = 1 \times (1^{(m-x)} \times 2^x)$

– Condition: $\begin{cases} P[i,j] \bmod 6 \neq 0 \\ P[i,j] \bmod 2 = 0 \\ P[i,j] > 0 \end{cases}$

- **Overlap, different recommendations :**
  - $p_{ij} = -1 \times \left(1^{(m-x-y-z)} \times 2^x \times 3^y \times 6^z\right)$
  - Condition : $\begin{cases} P[i,j] \bmod 6 = 0 \\ P[i,j] < 0 \end{cases}$

Where m is the total number of attributes, x is the number of inclusions ij amongst the attributes compared to y which is the number of inclusions ji, and where z is the number of overlaps.

To adapt the algorithm to our specific set of rules, which are quite different from the original ones, we had to make modifications, particularly in the parsing method. These adaptations will be discussed in the following section of this chapter. As a result, we now have lists of patterns represented by attributes in string format instead of simple values such as intervals or binary values like true/false.

The adapted algorithm will compare the rules by forming IDMs using regular expressions to compare the different strings. Our goal is to detect if rules repeat or have the same meaning. To achieve this, we simply analyze each attribute in each rule literally and check if parts of the attribute overlap, are included in others, are completely different, or if the two rules are identical. By comparing two lists of attributes from different rules, we can establish a code as defined earlier and construct the IDM and perform the algorithm. We define the different relationships between rules such as:

- **Disjunction :**

$$\forall x \in l_1, \forall y \in l_2, x \neq y => R_{ij} = 0$$

- **Equality :**

$$\forall x \in l_1, \forall y \in l_2, x = y => R_{ij} = 1$$

- **Inclusion of R_i in R_j :**

$$\forall x \in l_1, x \in l_2 \wedge \exists y \in l_2, y \notin l_1 => R_{ij} = 2$$

- **Inclusion of R_j in R_i :**

$$\forall x \in l_2, x \in l_1 \wedge \exists y \in l_1, y \notin l_2 => R_{ij} = 3$$

- **Overlap**

$$\exists x_1, y_1 \in l_1, \exists x_2, y_2 \in l_2 \wedge x_1 \in l_2 \wedge y_1 \notin l_2 \wedge x_2 \in l_1 \wedge y_2 \notin l_2 => R_{ij} = 6$$

In those equations, l_1 and l_2 are two attributes from rule 1 and rule 2 respectively.

5.2 Adaptation to the Semgrep syntax

The comparison algorithm was first designed for a completely different syntax than the Semgrep syntax. Therefore, we had to make some adaptations to the parser and the comparison function to be able to fully use the principles of the algorithm under the Semgrep syntax.

The first issue was the syntax, which was significantly different. The code was designed for energy-consumption-type syntax, while we use a YAML-type syntax in Semgrep. Since the Semgrep syntax has a lot of different attributes, 18 to be exact, we had to simplify the problem. We notice that some attributes were not essential to the proper functioning of the algorithm. We could either not take some attributes into account (fix, path, severity, message) or simplify them (pattern-either, pattern-regex, pattern-inside, metavariable-regex, metavariable-pattern, metavariable-comparison, pattern-not-inside, pattern-not-regex). Some of them are just rarely used in our case, so we decided to drop them since implementing them would be useless for us. This includes option, fix, metadata, and the path rule. Note that fix, severity, and message are attributes that do not change the purpose of a rule but are there for complementary information. Two rules could have the same meaning but with a different message, fix, or severity message, depending on who would have written the rule.

For the simplification of the different rules, we finally based ourselves on the id, pattern, pattern-not, and metavariable attributes. We decided to treat the pattern-either attribute as a list of different pattern rules, and to include them with the actual different pattern since at the end of the day it is the same thing. We also implemented our syntax to deal with regular expressions. We decided that any rule based on a regular expression should be flagged with an "r:" at the beginning of the rule. In this way, we know that we have to treat it as a regular expression later in the comparison process. Still in the parser, there is one case where we decide to change a normal expression into a regular expression. This

is whenever a "..." (which means "anything") is used in a rule. If the "..." syntax is used, we escape all the characters and modify it by the corresponding regular expression, which is .*, and then we add the flag "r:" at the start of the rule.

For instance, if we take a part of the "odoo-request-api-secured" rule, we have :

```
- pattern-inside: |
    $HEADERS=$DICO
    ...
    requests.get(...,headers=$HEADERS,...)
```

This rule will be transformed into a regular expression. To simplify, let's say that none of the $ values have corresponding metavariables. The first step is then to escape special characters. The rule becomes then :

```
\$HEADERS\=\$DICO\.\.\.requests\.get\(\.\.\.\,headers\=\$HEADERS
\,\.\.\.\)
```

Once it is done, we will be looking for "..." characters. These characters are now escaped so we are looking for their corresponding match(\.\.\.) and we replace them with their regular expression value(.*). And finally, we add the "r:" flag to say that it is now a regular expression. The rule will then be looking like this:

```
r:\$HEADERS\=\$DICO.*requests\.get\(.*,headers\=\$HEADERS\,.*)
```

Finally, the last thing we do in the parser is whenever there are one or more values in the metavariable attribute, we take this metavariable and replace all the occurrences in the other attributes. This simplifies a lot the process and the rules we have to take into account. A tricky corner case we had is whenever two metavariables begin with the same letters and one name is shorter than the other (for instance $VAR and $VARIABLE). If $VAR comes before $VARIABLE in the metavariable field, since we do a regex search and replace every $VARIABLE would be replaced by the value of $VAR + IABLE. This is annoying and we wanted to avoid that. The easiest solution we found was to first sort the different metavariables by length and then search and replace all the occurrences starting with the metavariable with the longer name. Let's continue with our Odoo example: in our case, we have 2 potential metavariables, $DICO and $HEADERS. Here only $DICO has been defined in the rule as a metavariable, so we will only be replacing this value. Let's say the metavariable has been defined as :

```
  - metavariable-pattern:
              metavariable: $DICO
              patterns:
                - pattern-regex: \bdico\b|\bdictionnaire\b
```

Then, we have to replace every metavariable dico in our previous regular expression with the right regular expression. Luckily, in our case, it is already a regular expression since we have a "pattern-regex". So we obtain as a final regular expression:

```
r:$HEADERS=(\bdico\b|\bdictionnaire\b).requests.get(.,
           headers=$HEADERS,.*)
```

Once we have correctly retrieved the simplified rule set from the CSV file, the comparison algorithm will take place. Once again, since the algorithm was designed for intervals containing integers and we have our Semgrep rule, the comparison function has to be changed.

All these modifications also helped to simplify another big problem, which was the depth of the rule. By replacing every metavariable immediately with a regular expression, we partially solved an issue. We also solved the same depth issue by simplifying all the different Semgrep syntax rules into 3 rules (pattern, pattern-not, and metavariable) combined to write all these combined and simplified rules into one simple array.
For our CSV, we also decided to change the syntax a bit. We kept the "," as a separator for different rules, but we decided to use the delimiter "[]" for an array. Inside an array, we delimit two elements by a ";", to make it more simple to understand for us in longer rule. For instance, if we want to create a rule that contains two different patterns, let's say pattern 1 is "a" and pattern 2 is "b", then inside the CSV it will look like "[a;b]" in the corresponding pattern column.

For the comparison function, we start by checking whether one (or both) of the rules is a regular expression. This is important to know whether a rule is already written in regular expression or not, because without that we cannot know if we have to escape special character or not. We have to escape them to transform them into regular expressions anyway because we use the "re" Python package, which only deals with regular expressions. Whenever a variable is still in the regular expression, instead of escaping it, which wouldn't make sense, we modify it by its equivalent regular expression, ".*", since if this value is still there that means that there is no matching metavariable and the value is actually 'anything'.

Once everything is translated into regular expressions and everything has been taken care of, we can finally sort the different comparisons into an equality, an overlap, or a difference, and let the algorithm do its job. To give the proper IDC value we compare every item of the rule 1 selected attribute with every item of this attribute from rule 2. And the operation is done reciprocally by comparing afterward the rule 2 attribute with the rule 1 attribute. By comparing the expressions one by one it is simple to check whether both attributes are equal, included,

overlapping, or completely different.

Finally, we decided that the GUI (graphical user interface) was no longer adapted to our needs. Therefore, we modified the "check" button, so that instead of comparing one rule at a time, we can compare all of them at the same time and get a popup with the corresponding double-entry matrix.

5.3 Guide to the algorithm

Here is a guide to properly use our algorithm. Unfortunately, it is not possible yet to upload directly from Semgrep all the rules. Therefore, to use the algorithm, we have to write all our rules by hand in a CSV file. The right way to write the rules into the CSV has been explained in the previous section. Once we are done writing our rules in our CSV file, we can run the "gui.py" script. Running this script will give us this interface :

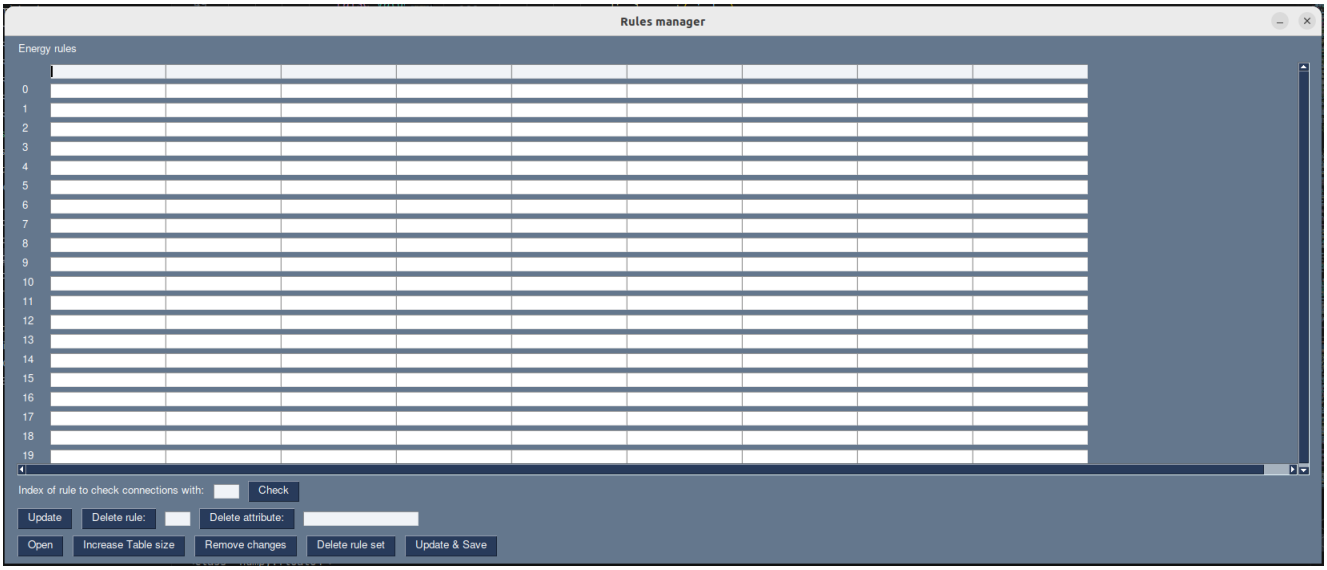

Figure 5.1: Graphical User Interface (GUI)

Note that running the script might involve installing new packages. Once we get to this GUI, we only need to perform two more steps. The first one is to import our handwritten rule set by clicking the "open" button. Once this is done, we will have to select it in our file. Once this is done, the last step is to click on the check button and a popup with all the comparisons will appear. Here is an example with a subset that we created to demonstrate :
We can see that we have imported a small set of invented rules for this demonstration.

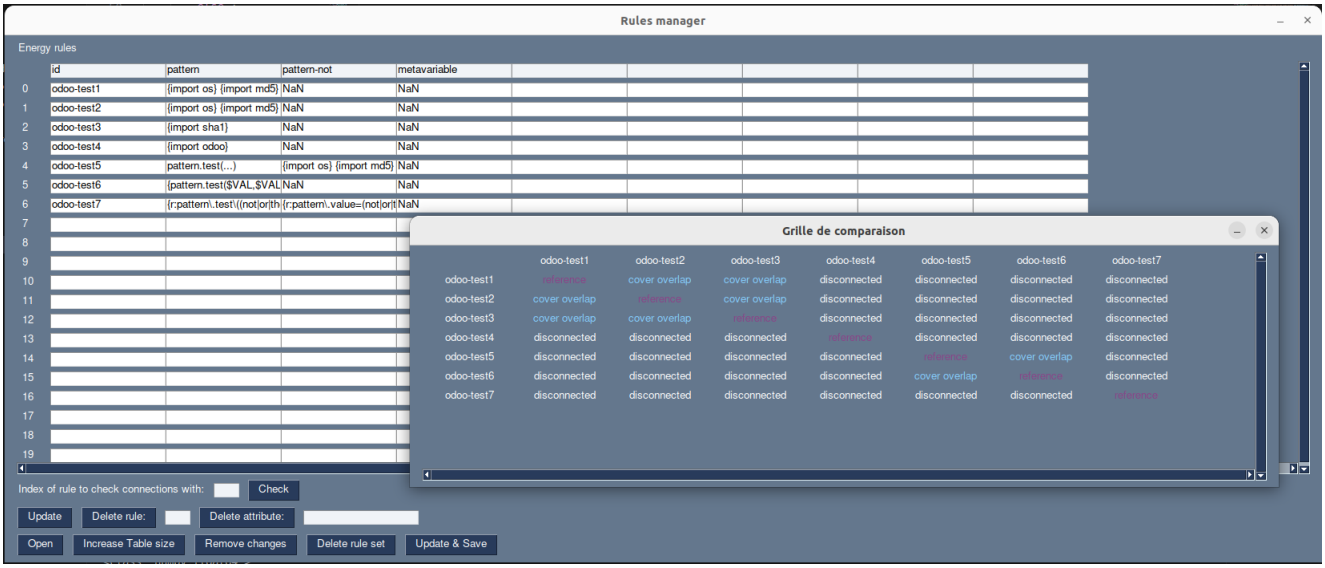

Figure 5.2: Subset comparison

In this example, we can see in the popup's diagonal a purple "reference", which makes sense. Reference means that we are comparing a rule with itself. In the case of a double-entry matrix, this is on the diagonal. Then, we can see some light blue cover overlaps between rules. This is because we inserted some parts of rules intentionally multiple times in different rule IDs. We can also see some white "disconnected" between rules, which means that there is no existing link between these rules. We can only read the upper right of the matrix since it is a symmetric matrix. This GUI is really easy to use and intuitive. In case we need it, there are also some other functionalities such as "delete attribute", "delete rule" or "delete rule set".

5.4 Results

It is now time to use the algorithm to check our rules written in Semgrep to detect any duplicates, redundancies, or contradictions. To do this, we have chosen to write about twenty rules in a CSV file that will be interpreted by the algorithm. We only included 20 rules out of 30 because the others are trivial for the algorithm in the sense that they only flag short patterns that are very different from the patterns found in other rules. Therefore, it is not relevant to include them in the algorithm as it would require more resources for the algorithm to process them, and it would also add unnecessary information for us, the users of the tool. The rules are manually written in the CSV file with a slightly different syntax than

what can be found in Semgrep, as these changes are necessary to match certain syntax choices made for the proper functioning of the algorithm. However, the rules retain their meaning, which is the most important thing. On the image [5.3], we can see the final list of rules used for the algorithm (the image can be found in the appendix more readable): As explained earlier, we use the "check" button to

	id	pattern	pattern-not	metavariable
0	odoo-safe_eval	eval(...) safe_eval(...) {$OBJECT.safe	{$OBJECT.safe_eval(...,self._get_eva	NaN
1	odoo-unstrusted-console-output	{r:\$CONNECTION\.(error|log|debug|	NaN	NaN
2	odoo-qweb-eval	r:_?qcontext r:_?eval_context	{r:def\s[a-zA-Z0-9_]*_qcontext} {r:de	NaN
3	odoo-handle-requests-separately-fror	{r:\bfrom\s+requests\b.*\bimport\b|\br	{r:\brequests\.exceptions}	NaN
4	odoo-sqlalchemy-execute-raw-query	{$CONNECTION.execute($SQL+$VA	{$CONNECTION.execute($SQL+self._	NaN
5	odoo-cr_execute	{$CONNECTION.execute($SQL+$VA	NaN	NaN
6	odoo-avoid-hardcoded-config-debug	{$M.update(DEBUG=True)} {$M.upda	NaN	NaN
7	odoo-avoid-hardcoded-config-env	{$M.update(ENV='=~/^development|p	NaN	NaN
8	odoo-avoid-hardcoded-config-secret	{$M.update(SECRET_KEY='=~/.*/')}	NaN	NaN
9	odoo-avoid-hardcoded-config-testing	{$M.update(TESTING=True)} {$M.up	NaN	NaN
10	odoo-crypto	md5 sha1 hmac hashlib cryptography	NaN	NaN
11	odoo-database-secret	{$SECRET=$SUDO.get_param('datat	{def$FUNCTION(...):\n...\n$SECRET	NaN
12	odoodisabling-sqli-linter	{re:pylint\s*:\s*disable.*=sql-injection	NaN	NaN
13	odoo-external-control-of-file-name-pa	{open($INPUT,...)}	{re:open\(^[a-zA-Z0-9_\-\.']+$,.*\)}	NaN
14	odoo-false-sanitize-attributes	{re:\bsanitizes*=\s*False} {re:\bsaniti	NaN	NaN
15	odoo-jwt-hardcoded	importjwt {fromjwtimport$ANYTHING}	NaN	NaN
16	odoo-kn64kbnw	setattr __getattribute__ attrgetter reso	{$ITERABLE=[...]\n...\nfor$VARin$IT	NaN
17	odoo-markup	Markup(...)	{frommarkupsafeimport$O\n...}	NaN
18	odoo-sensitive-information-post	{r:\$REQUEST\.post\(\.\.\.,data=(pass	NaN	NaN
19				

Figure 5.3: Actual rules set tested in the algorithm

Figure 5.4: Result of the algorithm

launch the algorithm and display in a two-dimensional table all the relationships between the rules.

Not surprisingly, the results show that the rules are quite independent of each other, with a majority of disjunctions. However, we can still observe an overlap between two rules: "cr_execute" and "sql_alchemy" against SQL injections. By

looking at the table the change of color. Here is the result obtained with the Python script to demonstrate how simple it is to detect an anomaly between the rules :

When looking at the rules, we notice that they are indeed similar:

```
  rules:
- id: odoo-cr_execute
  metadata:
    OWASP: A03 INJECTION
    CWE: CWE-89 sql injection
  patterns:
    - pattern: |
          cr.execute( $SQL.format($VAL), ... )
      - patterns:
          - pattern-inside: |
              $QUERY = $SQL + $VAL
              ...
          - pattern: |
              $CONNECTION.execute($QUERY, ...)
      - patterns:
          - pattern-inside: |
              $QUERY = $SQL % $VAL
              ...
          - pattern: |
              $CONNECTION.execute($QUERY, ...)
      - patterns:
          - pattern-inside: |
              $QUERY = $SQL.format($VAL)
              ...
          - pattern: |
              $CONNECTION.execute($QUERY, ...)
      - patterns:
          - pattern-inside: |
              $QUERY = f"...{...}..."
              ...
          - pattern: |
              $CONNECTION.execute($QUERY, ...)
  message: EXECUTE IS SENSITIVE TO SQL INJECTION
  WHEN USED WITH INPUTS
  languages:
    - python
  severity: WARNING
```

```
  rules:
- id: odoo-sqlalchemy-execute-raw-query
    languages:
    - python
  patterns:
    - pattern-either:
        - pattern: |
            $CONNECTION.execute( $SQL + $VAL , ... )
        - pattern: |
            $CONNECTION.execute( $SQL % $VAL , ...)
        - pattern: |
            $CONNECTION.execute( $SQL.format($VAL), ... )
        - patterns:
            - pattern-inside: |
                $QUERY = $SQL + $VAL
                ...
            - pattern: |
                $CONNECTION.execute($QUERY, ...)
        - patterns:
            - pattern-inside: |
                $QUERY = $SQL % $VAL
                ...
            - pattern: |
                $CONNECTION.execute($QUERY, ...)
        - patterns:
            - pattern-inside: |
                $QUERY = $SQL.format($VAL)
                ...
            - pattern: |
                $CONNECTION.execute($QUERY, ...)
        - patterns:
            - pattern-inside: |
                $QUERY = f"...{...}..."
                ...
            - pattern: |
                $CONNECTION.execute($QUERY, ...)
    - metavariable-pattern:
        metavariable: $VAL
        patterns:
          - pattern-not: self._tables
```

It will be necessary to modify one of the rules or even merge or separate them to avoid matching the same parts of files multiple times. Due to the similarities

between both rules, we can merge cr.execute and sqlalchemy.

Furthermore, these results provide certainty about the effectiveness and advantage of this tool, especially as the rule base expands and redundancies become more frequent.

Chapter 6

Future work

Future work in this research encompasses several areas that can further enhance the effectiveness and applicability of the proposed methodologies and tools. Firstly, an important aspect to consider is adding and sorting HTML and shell rules within Semgrep for Odoo. Even if these languages represent only 0.2% of the code, it is not excluded that they contain security flaws. In general, it would be very interesting to add rules in Semgrep regarding SCSS and CSS files. There are no rules for these extensions, and yet these are useful languages. Semgrep has a very wide rule set concerning security flaws and vulnerabilities in Python and JavaScript, but we believe that adding the same kind of rule set for other well-established languages could benefit many people. Furthermore, working on rules such as the sudo rule or the cryptography rule should be undertaken. In fact, these rules are important since they point out some critical aspects of the code that could let introduce vulnerabilities. Unfortunately, it would take time to focus on the different cases that allow these dangerous rules to be flagged because it is mostly studying sudo calls on a case-by-case basis and studying the sub-methods of cryptography that can be considered as safe.

While adding more rules to the open-source project is very important, another important aspect of this tool that could benefit many people is reducing false positives. Even if we already adapt Odoo's specific rules, we believe that with such a well-developed tool as Semgrep, it is possible to reduce even more false positives and customize the rules. This adaptation of rules can be considered for Odoo's specific rules as well as the more general Semgrep rules.

Finally, a last step in the right direction regarding this thesis and the rule sorting algorithm we have developed[29] would be to adapt the parser. We start from a CSV file where the user has to manually convert each rule they want to compare. With larger rule set like Semgrep, this is more difficult to convert

every single rule and it would take too much time. Therefore modifying the parser to accept the YAML syntax and retrieve every rule directly from Semgrep would be a step in the right direction and very helpful for the users of our algorithm.

Additionally, there is a need to enhance the integration between the algorithm and Semgrep. Currently, the process involves converting Semgrep rules to a CSV format and then applying the algorithm separately. A future direction would involve developing a direct integration where Semgrep rules can be uploaded and parsed within the algorithm, eliminating the intermediate steps and enhancing the overall efficiency of the analysis.

Conclusion

This thesis is useful to the field of software security, with a specific focus on the Odoo company. In this thesis, we have presented the Odoo enterprise and how they contribute to the world of ERP and CRM. Then we introduced the methodology that we followed to provide work as qualitative as possible. It began with a risk analysis of the company and the software they provide. We identified the different aspects of the security risk that an employee of Odoo can encounter but also one of their clients. To realize this analysis we relied on the CIA triad. The next part was the main focus of this thesis. In this part of the thesis, we adapted the rules already designed by Odoo to the syntax of Semgrep. With this tool, we have improved the rule set for Odoo to be more relevant and efficient for vulnerability detection. Lastly, we adapted an algorithm designed by a former student to detect redundancies within a rule set.

The risk analysis in Chapter 3 develops the aspects of the three different branches of the CIA triad which are the confidentiality, integrity, and availability of Odoo. We based our research on multiple sources of information, such as discussions with the Odoo security team, their webpage, their code base, and other information we could find online. We discuss all the methods implemented to mitigate the risks in the three different domains but also discuss what, for us could be improved. We have identified the remaining risky entry point for an attacker that could compromise one of the different areas of the triad. In the confidentiality section, we discussed the cryptographic algorithm used to protect sensitive data with a discussion about the PBKDF2 algorithm. We also discussed the user's access and the different privileges that employees, users, clients, and support clients could have on the application. Additionally, we discussed how the logs are used and made to identify the different connections and actions made on the application. In the integrity section, we discussed the critical assets of Odoo, the different threat actors, the security of their connections, their recovery point objective in case of faulty hardware, access to their database, and the potential impact on Odoo in case of a security flaw. In the availability section, we mainly discussed Odoo's policy about data replication, their recovery time objective, and their policy about DDoS attacks.

The Odoo security team gave us a file containing different regular expressions they use for preventing security leaks by checking every push committed by their developers in their main Github branch. We started with this file and converted it to run on a static analysis open-source tool, Semgrep, in Chapter 4. With the pre-defined rule set from Semgrep combined with the rule set from Odoo, we encountered a significant number of alerts triggered on the Odoo code base. To address the issue of false positives, we have studied the different rules together with the code and we have improved this rule set by modifying some of their rules. We also sorted and selected rules from the Semgrep rule set. We selected rules that were relevant and fitting to the context of the Odoo application. Furthermore, we considered the OWASP Top 10 and the various Common Weakness Enumerations (CWEs) associated with this ranking. Based on this, we sorted the rules and created additional ones to align as closely as possible with the specifications of the top 5 vulnerabilities. As a result of this work, we achieved a **30%** reduction in false positive detections in Odoo's rule set. We also improved the performance of the analysis tool Semgrep by reducing the number of rules tested on the extensive code base of Odoo.

Finally, in Chapter 5, we modified an algorithm developed by M. Legast to fit our needs. The algorithm was initially developed to search for overlapping or equal intervals. We adapted it to detect duplicate or overlapping rules. For this purpose, we had to modify some of the key functions, such as the parser and the comparison function, to fit in the algorithm. Additionally, we made modifications to the graphical user interface, as it was no longer suitable for our requirements. We tested the modified algorithm on our small set of rules, and we successfully identified one overlapping rule. This confirms the efficiency of the algorithm. We believe that our work on this algorithm can be beneficial to others in the future, as we are aware that a rule set can quickly become extensive, and take time to execute. By identifying duplicates, our work helps reduce the number of rules and, consequently, the time required for a rule set to complete its execution. Our work is freely accessible on gihtub[29].

Our work developed in this thesis proved that even if the Odoo company has good developers and a good security policy, they are not immune to security threats. We brought a solution that allows the company to maintain a good overview of the different security risks they can find inside their code base. Those alerts can help the company to either emphasize good practices but also prevent vulnerabilities. Through our work on reducing consequently the number of false positives, they can now have an easier overview of the different parts of the code that might be vulnerable. Furthermore, the platform made by Semgrep is very user-friendly and

open-source. The algorithm will be useful combined with the rule set provided but also for the future rules that Odoo will add. Indeed it is important to avoid duplicate rules and this algorithm is perfect for that. It will allow Odoo to be as efficient as possible by helping them avoid creating any duplicates or overlapping rules.

There are certain aspects of our thesis that could be enhanced in future research and can be pursued as a continuation of this thesis. This includes refining complex rules that require extensive analysis and exploring languages useful for Odoo beyond Python and JavaScript, such as HTML. Additionally, further progress can be made on the algorithm by developing a tool that directly converts Semgrep rules into a CSV file, making them readily available for the algorithm.

Odoo approached us for assistance in obtaining a more relevant rule set to enhance their risk management efficiency. This is precisely what we have explained and provided throughout this thesis. Therefore, we can conclude that both the work on the rules and the algorithm will be beneficial to the Odoo company. Furthermore, this work has the potential to be applicable to numerous other enterprises or individuals, as security rules often rely on a general ranking of vulnerabilities.

Bibliography

[1] Ivo Gomes, Pedro Morgado, Tiago Gomes, Rodrigo Moreira. An overview on the Static Code Analysis approach in Software Development.

[2] FRANTZ, Miles, XIAO, Ya, PIAS, Tanmoy Sarkar, et al. POSTER: Precise Detection of Unprecedented Python Cryptographic Misuses Using On-Demand Analysis. In : The Network and Distributed System Security (NDSS) Symposium. 2022.

[3] "Odoo wikipedia", [Online accessed 30 May 2023], https://fr.wikipedia.org/wiki/Odoo

[4] "Odoo codebase", [Online accessed 30 May 2023], https://github.com/odoo/odoo

[5] "Odoo website", [Online accessed 30 May 2023], https://odoo.com

[6] "Odoo documentation", [Online accessed 30 May 2023], https://www.odoo.com/documentation/16.0/developer/reference/backend/orm.html

[7] "Semgrep, [Online accessed 30 May 2023], https://semgrep.dev/

[8] "Pbkdf2", [Online accessed 30 May 2023], https://www.researchgate.net/figure/SHA-512-based-PBKDF2-scheme_fig1_265793318

[9] LEGAST, Magali et LEGAY, Axel. Assistance in the management of rule sets for rule-based expert systems.

[10] "odoo tutorial", [Online accessed 30 May 2023], https://www.odoo.com/documentation/16.0/developer/tutorials/getting_started.html

[11] "QWeb", [Online accessed 30 May 2023], https://www.odoo.com/documentation/16.0/fr/developer/reference/frontend/qweb.html

[12] "Odoo security", [Online accessed 30 May 2023], https://www.odoo.com/fr_FR/security#part_16

[13] "Semgrep tutorial", [Online accessed 30 May 2023], https://semgrep.dev/learn/

[14] "Regular expression", [Online accessed 30 May 2023], https://regexone.com/

[15] Legast, Magali. Rule-based expert system for energy optimization : detection and identification of relationships between rules in knowledge base. Ecole polytechnique de Louvain, Université catholique de Louvain, 2021. Prom. : Legay, Axel. http://hdl.handle.net/2078.1/thesis:30588

[16] "CodeQL", [Online accessed 30 May 2023], https://codeql.github.com/

[17] "OWASP top 10", [Online accessed 30 May 2023], https://owasp.org/Top10/fr/

[18] "WeSmart", [Online accessed 30 May 2023], https://wesmart.com/

[19] "Open SSL", [Online accessed 30 May 2023], https://www.ssllabs.com/

[20] "SQLMap", [Online accessed 30 May 2023], https://sqlmap.org/

[21] "PyLint", [Online accessed 30 May 2023], https://pypi.org/project/pylint/

[22] "Bandit", [Online accessed 30 May 2023], https://www.overleaf.com/project/6419c48aff1ca6fd718d4978

[23] "ESLint", [Online accessed 30 May 2023], https://eslint.org/

[24] "Semgrep performances", [Online accessed 30 May 2023], https://semgrep.dev/blog/2022/static-analysis-speed

[25] "CWE", [Online accessed 30 May 2023], https://cwe.mitre.org/

[26] "RuleSet in Semgrep", [Online accessed 30 May 2023], https://semgrep.dev/docs/faq/#how-are-semgrep-and-its-rules-licensed

[27] "Relationsip identification tool", [Online accessed 30 May 2023], https://gitlab.com/legastm/Relationship-Identification-Tool

[28] "Github repository with rule set", [Online accessed 30 May 2023], https://github.com/RomTourpe/Semgrep-rules

[29] "Github repository RIT", [Online accessed 30 May 2023], https://gitlab.com/RomTourpe/Relationship-Identification-Tool

[30] "ISO 27001 certification ", [Online accessed 30 May 2023], https://www.iso.org/fr/standard/27001

[31] "PCI certification", [Online accessed 30 May 2023], https://www.pcisecuritystandards.org/

[32] "Soc 2 comparison", [Online accessed 30 May 2023], https://www.itgovernance.eu/blog/en/iso-27001-vs-soc-2-certification-whats-the-difference,

[33] "Soc Type comparison", [Online accessed 30 May 2023], https://www.ovhcloud.com/fr/enterprise/certification-conformity/soc-1-2-3/

Appendix A

Comparison algorithm

Figure A.1: Result of the algorithm for 20 of Odoo's modified security rules

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**