FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

An IDE Plug-in to detect security vulnerabilities in Infrastructure-as-Code Scripts

Tiago José Antunes Ribeiro

DISSERTAÇÃO DE MESTRADO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Filipe Lima Maranhão de Abreu

July 21, 2021

An IDE Plug-in to detect security vulnerabilities in Infrastructure-as-Code Scripts

Tiago José Antunes Ribeiro

Mestrado Integrado em Engenharia Informática e Computação

July 21, 2021

Abstract

Cloud computing has made it possible for more individuals and corporations than ever to access complex IT infrastructures. Although accessing them became easier, properly installing and configuring them still remained a challenge, as this was still a long manual process that needed to be repeated across all systems. Infrastructure-as-Code promises to change all of that, by automatizing system installation and configuration. It allows the formalization of the configuration process in sequences of lines of code that can then be executed multiple times in different places to ensure consistency. This also means that any security vulnerability present in that same code will be replicated across all infrastructure, revealing the importance of tools that are able to detect these.

For the *Puppet* language, only the SLIC tool exists. Although having promising results in initial tests, further ones revealed several issues related to a low precision in detecting vulnerabilities in code. This motivated the development of a new and improved tool, capable of addressing these issues with high levels of precision and providing developers, in a convenient manner, the assurance that their code is secure and ready for deployment.

The developed tool integrates seamlessly with the average developer workflow, providing warnings to vulnerabilities in real time, right when they are written. Its architecture allows not only this integration with virtually every IDE in the market, but it also provides the flexibility and modularity to support the future improvement and integration with even more precise and complete analysis rules. This seamless user experience was approved by the participants of the conducted User Study.

Keywords: Infrastructure-as-code, Linter, Cybersecurity, Puppet, User Experience, Plugin

Resumo

A Computação em Nuvem tornou possível o acesso a infraestruturas informáticas complexas por parte de muitos mais indivíduos e organizações. Embora o acesso tenha ficado mais fácil, o processo de instalação e configuração permaneceu um desafio, visto continuar a ser um processo longo e manual que precisava de ser repetido por todos os sistemas. A "Infrastructure-as-Code" (Infraestrutura como código) promete mudar tudo, automatizando a instalação e configuração de sistemas. Permite a formalização do processo de configuração em sequências de linhas de código que podem depois ser executadas múltiplas vezes em diversos locais, de modo a garantir consistência. Isto também significa que qualquer vulnerabilidade de segurança presente no código irá ser replicada ao longo de toda a infraestrutura, realçando a importância da existência de ferramentas que as possam detetar.

Para a linguagem *Puppet*, apenas a ferramenta SLIC existe. Embora esta apresentasse resultados promissores nos testes iniciais, após alguma investigação foram revelados diversos problemas relacionados com a baixa precisão da ferramenta em detetar vulnerabilidades no código. Isto motivou o desenvolvimento de uma nova e melhorada ferramenta capaz de corrigir estes problemas, com altos níveis de precisão e dando a garantia aos programadores, de uma forma conveniente, de que o seu código é seguro e está pronto a ser lançado.

A ferramenta desenvolvida integra-se perfeitamente com a rotina de trabalho do desenvolvedor comum, fornecendo avisos às vulnerabilidades em tempo real, no exato momento em que estas são escritas. A sua arquitetura permite não só esta integração com virtualmente qualquer IDE no mercado, mas também a flexibilidade e modularidade necessárias para suportar futuras melhorias e integração com regras de análise ainda mais precisas e completas. Esta experiência de utilização completamente integrada foi aprovada pelos participantes do estudo de utilização realizado.

Keywords: Infrastructure-as-code, Linter, Cybersecurity, Puppet, User Experience, Plugin

Acknowledgements

I would like to start by thanking Professor Rui Maranhão for orienting my Master Thesis. He, together with Sofia Reis, were a very valuable help during the development and writing of this project. Their continuous support and also previous experience and research made it possible for me to build this program.

I would also like to thank everyone who volunteered to participate on my User Study. Their availability to sit down during an hour to have a call with me was critical to ensure that my work could be properly evaluated by potential users. Their feedback will definitely prove very valuable for the future continuity of the project.

Finally, a special thanks to my family and to my girlfriend for their continuous support during this long journey and for never letting me give4 up.

Tiago Ribeiro

"You can never protect yourself 100%.
What you do is protect your self as much as possible
and mitigate risk to an acceptable degree.
You can never remove all risk."

Kevin Mitnick

Contents

List of Figures

List of Tables

Abbreviations

IT	Information technology
IaC	Infrastructure-as-Code
SLIC	Security Linter for Infrastructure as Code scripts
IDE	Integrated Development Environment
AWS	Amazon Web Services
DevOps	Development Operations
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
TLS	Transport Layer Security
UML	Unified Modeling Language
LSP	Language Server Protocol
HTML	Hypertext Markup Language
JSON-RPC	JavaScript Object Notation - Remote Procedure Call
NLP	Natural Language Processing
VSCode	Visual Studio Code
CI/CD	Continuous Integration/Continuous Delivery
GUI	Graphical User Interface
GPG	GNU Privacy Guard
APT	Advanced Packaging Tool
SAT	Static Analysis Tool
CSS	Cascading Style Sheets

Chapter 1

Introduction

With the growth in recent years of cloud computing, more complex IT infrastructures became much more accessible to companies and individuals. Although these systems became more< affordable to obtain, proper installation and configuration of the software still remained a very difficult and arduous task. Infrastructure-as-code promises a solution for this problem, by allowing the formalization and automatization of these tasks. This saves greats amount of time and makes the replication of systems a very easy task.

However, having a sequence of lines of code that can easily setup and deploy a system and be replicated across several places also poses an essential security threat. These lines that are consistently executed in several places make it so that a security flaw is also replicated the exact same way everywhere. This makes the existence of tools that can detect security problems in IaC very crucial. Focusing on the *Puppet* language[1], the scenario is not very positive, with only a relatively recent attempt at making a tool like this. The SLIC tool, created by Rahman et al., approaches the problem by looking for some common security vulnerabilities in the code, known as *The Seven Sins* [13].

Although the author of the SLIC tool reported a precision[2] of around 99% in his original study, a later independent study showed a much worse scenario, with a precision as low as 8% [17]. This discovery highlights not only the weakness of the tool and the importance of independent evaluation of software, but also the urgent need of a precise security analysis tool for the *Puppet* language that will not scare developers away with a high amount of false positives[3].

With this in mind, the goal of this project is to build a new tool, based on SLIC, which is capable of analyzing *Puppet* code in real-time, capable of detecting security vulnerabilities and other problems with a much higher degree of precision. Integration with several IDEs is the key

[1]Puppet is a declarative language used to describe system configuration, being one of several IaC tools available in the market.

[2]The precision is calculated by the ratio of actual vulnerabilities detected and the total number of alerts given by the tool

[3]False positives are alerts raised by the tool that do not correspond to real security flaws

to ensure a good User Experience and a seamless integration into most development processes. The final goal of the tool consists in preventing security vulnerabilities from being introduced into systems early in the development process.

The list of contributions is:

- Development of a tool able to analyze *Puppet* code and detect, with great precision, vulnerable sections that might pose a security risk. This tool was developed based on a universal and expandable architecture able to adapt to new security vulnerability scenarios in the future.
- Development of a plugin to integrate the tool with an IDE, able to demonstrate the true capabilities offered to the end user. This plugin feeds the tool with the code being written on the IDE window and in real time display alerts and other useful information to the developer about the vulnerabilities in his code.
- An User Study focused on evaluating the user experience of the tool and how likely is for developers to start using it in the near future. The educational aspect of the tool is also a big focus of the study, as teaching developers on how to avoid these vulnerabilities in the first place is probably the most important contribute this tool can give to the community.

The document will start with a deeper analysis on what IaC is, how it works and its benefits for companies and IT professionals, alongside with some of the security dangers that it might introduce. This will be followed by an overview of the state of the art of Security Tools for IaC software, namely the SLIC tool for *Puppet*. A deeper analysis on how it works, its structure and logic will also be conducted, alongside its weaknesses and points for improvement. Taking on the lessons from this analysis, the developed tool is presented. First, by a deep analysis of its modular and expandable architecture, followed by the process used to implement this designed architecture into practice. The integration with IDE is also mentioned, as it constitutes one of the main differentiating points of the linter. Finally, the results of the User Study made to evaluate the user experience of this software are presented, with the lessons learned from it being an important contribute to the conclusion and future improvements of this Thesis.

Chapter 2

Infrastructure-as-Code scripts security analysis

2.1 Infrastructure-as-Code explained

Infrastructure-as-Code (IaC) consists in the provisioning, configuration and deployment of IT infrastructures using machine readable and reusable scripts, contrasting with the traditional interactive and manual process carried out by a human. Using a tool like *Puppet* for example, it is possible to write code that sets up a complete IT infrastructure without ever needing to manually install or configure software using interactive tools or even by physically configuring the hardware [8].

2.1.1 Infrastructure-as-code in action

Using a practical example, it is possible to imagine an online service, like a store that needs to be deployed into production. The company has already the hardware available, in the form of several servers commissioned from an Cloud Computing service and needs to configure them to run the already developed software[1]. This software consists in the website that is going to be available to customers and all the supporting applications that will allow staff to see, process and ship orders, while being able to properly account all transactions made. All of these applications will also need available storage for all the data generated. Because the company operates in several different countries around the globe, they commissioned redundant servers in several locations that need to be similarly configured.

Traditionally, this would be a very tedious process. A team of professionals would need to manually login into every server, install the operating system, make all network configurations, setup load balancers[2] according to the expected demand of customers in a particular area, install

[1]As a simplification, the hardware is commissioned from a Cloud Computing service like AWS or Azure, meaning that its maintenance and configuration is not handled by the store company.

[2]A load balancer allows a system to handle a large amount of concurrent users, by distributing them across several similar running servers that can satisfy their requests [14].

the website application in a server, alongside all the other supporting applications and many other tasks. The problem only gets worse when considering the fact that this process will need to be repeated across all regions where the company will operate, meaning a lot more time spent and the possibility of errors or inconsistencies in the configuration.

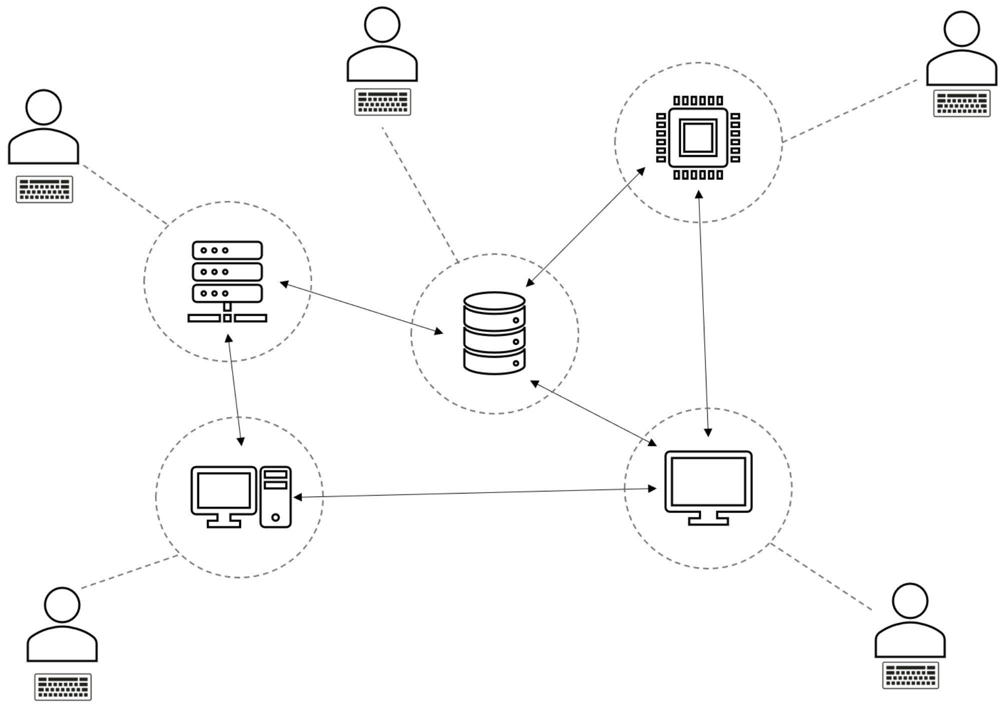


Figure 2.1: IT Infrastructure manual configuration without IaC

A much better alternative to this scenario would be if a team of developers wrote a script that described exactly how to configure the system: How many servers, the software needed in each server, network and any other configuration needed. This script could then be run on the environment and after some time, the entire infrastructure should be configured and the website up and running. When configuring other regions, the same script can be used, ensuring that all configurations will be identical.

2.1.2 Main advantages

As shown in the previous example, Infrastructure-as-Code has the potential to completely change the way IT infrastructures are configured. Suddenly, this process can also be included into the DevOps[3] cycle, meaning that software can not only be continuously versioned and tested, but also deployed.

Automating the infrastructure management process allows for rapid and consistent provisioning of systems. The code written can be documented and versioned, allowing for much greater

[3]DevOps is the set of practices aiming to merge software development with IT Operations, allowing for much faster development cycles by automating continuous delivery processes and incorporating many Agile philosophies.

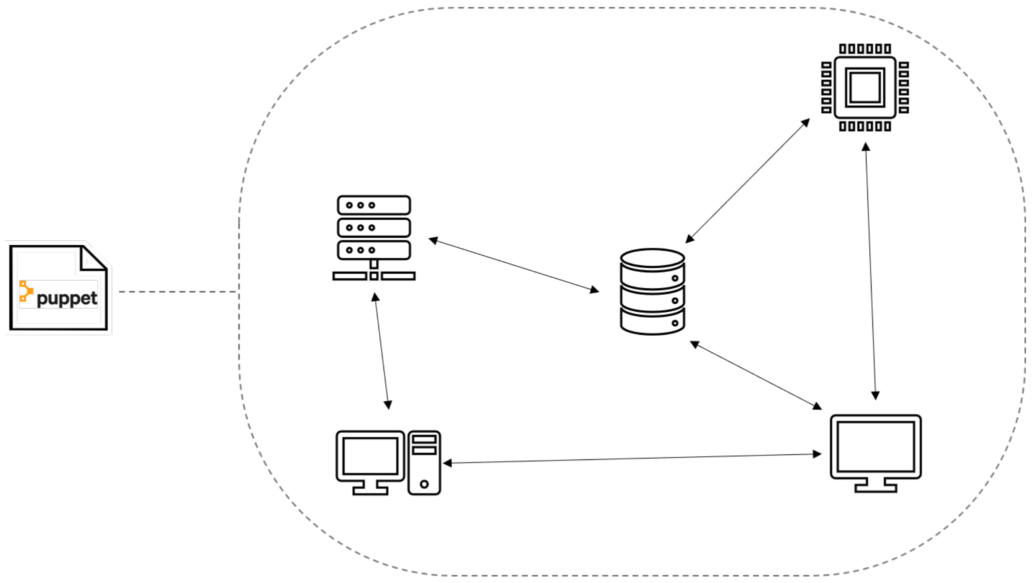


Figure 2.2: IT Infrastructure automatic configuration with IaC

transparency in tracking defects or changes. As with the software code itself, different parts of an infrastructure can easily be reused for other purposes, allowing for a much quicker process of assembling new environments [5].

Companies using IaC will definitely have an advantage, especially in a dynamic software environment, where requirements change and quick adaptation is vital. Software development following Agile methodologies no longer needs to be held back by the time it takes to deploy it. Instead, this deployment process can be brought into the Agile cycle and also be easily adapted whenever needed.

2.2 Security risks

Just like any other piece of code in an IT infrastructure, IaC is also vulnerable to security threats. Security is one of the main aspects of any IT system, especially if dealing with personal or confidential information that is highly desirable to a malicious entity. Security breaches are big business [10] and a secure infrastructure that is properly protected goes a long way to prevent these kind of attacks.

It is true that IaC can increase security by avoiding instances where vulnerabilities are introduced by human error during a manual configuration process, but this does not mean that those are not a possibility. Actually, because IaC ensures consistency across every instance by centralizing all configuration into a script file, any flaw in that same file will replicate itself across the entire infrastructure. This becomes an even worse problem considering the fact that this flawed configuration is now formalized into a code file, being in plain sight of a malicious attacker able to access its location. This raises the need to ensure the security of all IaC produced.

Although the security consequences of bad coding practices are a well-known phenomenon for several programming languages and tools, for IaC and specifically for the *Puppet* language, there is a huge lack of studies and knowledge [13]. This is a very worrying scenario because it makes it easy for developers to unknowingly produce vulnerable code without ever realizing its true consequences. For IaC to truly have a chance to become the new industry standard, good and safe code practices need to be studied and then made easily available to everyone who works in the area. These practices should then be enforced by trusted linters[4] and other security tools, ensuring the quality and safety of the code to developers and companies.

2.2.1 Practical example of a vulnerable IaC script

To illustrate how important is ensuring security on IaC scripts, an example will be used. In the following *Puppet* class, a new user will be created in the system, with a *username*, *password* and *home folder* defined.

```
class add_user {
  # add user, create homedir and set a password
  user { 'john':
    ensure     => 'present',
    comment    => 'Example user',
    managehome => true,
    password   => 'password123456',
  }
}
```

After this code is run, the system should have a new user named "john" with the password "password123456". Assuming a best case scenario, where the system properly encrypts passwords, it is easy to assume that no malicious attacker will ever get hold of the credentials of the account, even if it manages to gain access to the system. If this password is used to encrypt sensitive files, it is impossible for him to steal any important information.

This would be true in an infrastructure assembled in the traditional way, where a professional would pick a password that only he knew. Even if the password never gets changed, only this individual knows it and assuming that he does not write it down in any dangerous place, no one else will ever know. The problem here is that the password is defined not by a professional at the moment of setup, but it is already written down in the script. After the system is set up, the script code will still continue to exist, meaning that the password will be there in plain sight to anyone who gains access to the file.

It is easy to imagine a scenario where this file is kept in a code repository. This location could then be compromised by a malicious attacker, who is able to gain access to all files. By opening

[4]A linter is a software tool able to analyze static code in search of problems like errors, bugs or lack of compliance with a convention.

this one, he can easily understand that the password for that user is the one written down on the code. With this information, he is able to finish the attack and gain access to the entire system.

In this particular example, the best solution is to invest in a password management software. All passwords of the system would be stored in a secure location and the IaC scripts would simply query it for the values. No password would be left exposed in any piece of code.

A security analysis tool should be able to identify this piece of code as a vulnerability and warn the developer immediately, informing him of the risks he is being subjected to and the potential ways this could be fixed. This way, vulnerabilities like this one are avoided right at development time.

2.3 SLIC: A security analysis tool for Puppet

During the development of IaC scripts, developers can introduce security smells into their code. These are code fragments that are proven to be security weaknesses. They are weak points in the software that can potentially be exploited by a malicious attacker. In the previous IaC security vulnerability example at 2.2.1 it was possible to see that the presence of an hard-coded[5] password in the source code opened a door for a malicious attacker to entry in the system. This is a clear example of a code smell. *Rahman et al.* in his study of thousands of *Puppet* scripts [13] was able to find more code smells like this. He listed them as *The Seven Sins*. This knowledge was then used to build an automated code analysis tool named SLIC.

2.3.1 The Seven Sins

Rahman et al. worked on finding and categorizing types of code smells that could be present in *Puppet* scripts. For this, he gathered thousands of scripts from several different projects and after a process of analyzing the source code, he came up with a list organized by type, which he named *The Seven Sins* [13]. This list is composed by the following code smell types:

1. **Admin by Default** - An user is given administrator privileges by default, making it easier for accounts to have more authorizations than what they need, violating the *principle of least privilege*[6].

```
        $username = 'admin'
```

2. **Empty Password** - An account with an empty password makes it very easy for attackers to guess its value during an attack.

[5]Hard-coding is the practice in software development of embedding data directly into the source code, meaning that the only way of editing it is by modifying and recompiling the source code. It also means that it is visible to anyone with access to the source files.

[6]The principle of least privilege is the concept of giving users only the strictly necessary authorizations in the system for the tasks they need to perform. This grants a greater protection against malware, confidential data loss and other kinds of attacks by limiting what an attacker could do if he gained access to the user's credentials [12].

```
        $password = ''
```

3. **Hard-coded secret** - As shown in a previous example 2.2.1, having secrets written on the source code itself can open the door for attacks as password management becomes nearly impossible (recompilation of source code is needed for any change of credentials) and someone with access to the code repository could easily get all the passwords to attack the system.

```
        $password = 'securepassword1234'
```

4. **Invalid IP address binding** - Using the 0.0.0.0 IP address can cause security issues as this address allows connections from any other address in the network.

```
        listen_ip  => '0.0.0.0',
```

5. **Suspicious Comment** - A comment in the source code indicating a problem or a feature that was not done can be a rather obvious indication to an attacker that the system is not finished or possibly not fully protected in that specific area.

```
        # TODO: secure this password
```

6. **Use of HTTP without TLS** - HTTP connections without TLS are not encrypted at all, meaning that they are vulnerable to *Man-in-the-middle*[7] attacks.

```
            update_server          => 'http://company.com/updates/latest',
```

7. **Use of weak cryptography algorithms** - Security flaws can be found on cryptography algorithms, rendering them vulnerable to malicious attacks. Using vulnerable algorithms (*Rahman et al.* gives MD5 and SHA-1 as examples) to encrypt sensitive information will not ensure their confidentiality.

```
            $encrypted_pass = md5($password)
```

[7] In a Man-In-The-Middle attack, the communication between two endpoints is compromised by an attacker, who is able to intercept and even modify the traffic without anyone being aware of it [4].

These are the types of code smells found in the study and later used as a basis to build the tool. This does not exclude the existence of more types of code smells in IaC scripts that can open the door for security attacks.

2.3.2 Tool analysis

After organizing the list of different types of security smells, *Rahman et al.* realized that the only way to ensure that developers would stop committing them was through the development of a linter. So he presented the *Security Linter for Infrastructure as Code Scripts*. This tool promised the ability to scan through *Puppet* code and automatically detect the presence of any of the studied code smell types. The tool itself was developed using *Python* and from studying the source code[8] and the document itself, it is possible to conclude that its architecture is based off two main modules:

- **Parser** - This module reads through all *Puppet* code and each element detected is converted to a standard notation known as *Token*. Tokens are presented in the following format:

```
    <TYPE, NAME, CONFIGURATION VALUE>
```

  Some examples of tokens for different types of elements present in *Puppet* scripts are presented in table 2.1.

  Table 2.1: Examples of tokens generated by the SLIC Parser Module

Puppet code	Respective token
$username = "John Doe"	<VARIABLE, 'username', 'John Doe'>
user => 'John'	<ATTRIBUTE, 'user', 'John'>
#This function adds an user	<COMMENT, 'This function adds an user'>

- **Rule Engine** - After having every element of the script standardized into tokens, the tool uses rules to try to recognize patterns that give away the existence of a code smell. Each rule consists on the combination of several functions that recognize different patterns in the tokens. Each function uses a regular expression[9] to detect that pattern. If the combination of all functions happens, then the tool knows that the smell type associated with that rule is present in the script and it returns an alert. Here is the rule used to detect the previously mentioned *Admin by default* code smell type:

```
(isParameter(x)) ^ (isAdmin(x.name) ^ isUser(x.name))
```

[8]https://github.com/akondrahman/IacSec/tree/master/SLIC
[9]A regular expression is a pattern that can be matched against a string of characters [16].

This can be translated as having a token that is a parameter and it represents an user which is also an administrator. For each function, a regular expression allows the detection of that pattern in text. The regular expressions used in this rule, plus a more complex one used to detect if a private key exists inside a *Token* are represented in table 2.2.

Table 2.2: Examples of regular expressions used by the SLIC Rule Engine Module

Rule function	**Regular Expression used**
isAdmin()	‘admin’
isUser()	‘user’
isPvtKey()	‘[pvt|priv]+*[cert|key|rsa|secret|ssl]+’

The end result of the script, for each line of *Puppet* code analyzed, is either the confirmation that no code smell was detected or the type that was detected.

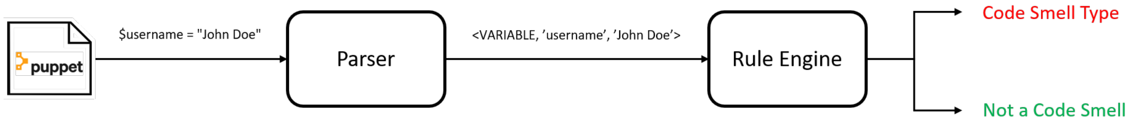


Figure 2.3: Diagram of the SLIC Architecture

Chapter 3

Building a better security linter for Puppet

3.1 Taking a better look at SLIC

Rahman et al., in his study of the SLIC tool [13], reported near perfect precision and recall values for his SLIC software. These results raised some questions, especially because achieving these kind of results in a tool that performs very simple pattern-matching in text, without any awareness of the context, seems to be too good to be accurate. These worries were addressed in a later independent study [17], where the tool was evaluated against different and much more varied datasets, composed of *Puppet* scripts from several open-source projects from *GitHub*, with the results being represented in table 3.1.

Two evaluations were conducted in the study:

- Firstly, a sample of 9144 *Puppet* scripts, where SLIC detected security smells, was collected. These where then evaluated by the authors of the study by hand in order to understand if they were actual problems or just false positives[1]. In the end, the overall precision of the tool never got over 64% for every type of security smell detected.

- Intrigued by the preliminary results, the authors conducted a second study, where an automated tool was built to run SLIC in several open source GitHub repositories. Any detected alerts were automatically posted as issues on the respective projects. The high number of false positives ended the project abruptly and after analyzing the responses from the authors, the final precision came around 8%.

3.1.1 The importance of precision in a code analysis tool

When studying code analysis tools, precision is one of the most important measures of the overall performance of a program, as it allows for a representation of the total number of false positives

[1]A warning from the tool is considered a false positive when it is determined that the issue does not exist or the developer considers that it does not affect the overall software security.

Table 3.1: Comparison of the precision of SLIC in the several conducted studies

Study	Overall Precision
Original study by the author	*99%*
Preliminary Independent Study	*64%*
Large Scale Independent Study	*8%*

generated. One of the main difficulties faced by this kind of tools is the high presence of alerts that are then dismissed by developers as being false or simply not important enough to be dealt with. An high presence of these in the program results will seriously affect the confidence in the tool, besides being an enormous inconvenience and disruption of a product development workflow [15]. Most developers try to maximize their work efficiency and analyzing false positives is a waste of that same precious time, meaning that this is, in many occasions, the main reason why a tool like this is abandoned.

This is why precision was the highlighted result from the analyzed studies, and by comparing the results, it is possible to conclude that SLIC has significant problems in this area. Just by observing the results obtained by each study, it is possible to conclude that:

- The original author of the tool seems to have significantly overestimated the performance of his tool. Besides the ethical problems this might bring, the false expectations created in potential users will only make the experience worse. As they expected a near perfect list of results, the tolerance for the many errors that are expected (from the previous results) to occur will be minimal and will certainly drive them away immediately from this kind of tools.

- The results from the independent study reveal a much darker picture, where the large majority of warnings that the tool produces are completely false. For a developer, this means massive amounts of time wasted looking for rare true positives that may or may not appear in the middle of the list of estimated 92% of completely false alerts.

Considering all of this, good precision will definitely be the main goal of the projected tool, as it constitutes one of the most important metrics and factors that potential developers will take into consideration when evaluating its value for their projects.

3.1.2 Poor User Experience is also driving developers away

Besides the low precision reported by the tool, the analysis of the feedback from the several different developers that were targeted by one of the conducted studies [17] also reveals that they do not approach all security alerts the same way. Some of the detected smells were definitely severe and they needed to be dealt with as soon as possible. However, this is not the case with most of them, where their resolution can wait for less intensive development periods or not even be important enough to spend time addressing them. Unfortunately, SLIC treats all of them in the exact same way, creating a large and unorganized list that developers need to sort through manually. In larger projects, where the number of warnings can grow considerably, it is easy to understand that this will negatively impact the user experience. Organizing this list in a hierarchical structure that makes sense to developers and combining it with a measure of the importance of each detected issue goes a long way in not only providing a much more pleasant User Experience, but also in easing the burn of dealing with non-relevant or even false warnings [9].

SLIC is also not customizable at all. This means that developers are not able to adjust how or which warnings are analyzed and returned by the tool. This has a considerable impact on the type and amount of potential false positives that might be displayed. For this reason, developers highly value the possibility of customizing the analysis process, by changing the parameters on how different types of warning are detected or even by suppressing some completely if they do not find them relevant [9].

Another issue with the tool is the very weak output it produces. Simply stating which type of vulnerability was detected, without providing any explanation, context or suggestion on how to fix it makes it very easy for developers to simply ignore them, contributing to a bad and dangerous experience, where important flaws might get overlooked [9]. This fact combined with the total lack of organization of the output results constitutes an additional reason for developers to overlook this solution on their *Puppet* project.

Finally, developers usually do not stop their development process to run additional analysis tools [15]. SLIC is an isolated *Python* script that is not incorporated into any kind of development tool or workflow. This creates an additional barrier for its use, meaning that either by lack of will to perform an additional command or just by pure laziness, developers might not run the tool regularly, causing a pilling up of warnings that catalyzes the previous analyzed user experience issues. A new and improved tool, to have a chance to be regularly used by developers, needs to integrate into the development cycle, either by running periodically in the code infrastructure or by detecting issues in real time inside the programming software.

These problems reveal the importance of a good User Experience even in a simple code analysis tool. This can either convince users to use the tool regularly, spending time in learning how it works, customizing and optimizing it for their needs and getting the benefit of having an additional confirmation of their code quality, or drive them away from static analysis altogether after a frustrating experience. It also highlights the fact that SLIC is still far away from being a feasible tool for anyone who wishes to analyze their potentially vulnerable *Puppet* code.

3.2 Where can improvements be made?

It is clear that SLIC has a lot of room to improve before it can become the standard security analysis for *Puppet* programmers. As it was described in section 2.3.2, it has a very simple architecture which reflected itself on the very naive errors committed during the precision studies described in section 3.1. It became clear that a critical step to avoid the false positives presented by the tool is getting an idea of the context where the *Puppet* code is executed. The instructions used and even the naming of the variables can have completely different meanings depending on the programmer, the software where is inserted and even the section where is written. Besides, developers, due to several external factors like the security level of the tool or budget, can intentionally use one or more smell types in the code.

Alongside with the problems with precision, User Experience is also an area for big improvements. Like discussed in section 3.1.2, it plays an important role in the decision of developers to use or not use the tool. It also became clear that developers are not going to get out of their development routines to run a separate tool neither deal with large, unorganized lists filled up with false alerts.

3.2.1 An improved rule engine

The architecture of SLIC, analyzed in section 2.3.2, has two main sections of data processing: A Parser which is able to analyze every line of code and convert it to a standard token form; and a rule engine which is responsible for analyzing the generated tokens against a set of rules and regular expressions that represent the different types of code smells.

After a preliminary analysis of the source code, the Parser module seems to accomplish its goal of translating all different lines of code into a standardized and easy to analyze form. Because of this, very few changes can actually be made to improve it and it will, most certainly, be reused in the new and improved analyzer.

The biggest source of problems is, by far, the Rule Engine. After a preliminary analysis of the author's paper [13] and the source code, it is evident that the tool is still very simplistic in its approach to the problem in hand. The simple pattern matching of certain words and expressions is behind the vast majority of false positives returned to the user. The only way of improving the precision, like discussed previously, is going further in terms of complexity of the analysis. This could theoretically be accomplished in several ways:

- **Customizable rules** - Developers, better than anyone else, know exactly the context of the program and all special factors specific to it that have to be taken into account during a security analysis. They also know which kind of bugs and smells they are more prone to commit and as such, they know which kind of rules should be enforced [11]. This means that the rules cannot be hard-coded into the Rule Engine like they are currently on SLIC. Instead, they need to be adaptable to any type of configuration made by the user. There could also be the possibility for the introduction of new rules by the developers anytime they find it adequate.

- **Machine Learning** - To better understand the context in which the instructions are inserted, comparison with other scripts with similar contexts and which are known to be secure can be very valuable. Machine learning can be used here as a way of making the Rule Engine very powerful and aware of different development patterns and types of software. By having a model which is constantly trained[2] using, for example, other open source *Puppet* scripts that are known to be secure and have a very similar application context [1], it is possible to detect any unusual variation that could certainly be a signal of the presence of a vulnerability.

Other improvements that could have a positive impact on the precision of the tool will also be analyzed.

3.2.2 Providing a better User Experience

Besides an increased precision, ensuring a good and pleasant user experience is also vital to encourage developers to adopt this new software. The current SLIC tool is basically an isolated *Python* script that is run manually against isolated *Puppet* scripts. As it has been seen previously in 3.1.2, developers do not want to constantly stop their development process to periodically run code analysis tools. Because of this, it is imperative that the new tool becomes part of any development process. To satisfy this requirement, the new tool will integrate seamlessly with several different IDEs, meaning that code alerts will be provided in real-time right in front of developers. Instead of having to worry about running it manually, the tool will provide its output automatically where it matters the most: At the place where the vulnerability is in the code. This will effectively make the tool an integral part of the entire process.

In addition, the provided alerts will have to be classified according to their importance and urgency to be addressed. Using a simple classification system of colors displayed [9], for example, will enable developers to have an immediate idea of the severity of the potential smell they just wrote.

Lastly, for each kind of code smell detected, an explanation and several examples of how it can be a problem will be provided in a "More information" button. This will contextualize the user on the consequences of the code smell he iAntroduced. Machine learning could once more be used to suggest a possible fix to the problem based on other pieces of code that are known to be correct.

[2]A model is a program that can be trained via the observation of several examples that are known to be correct. It is able to identify patterns and similarities between them that then can be used in analyzing new data. By trying to apply these learned patterns to the new data, it can try to classify it. Classifications are possible, like in which type from several it is part of, if its relevant or not to the training data or even making a mathematical prediction of a certain value, by the use of regression algorithms.

Chapter 4

Taking a deeper look into the tool's Architecture

4.1 Challenges addressed

The tools' architecture was developed with several challenges in mind: From the need to integrate with several different IDEs to allow for a much better User Experience and integration with most developer workflows, to having highly customizable rules able to be tuned up for every situation.

With these in mind, the end result was an highly modular architecture, with possibilities for user customization and also future expansion in several key components, from new rules to IDEs and even configuration interfaces.

4.1.1 Better integration with developers' workflow

One of the main problems of the existing SLIC tool is the fact that it is only a command line linter. This means that any developer interested in analyzing his code needs to stop working to manually run the linter. Unless a CI/CD solution capable of running the software periodically is used, the most likely scenario is that users will eventually stop running the command, as that is a major disruption of their workflow [15]. Even with an automated solution, the feedback will not be immediate, which might promote the accumulation of vulnerabilities and the increased likelihood of them not being properly addressed.

As such, the proposed security linter needs to integrate much more tightly with the average developers' workflow, allowing them to get instant feedback on their code without any extra effort. For that, the architecture was built from the ground up with abstraction in mind, separating the core functionality from the presentation. With this system, different plugins can be built for any IDE on the market, all working from the same universal core.

Finally, the developer ends up with a tool that presents all vulnerabilities right were they will be definitely noticed - on the code being developed.

4.1.2 User customization of rules

All software projects are different, and Puppet is not immune to that. Rules that might be obvious in some situations can be completely unthinkable to enforce in others. As such, one of the most critical features of any Static Analysis tool is the ability to be user configurable [9].

To overcome this important challenge, the architecture was built with a central system to manage, display and apply all sorts of configurations to every rule. This system is what makes it possible for users to easily customize the exact behavior of the linter on their particular project, ultimately reducing the number of false positives generated.

4.1.3 Powerful vulnerability detection rules

Arguably the main problem with the existing SLIC tool was that high presence of false positives. This was due, among other factors, to the low complexity of the applied rules, being mainly based in non configurable regular expressions. This was made worse by that fact that adding new rules was neither easy or officially supported.

Being expandable to new challenges was a priority from day one. The designed rule architecture is fully modular, allowing for an easy future expansion with new fully user configurable rules. Tomorrow, a new developer can come around adding new powerful features like machine learning or natural language processing with ease, allowing for almost endless possibilities on the vulnerability detection capabilities.

4.2 Architecture Overview

Considering the challenges presented previously, an UML Class diagram with the software architecture for the Security Linter was designed. It includes practically all classes that were developed in the final code.

The usual workflow starts either by the Command Line Interface or via the LSP server classes, if the user decides to use an IDE such as Visual Studio Code. These classes serve as the gateway to the logical core of the analysis, providing access to the content of the file and receiving the list of results from the analysis to be presented to the user.

The most important class in this entire schema is without doubt the Rule Engine. It is the center piece that coordinates all activities. It keeps track of all rules currently created and being applied, being responsible for receiving the content of the file and controlling which rules will be called and applied, returning the end result. It is also responsible for transforming the received Puppet code into a list of tokens, importing for this task the Tokenizer class from the existing Puppet-Lint gem.

The Rules themselves are represented by their own class. It has all the logic that evaluates if a certain kind of vulnerability is present in the list of tokens being analyzed. If yes, it then returns the list of locations where that happens.

Inside these rules lives the Configuration class, being home to the value and properties of all user-configurable aspects of the applied rules. There are several configuration subclasses according to the data type of the configuration, which require different kinds of input fields to be presented to the user. Another possibility is having rules that are called by other rules, very useful if there is the need of sharing the same logic across different rules.

This is the normal flow of information, from the input of a file to analysis until the production of the list of vulnerabilities found, represented by the Sin class that can also have different types.

Another important feature of the tool is the amount of user customization supported. The user can configure the rules' behavior via a HTML page, generated and managed by the ConfigurationPageFacade class. These rules are then persisted in a file using the ConfigurationFileFacade class. Both of these classes use the ConfigurationVisitor to obtain the list of all configurations present on the system that are spread across all rules.

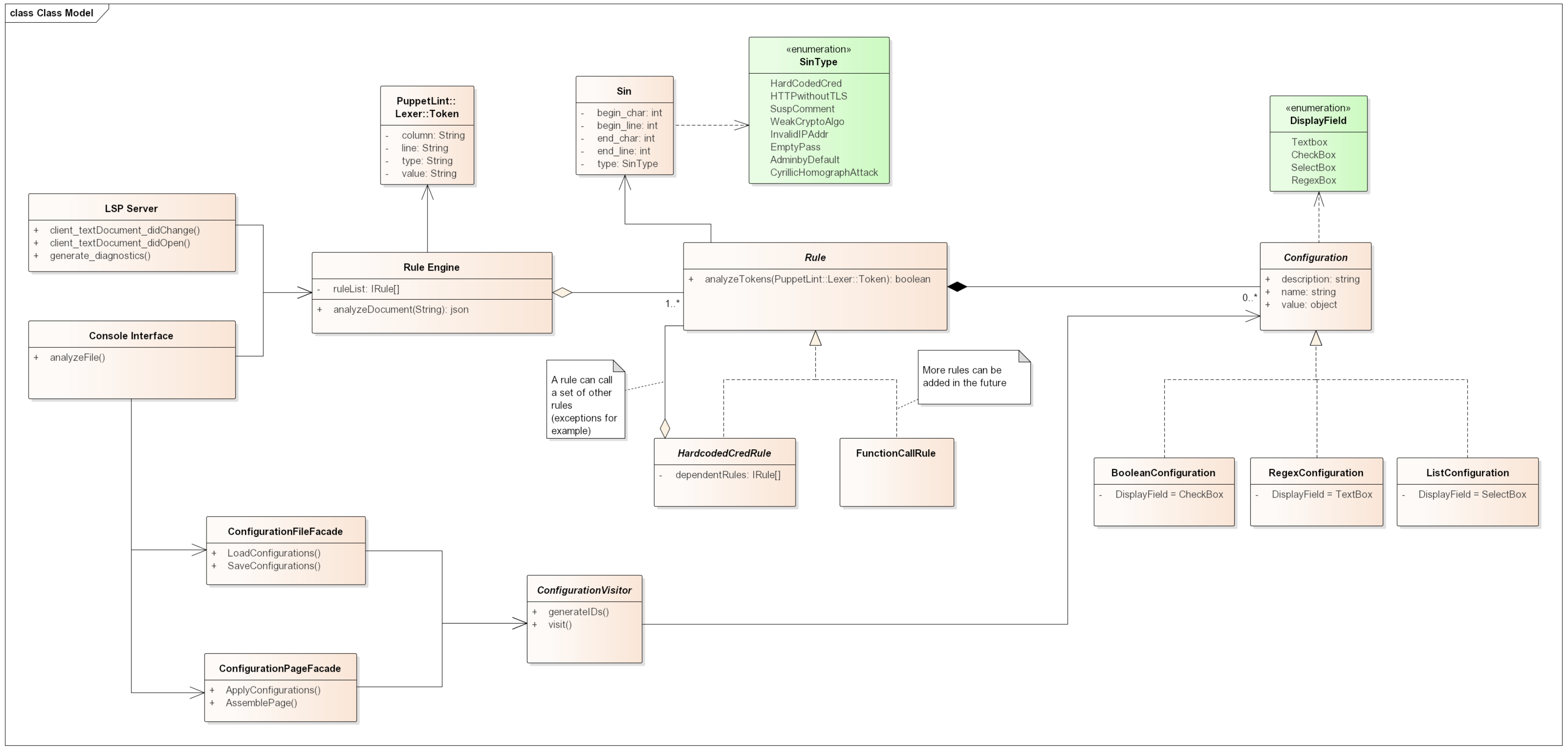


Figure 4.1: Puppet Security Linter Architecture

4.3 In-depth sections and design patterns used

4.3.1 Rule Engine

A linter is a collection of different rules that are applied over a piece of code that it being evaluated. This tool is no different. As such, a solution to properly organize all rules is essential to ensure the maintainability and expandability of the software.

An easy solution would be to just have all rules being applied to the same code file with conditional branches to determine what kind of tokens should be evaluated there. This is not a very good solution as all the code is concentrated on the same place, making hard to read and properly maintain.

As such, a better solution was designed, based on the Rule Engine design pattern [6]. All rules have their own class, based on an abstract Rule class. All of them share a common method that executes the logic. The rule engine has the important role of keeping track of all rules, choosing and executing them.

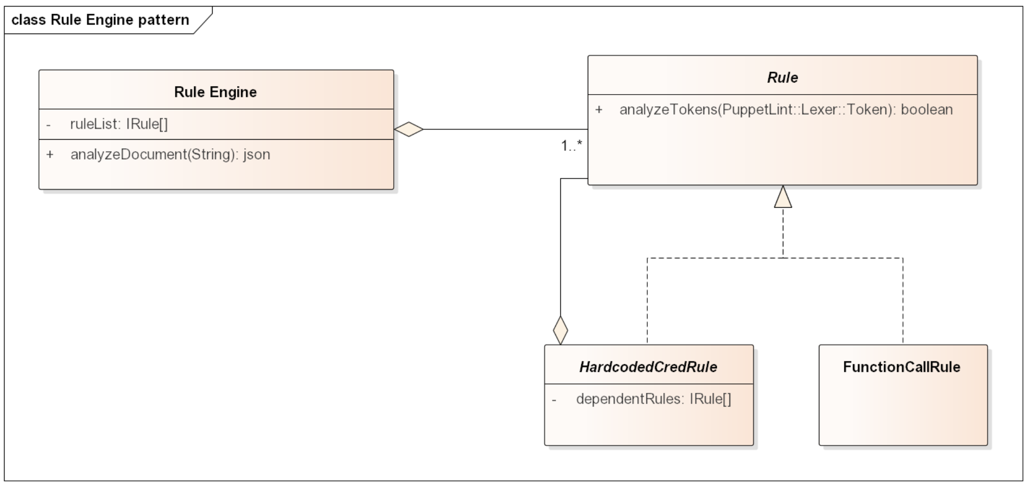

class Rule Engine pattern
Rule Engine
- ruleList: IRule[]
+ analyzeDocument(String): json
Rule
+ analyzeTokens(PuppetLint::Lexer::Token): boolean
1..*
HardcodedCredRule
- dependentRules: IRule[]
FunctionCallRule

Figure 4.2: Implemented Rule engine pattern classes

This ensures that all rules are implemented on separated independent classes, allowing for a much greater modularity. This in turn makes it easier to maintain the current code and also to improve or add more rules in the future, ensuring the continuous improvement of the quality of analysis as analyzed in the next component.

4.3.2 Modular Rules

Precision is an essential aspect of any code analysis tool [15]. This is directly correlated with the ability of the rules to detect the correct security vulnerabilities. After recognizing that achieving great levels of precision without a group of fairly sophisticated rules and that these involve working with complex technologies (like Natural Language Processing or Machine Learning) that are projects on their own, the best way to ensure that the tool can eventually achieve this level of precision is to design it in a way that future projects can build upon it. This demands a rule architecture able to be modular and easily expanded.

As such, based on the *Template* design pattern [7], the designed system makes it so that every rule is its own logical class, derived from a common abstract one. This makes it possible to see and run all rules the same way (making it possible for the rule engine to work), but with each one having it is own logic implemented behind the common method.

In the future, another developer interested in improving the tool precision or adapting it to new scenarios can easily add new rules, without ever having to modify anything from the existing ones. Much more complex technologies can, as such, be integrated later on.

4.3.3 Centralized Configuration Management

A very requested feature of any linter is the ability of the user to customize the behavior of the analysis [9]. For this same reason, it was given an high importance to the design and incorporation of a customization system in the *Software Architecture*.

Because of the previously mentioned *Modular Rules*, and also because each Rule has its own set of configurations, hard-coding them inside the already developed Rules would not be a feasible solution. As developers can change and add Rules at will, the system needs to be able to accept new Configurations without too much trouble. Also, the configurations are all spread across different Rules, so without a default policy of uniformization, it would be impossible to provide to the user a centralized place to customize everything. At worst, most rules would have to be changed in different places or even in the source code itself, creating an extremely user unfriendly system to use and making it very difficult to be adopted by the Puppet Community.

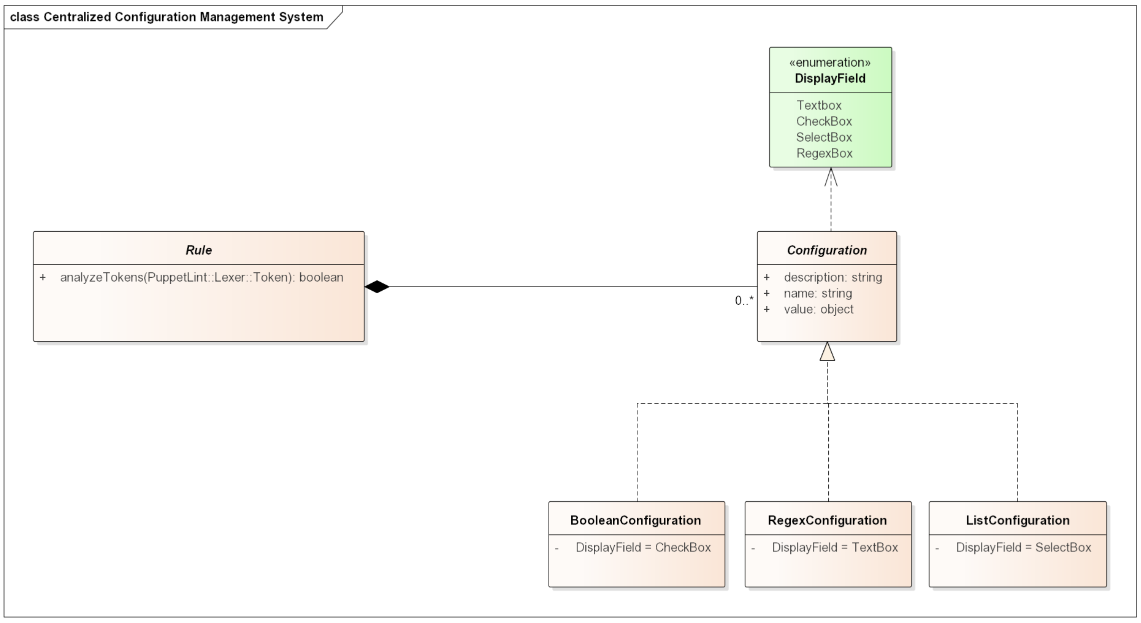


Figure 4.3: Centralized Configuration Management System

To avoid this situation, a Centralized Configuration Management system was designed. Each Rule has a set of different objects from the abstract Configuration class, representing the different parameters that can be user customizable. Inspired by the *Strategy* Design Pattern [7], each subclass from the abstract one represents a different type of data, as different types of data require different kinds of fields to be displayed and also different strategies to process and validate that data. Inside each Rule, all configurations are instances of these subclasses according to the type of data stored in them.

Inspired by the *Visitor* design pattern [7], these configurations that are spread across different rules are all managed by a Visitor class who goes around all Rule classes and lists every rule that exists, creating effectively a list of all configurations in the system. Because this list is compiled in execution time, it allows each rule to be developed and customized any way the developer sees fit, with worrying updating any centralized configuration list in the code. This list is then used to centrally manage all configurations.

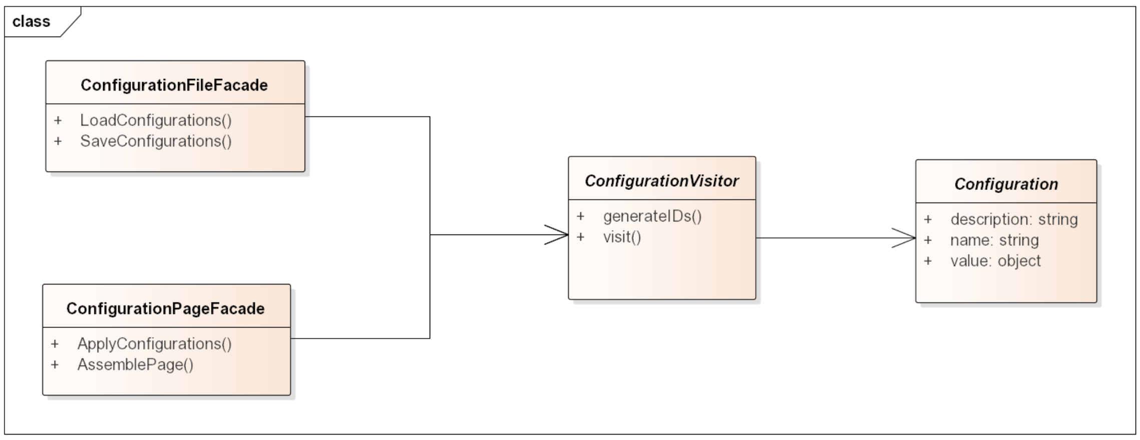
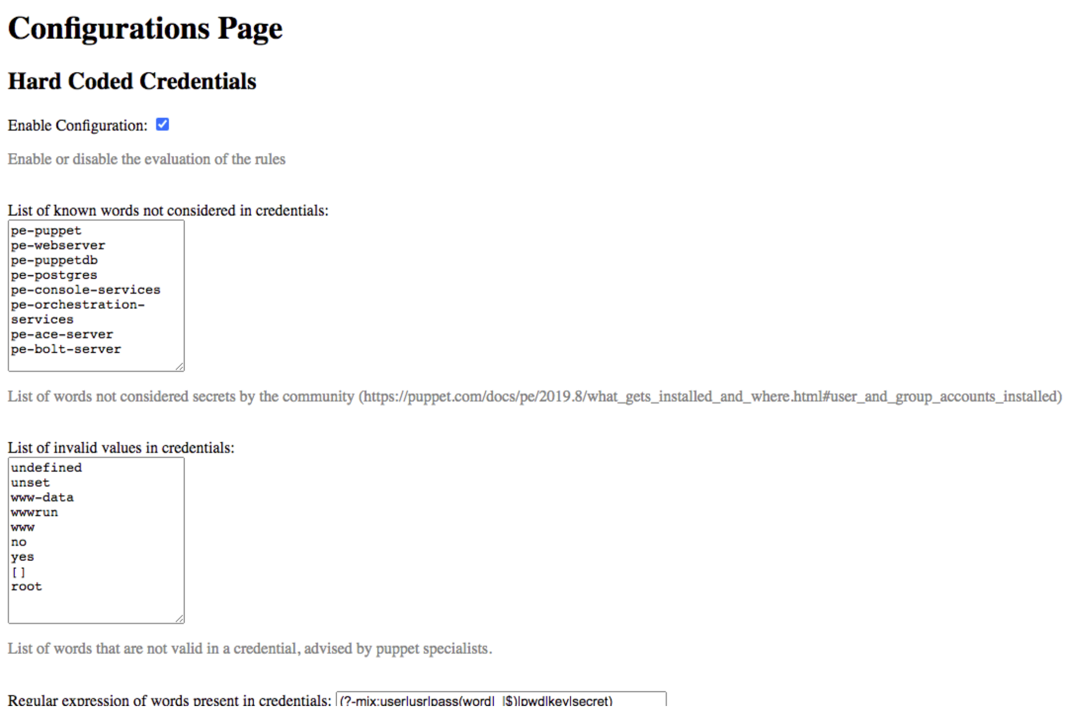


Figure 4.4: User interfaces for configuration management

4.3.3.1 Configurations' Web Page

In any configurable system it is essential to the User Experience that the user is able to configure it from a single place, preferably at run time without any sort of source code modification. To achieve this, an HTML web page is generated at runtime containing all data fields to edit any kind of configuration data needed. The user can then, from his web browser, manage the entire system.

This is done using a system based on the *Facade* design pattern [7]. The Facade class provides this simple to use web interface to the user to view and manage all configurations in the system. Behind the scenes, the class is responsible for using the previously referenced Visitor to compile the list of all configurations and then it generates the HTML code adequate to each element of that list, meaning that different kinds of configurations get different data fields displayed.



Configurations Page

Hard Coded Credentials

Enable Configuration: ✓

Enable or disable the evaluation of the rules

List of known words not considered in credentials:

```
pe-puppet
pe-webserver
pe-puppetdb
pe-postgres
pe-console-services
pe-orchestration-
services
pe-ace-server
pe-bolt-server
```

List of words not considered secrets by the community (https://puppet.com/docs/pe/2019.8/what_gets_installed_and_where.html#user_and_group_accounts_installed)

List of invalid values in credentials:

```
undefined
unset
www-data
wwwrun
www
no
yes
[]
root
```

List of words that are not valid in a credential, advised by puppet specialists.

Regular expression of words present in credentials: (?-mix:user|usr|pass(word|_|$)|pwd|key|secret)

Figure 4.5: Web Page for User Configurations

After the user is done with tuning up the system, pressing save makes the Facade read the values of all HTML fields and store them back into the original Configuration classes at runtime, effectively allowing the system to be configured in real-time.

4.3.3.2 Configurations' File

Another important aspect is the persistence of the configurations. It is not practical to demand a system configuration everytime the user needs to analyze a different code file. As such, a way to persist the user settings was needed.

This was achieved again by using the *Facade* Design Pattern [7]. Similarly to the Configuration's Web Page, the class uses the Visitor to obtain the list of current configurations. Then, when the User modifies these configurations, the class is responsible for generating a .ini file with all values. This file is then persisted in the machine file system. When the program is called again for a new analysis, the same configurations facade is called, but this time it reads the contents of the settings file and makes sure that all configuration classes have the appropriate values.

This way, the system can maintain all User defined configurations everytime a new analysis is done, contributing to a much better User Experience. It is also technically possible to use this file to configure the system, bypassing the previously mentioned Settings Web Page in systems that do not have a graphical user interface for example.

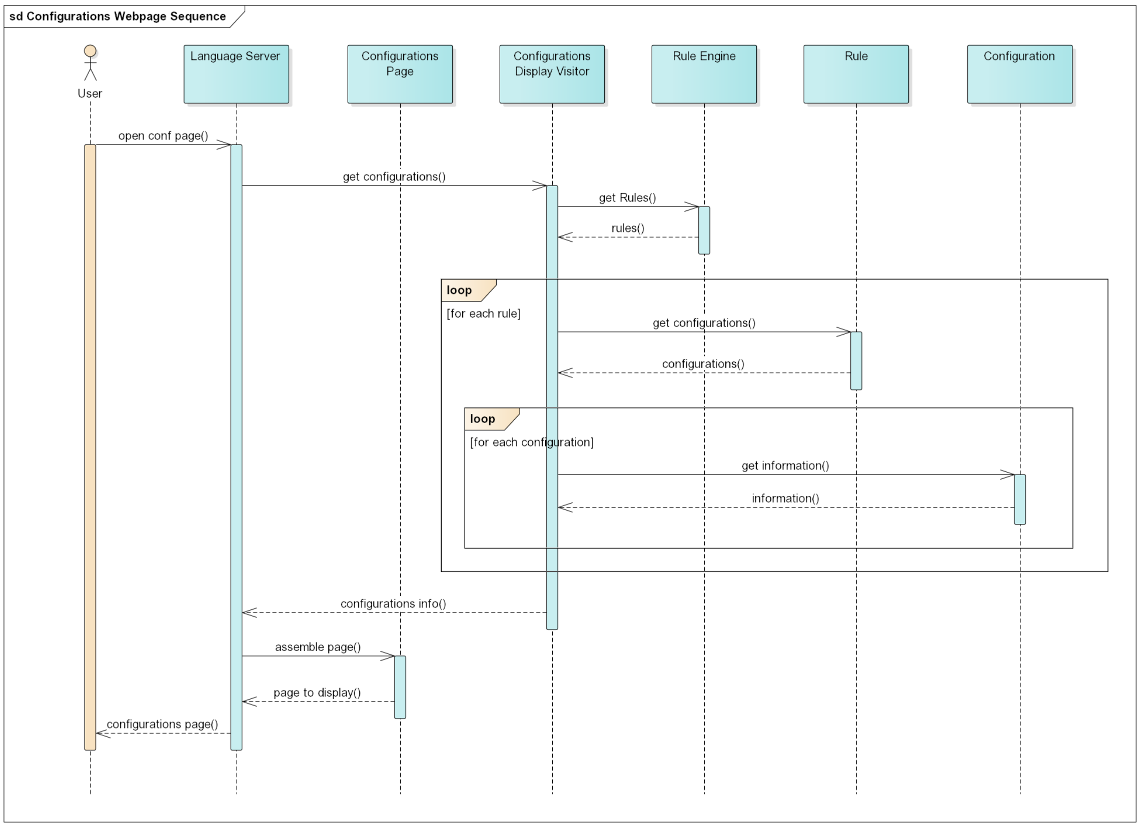


Figure 4.6: Configurations Webpage Sequence Diagram

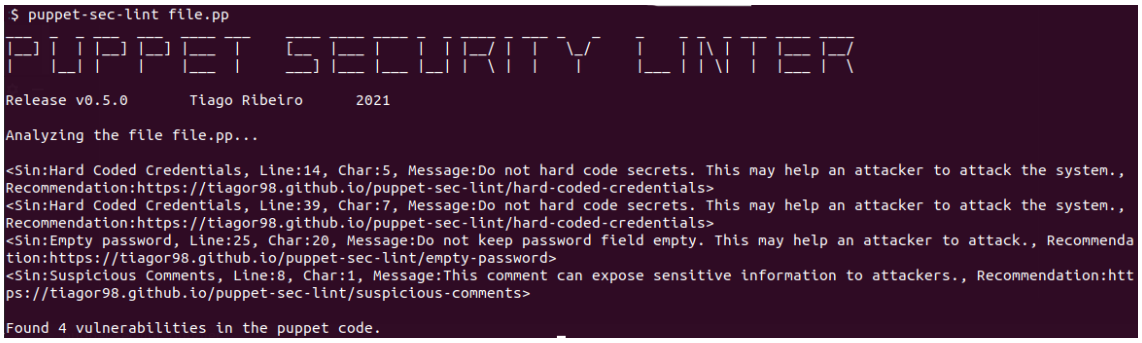
4.3.4 Interaction with the User

The interaction with the user is arguably the most important contribute to a positive User Experience. It is what the User sees and uses to interact with the system, inputting data and obtaining the resulting output.

For this linter, besides the Console Interface standard in any software of his kind, an LSP interface was also design, allowing for an easier integration with IDEs.

4.3.4.1 Command Line Interface

Similar to many code linters on the market, the user can perform an analysis from the Command Line interface.



```
$ puppet-sec-lint file.pp
 ___  _  _  ___  ___  ____ ___    ____ ____ ____ _  _ ____ _ ___ _  _    _    _ _  _ ___ ____ ____
 |__] |  | |__] |__] |___  |     [__  |___ |    |  | |__/ |  |   \_/     |    | |\ |  |  |___ |__/
 |    |__| |    |    |___  |     ___] |___ |___ |__| |  \ |  |    |      |___ | | \|  |  |___ |  \

Release v0.5.0       Tiago Ribeiro      2021

Analyzing the file file.pp...

<Sin:Hard Coded Credentials, Line:14, Char:5, Message:Do not hard code secrets. This may help an attacker to attack the system.,
Recommendation:https://tiagor98.github.io/puppet-sec-lint/hard-coded-credentials>
<Sin:Hard Coded Credentials, Line:39, Char:7, Message:Do not hard code secrets. This may help an attacker to attack the system.,
Recommendation:https://tiagor98.github.io/puppet-sec-lint/hard-coded-credentials>
<Sin:Empty password, Line:25, Char:20, Message:Do not keep password field empty. This may help an attacker to attack., Recommenda
tion:https://tiagor98.github.io/puppet-sec-lint/empty-password>
<Sin:Suspicious Comments, Line:8, Char:1, Message:This comment can expose sensitive information to attackers., Recommendation:htt
ps://tiagor98.github.io/puppet-sec-lint/suspicious-comments>

Found 4 vulnerabilities in the puppet code.
```

Figure 4.7: Command Line Interface

From this Command Line it is possible to perform any action, using flags to specify which one the user needs. He can just start the LSP server running the command without anything. Specifying a file or a folder will make the software perform an analysis recursively (in the case of the folder), displaying the results immediately on the console itself. There are also flags available to launch the configuration page, run in Verbose mode or even to specify in which port the LSP server should run in.

From the Architecture point of view, this is the main entry point which gives access to all previously mentioned components and also to where their output is conducted to.

4.3.4.2 Language Server Protocol

As previously mentioned, one of the options to the the software is to start a LSP server. The Language Server Protocol was created to address the problem of how to implement multiple languages on different IDEs without replicating the code [3]. By decoupling and centralizing the language logic on a server, all IDEs only need to send and receive a standardized JSON-RPC message to process the input of the user. This way, the same implementation of a language can be shared in real time across multiple IDEs with minimal effort.

In this particular case, the LSP server is responsible for receiving the code of the Puppet Manifest currently open and returns to the IDE the location and description of the security vulnerabilities. This means that only a simple plugin is needed for each IDE capable of handling the message exchange with the LSP Server and displaying the results on the screen (which in a lot of cases, like in Visual Studio Code, can be easily accomplished with an API). This makes it possible to incorporate the software with virtually any IDE in the market that supports expansion by Plugins, making the application accessible to even more developers.

For the purpose of this project, a plugin was developed for Visual Studio Code, as described in 6.4. This plugin sends to the LSP server the code that the user is currently working on and receives the location of the security vulnerabilities if found. The LSP server simply calls the previously mentioned Rule Engine in 4.3.1, which is responsible to conduct the analysis process and return the list of results back.

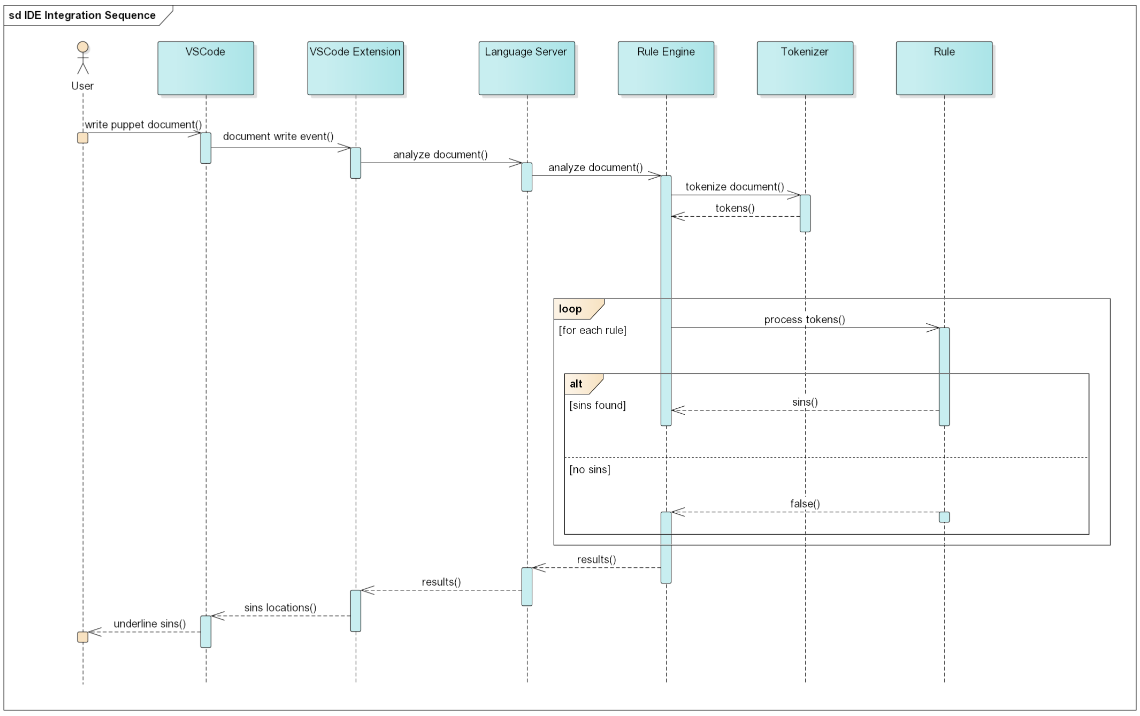


Figure 4.8: IDE Integration Sequence Diagram

4.4 Educational Purpose

The main function of a security linter is to analyze the code and detect any vulnerability that might compromise the system if left unchecked. The developer can then look at them and decide if he wants to address them or not. But what happens if he does not know how to address them or what those vulnerabilities even mean? As such, one of the main priorities in building this linter is not only making sure that vulnerabilities are detected and presented in a convenient way, but also to educate the user on the importance of addressing them. This is the best way to ensure that not only these warnings are properly addressed by anyone who uses the tool, but also that developers in general become more knowledgeable about how they can avoid introducing these vulnerabilities into their code in the first place. This is arguably one of the most noble and important contributes the tool can give to the Puppet Community and to the security and future of the language itself.

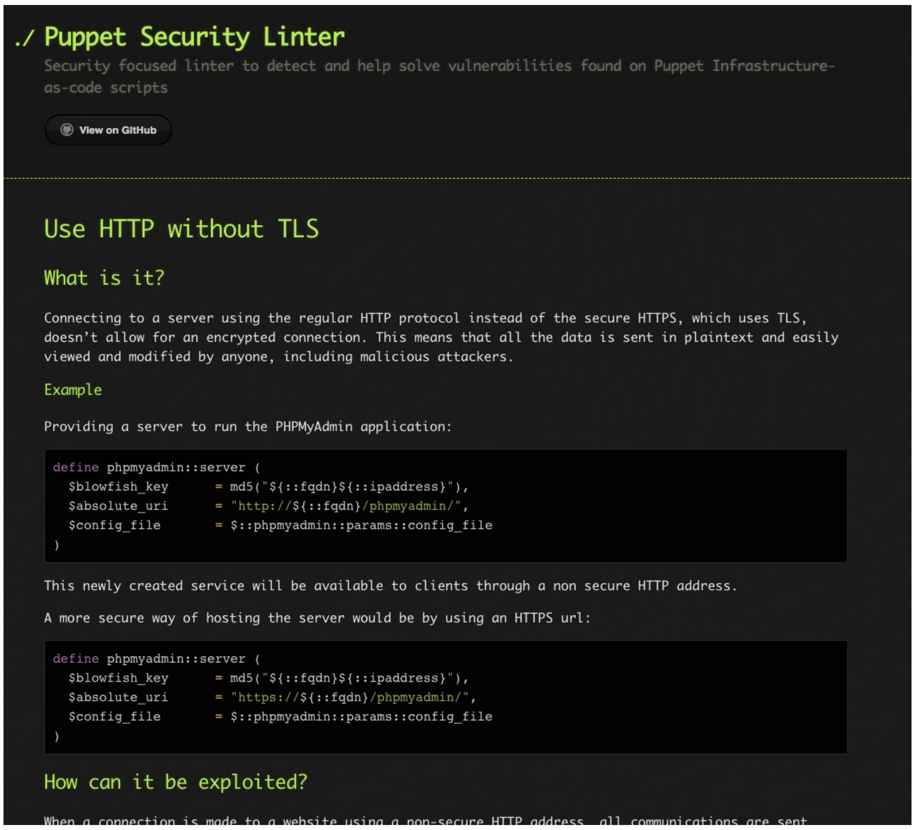
To achieve then this critical goal, besides providing a simple description of the warning inside the interface of the IDE, a page was made for each type of vulnerability. This page contains a more detailed description of the problem. More specifically:

- **What is it?** - Section dedicated to explain in a more detailed way in what exactly the vulnerability consists on. Some examples are also provided to demonstrate how this vulnerability can be found in a Puppet Manifest. The goal is to contextualize the developer on what exactly the warning message means.

- **How can it be exploited?** - After contextualizing on what the vulnerability is, in this section it is demonstrated how a potential attacker could exploit it when targeting a system. The goal is to demonstrate to the user why it is important to address this vulnerability by showing the possible consequences of having it in the system.

- **How to avoid it?** - The final objective of this page is to convince the end user to address the vulnerability found on his code. As such, it is critical to show how to actually accomplish this, as it is natural for someone who does not even know about the existence of the vulnerability to have no clue on how to mitigate it. Some examples might be provided on how to do it with a Puppet Manifest, having the user only to adapt these to the context of his code.

- **More related information** - At the end, links to additional resources that detail the importance of the vulnerability are provided for further consultation and to validate the content of the page itself.



./ Puppet Security Linter

Security focused linter to detect and help solve vulnerabilities found on Puppet Infrastructure-as-code scripts

Use HTTP without TLS

What is it?

Connecting to a server using the regular HTTP protocol instead of the secure HTTPS, which uses TLS, doesn't allow for an encrypted connection. This means that all the data is sent in plaintext and easily viewed and modified by anyone, including malicious attackers.

Example

Providing a server to run the PHPMyAdmin application:

```
define phpmyadmin::server (
  $blowfish_key      = md5("${::fqdn}${::ipaddress}"),
  $absolute_uri      = "http://${::fqdn}/phpmyadmin/",
  $config_file       = $::phpmyadmin::params::config_file
)
```

This newly created service will be available to clients through a non secure HTTP address.

A more secure way of hosting the server would be by using an HTTPS url:

```
define phpmyadmin::server (
  $blowfish_key      = md5("${::fqdn}${::ipaddress}"),
  $absolute_uri      = "https://${::fqdn}/phpmyadmin/",
  $config_file       = $::phpmyadmin::params::config_file
)
```

How can it be exploited?

When a connection is made to a website using a non-secure HTTP address, all communications are sent

Figure 4.9: Example of a Vulnerability details page

These pages are hosted on the GitHub repository of the main core and they are accessible either by the command line, presented as a link on each detected vulnerability, or by the IDE, where the user can click in the link shown in the warning message.

4.5 Ready for future expansion

Arguably the most important aspect of the designed architecture is the attention given to the possibility of expanding the software functionality. Several aspects of the tool were designed in a modular fashion with this precise goal in mind.

Firstly, the compatibility with several IDEs is one of the biggest differentiating features compared to other previous offers on the market. As mentioned earlier in 4.3.4.2, the tool uses a LSP server to make its functionality available to external programs. This makes it so all complicated analysis logic stays on the server side, with the client (in this case, a plugin for the IDE) responsible only for the input and display of data. This makes it very easy to develop new plugins for various IDEs, all compatible with the core software and all sharing the same exact functionality. For this study, only a Visual Studio Code Plugin was developed, but for the future, plugins for other popular programs on the market can be easily developed and integrated into the linter. This ensures that the tool has the potential to be used by virtually every developer in the community, independent of which IDE they use on their workflow, which naturally is a very import condition for it to become the standard among the Puppet Community.

Secondly, the precision of the analysis is a critical aspect evaluated by developers who intend to use a Static Analysis tool. As seen previously in 3.1.1, an high number of false positives is enough to turn away potential users, as it is a big waste of precious development time. Increasing precision means developing much more powerful and complex rules capable of recognizing different contexts and adjusting its analysis accordingly. This clearly requires a great amount of development time and effort, being practically impossible in the context of the current project to incorporate more advanced technologies like Machine Learning and NLP. As such, a big effort was put into developing a Modular Rule system, as described in 4.3.1, meaning that the linter can be later expanded with these more complex rules, developed by dedicated experts. This is very important, as multiple dedicated development teams can contribute and work on the same tool, freeing up the pressure on the original developer to predict and include ever possible functionality. Another important aspect of the world we live in is the constant evolution and appearance of new security threats. This architecture gives to the linter the possibility of adapting to the always evolving threats, allowing for the update of current rules and constant addition of new ones.

Lastly, the configuration system mentioned in 4.3.3 allows for the easy and complete customization of the behavior of the system. Besides the possibility of adding new settings, the already existing ones allow the system to adapt to new and evolving scenarios in various different projects. This effectively means that the tool is adaptable and expandable to be used in new projects in the future.

Chapter 5

Development Process

A considerable part of the project was dedicated to the development of the software itself. The choice of programming tools was then a very important process with a crucial influence in the potential success of the thesis.

5.1 Choosing a Programming Language: Ruby

One of the first decisions regarding the development of the Puppet Security Linter was which programming language to use. Considering that previous linters developed for Puppet (on which this tool is built on), including the tokenization function which is part of the existing Puppet-Lint gem, are all developed in Ruby, the decision to use this language brought great easiness in integrating with all of these external dependencies.

Ruby itself is a very powerful and easy to learn language and it was perfectly possible to implement the designed architecture with it.

The Visual Studio Code extension was developed in Typescript, as this is the default language used in those extensions.

5.2 Code Versioning and Distribution

The tool used for the code versioning was Git, more precisely, the GitHub website [1] [2], as it provides a reliable and easy to use repository able to store and version all the source code. The possibility to have CI/CD pipelines to run unit tests and code analyzers is also a big plus. The author was also already very familiarized with the platform.

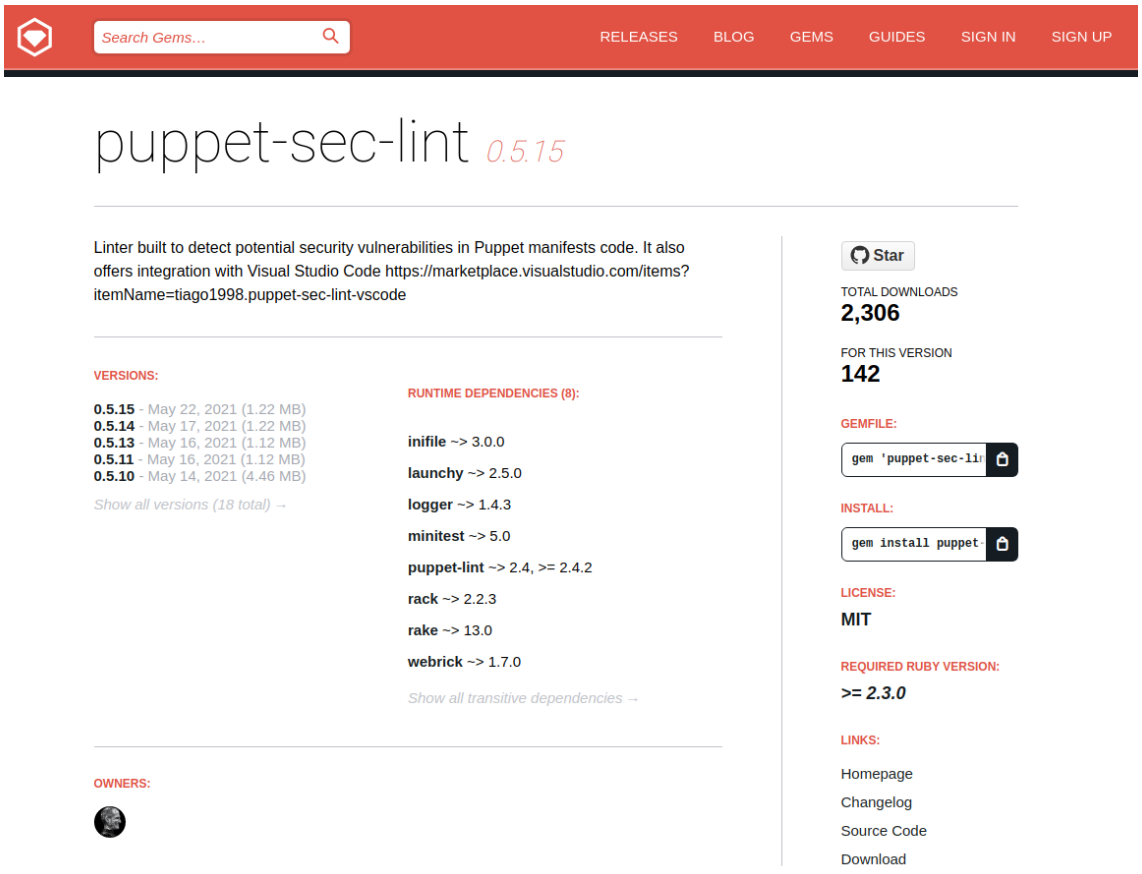
[1] The source code for the linter core is available at Github in `https://github.com/TiagoR98/puppet-sec-lint`

[2] The source code for the Visual Studio extension is also available at Github in `https://github.com/TiagoR98/puppet-sec-lint-vscode`

5.2.1 Using Ruby Gems

The use of Ruby gems [3] to distribute the core software was also a relatively easy decision, as this system allows for a very easy installation on the part of the end user. Considering that he already installed Ruby and Ruby gems on its machine, the installation of the tool can be done with a simple "install" command on the command line. Integration with existing dependencies was also made very easy as gems can be included on each other.

The easiness of publishing a gem from the source code was also a deciding factor in choosing this distribution method.



puppet-sec-lint 0.5.15

Linter built to detect potential security vulnerabilities in Puppet manifests code. It also offers integration with Visual Studio Code https://marketplace.visualstudio.com/items?itemName=tiago1998.puppet-sec-lint-vscode

VERSIONS:

0.5.15 - May 22, 2021 (1.22 MB)
0.5.14 - May 17, 2021 (1.22 MB)
0.5.13 - May 16, 2021 (1.12 MB)
0.5.11 - May 16, 2021 (1.12 MB)
0.5.10 - May 14, 2021 (4.46 MB)

Show all versions (18 total) →

RUNTIME DEPENDENCIES (8):

inifile ~> 3.0.0
launchy ~> 2.5.0
logger ~> 1.4.3
minitest ~> 5.0
puppet-lint ~> 2.4, >= 2.4.2
rack ~> 2.2.3
rake ~> 13.0
webrick ~> 1.7.0

Show all transitive dependencies →

OWNERS:

TOTAL DOWNLOADS
2,306

FOR THIS VERSION
142

GEMFILE:
`gem 'puppet-sec-lin`

INSTALL:
`gem install puppet-`

LICENSE:
MIT

REQUIRED RUBY VERSION:
>= 2.3.0

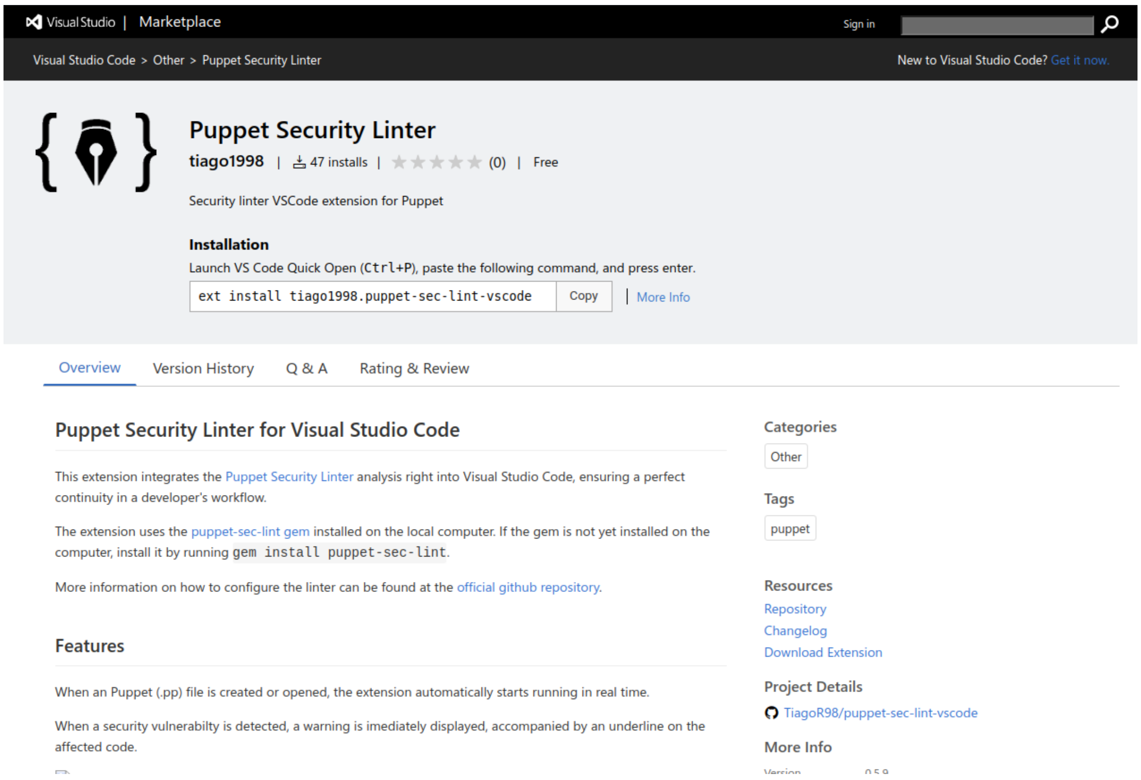
LINKS:
Homepage
Changelog
Source Code
Download

Figure 5.1: Page of the developed linter in RubyGems

5.2.2 Using the VSCode Extension Marketplace

To distribute the Visual Studio Code plugin, the logical choice was the Microsoft provided Marketplace [4]. This is the default way to distribute extensions for the IDE. The publishing process is fairly easy and the end user can install the extension right from the IDE itself.

[3]The gem of the linter core can be installed from https://rubygems.org/gems/puppet-sec-lint
[4]The Visual Studio extension can be installed from https://marketplace.visualstudio.com/items?itemName=tiago1998.puppet-sec-lint-vscode


Visual Studio | Marketplace
Sign in
Visual Studio Code > Other > Puppet Security Linter
New to Visual Studio Code? Get it now.

Puppet Security Linter
tiago1998 | 47 installs | (0) | Free

Security linter VSCode extension for Puppet

Installation
Launch VS Code Quick Open (Ctrl+P), paste the following command, and press enter.

```
ext install tiago1998.puppet-sec-lint-vscode
```

Copy | More Info

Overview | Version History | Q & A | Rating & Review

Puppet Security Linter for Visual Studio Code

This extension integrates the Puppet Security Linter analysis right into Visual Studio Code, ensuring a perfect continuity in a developer's workflow.

The extension uses the puppet-sec-lint gem installed on the local computer. If the gem is not yet installed on the computer, install it by running `gem install puppet-sec-lint`.

More information on how to configure the linter can be found at the official github repository.

Features

When an Puppet (.pp) file is created or opened, the extension automatically starts running in real time.

When a security vulnerabilty is detected, a warning is imediately displayed, accompanied by an underline on the affected code.

Categories
Other

Tags
puppet

Resources
Repository
Changelog
Download Extension

Project Details
TiagoR98/puppet-sec-lint-vscode

More Info

Figure 5.2: Marketplace page of the VSCode extension

5.3 Unit testing rules with Ruby MiniTest

In order to easily validate the correct functioning of the different Rules, a Ruby library called MiniTest was used. It is a widely used way of developing unit tests and in this case it demonstrated total ability to represent the intended test cases.

For each rule, a couple of test cases were developed evaluating if for a piece of Puppet code the rule returned the expected list of vulnerabilities. If any error was made during the implementation of these, it would show up as a failure in the test. This also allows future developers to first develop the tests and then guide the development of the rule on them, following a Test-Driven-Development process [2].

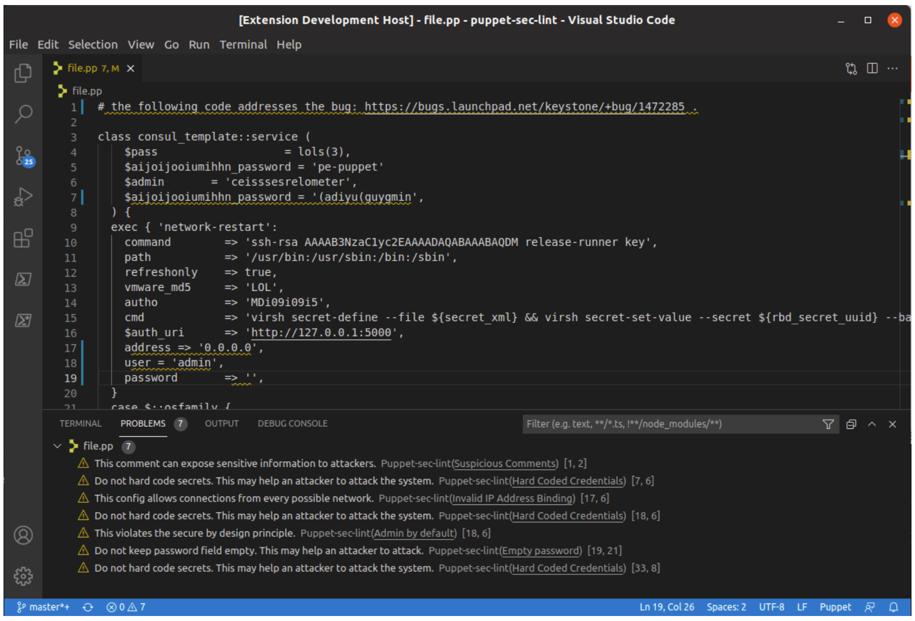
Chapter 6

Working with IDEs

6.1 Fitting into a developer's workflow

As seen previously in 3.1.2, developers do not want to stop their work to evaluate if their code is vulnerable or not. As such, blending with the existing workflows is a very critical requirement for this new software solution.

To achieve this, besides the traditional Command Line interface, the tool was also integrated with the Visual Studio Code IDE, as mentioned in 4.3.4.2. This allows the tool to analyze the code in real time, displaying the detected vulnerabilities right in the IDE window while the code is being developed. This way, the developer gets immediate feedback without having to stop the process to run any external tool. The traditional Command line interface makes it also possible to integrate the analysis with a CI/CD solution to validate any new addition to the code.


[Extension Development Host] - file.pp - puppet-sec-lint - Visual Studio Code
File Edit Selection View Go Run Terminal Help
file.pp 7, M
file.pp
TERMINAL
PROBLEMS 7
OUTPUT
DEBUG CONSOLE
Filter (e.g. text, **/*.ts, !**/node_modules/**)
master*+
0 7
Ln 19, Col 26
Spaces: 2
UTF-8
LF
Puppet

```
# the following code addresses the bug: https://bugs.launchpad.net/keystone/+bug/1472285 .

class consul_template::service (
    $pass                     = lols(3),
    $aijoijooiumihhn_password = 'pe-puppet'
    $admin       = 'ceisssesrelometer',
    $aijoijooiumihhn_password = '(adiyu(guygmin',
  ) {
  exec { 'network-restart':
    command        => 'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDM release-runner key',
    path           => '/usr/bin:/usr/sbin:/bin:/sbin',
    refreshonly    => true,
    vmware_md5     => 'LOL',
    autho          => 'MDi09i09i5',
    cmd            => 'virsh secret-define --file ${secret_xml} && virsh secret-set-value --secret ${rbd_secret_uuid} --ba
    $auth_uri      => 'http://127.0.0.1:5000',
    address => '0.0.0.0',
    user = 'admin',
    password       => '',
  }
  case $::osfamily {
```

```
file.pp 7
  This comment can expose sensitive information to attackers. Puppet-sec-lint(Suspicious Comments) [1, 2]
  Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials) [7, 6]
  This config allows connections from every possible network. Puppet-sec-lint(Invalid IP Address Binding) [17, 6]
  Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials) [18, 6]
  This violates the secure by design principle. Puppet-sec-lint(Admin by default) [18, 6]
  Do not keep password field empty. This may help an attacker to attack. Puppet-sec-lint(Empty password) [19, 21]
  Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials) [33, 8]
```

Figure 6.1: Warnings displayed in real time right in the IDE window

With a more traditional command line only linter, developers needed to stop the development process and manually open and run the analysis themselves. Only then the vulnerabilities would be detected and they would have the chance to address them. This caused several problems as it is not very convenient for developers to perform this manually. A lot of times, the analysis would not be run at all, leaving important vulnerabilities reaching production without detection. Even for those teams that managed to integrate these tools with their CI/CD pipelines, running the analysis only periodically usually leads to an accumulation of vulnerabilities that are more difficult to deal with than if they were detected right at the moment they were written.

This is where IDE integrated tools like this one shine. As soon as a new vulnerability is introduced, the developer gets an immediate warning and has the opportunity to address it while his mind is still focused on the feature in hand, not later when the feature might already be implemented, being harder to make any corrections.

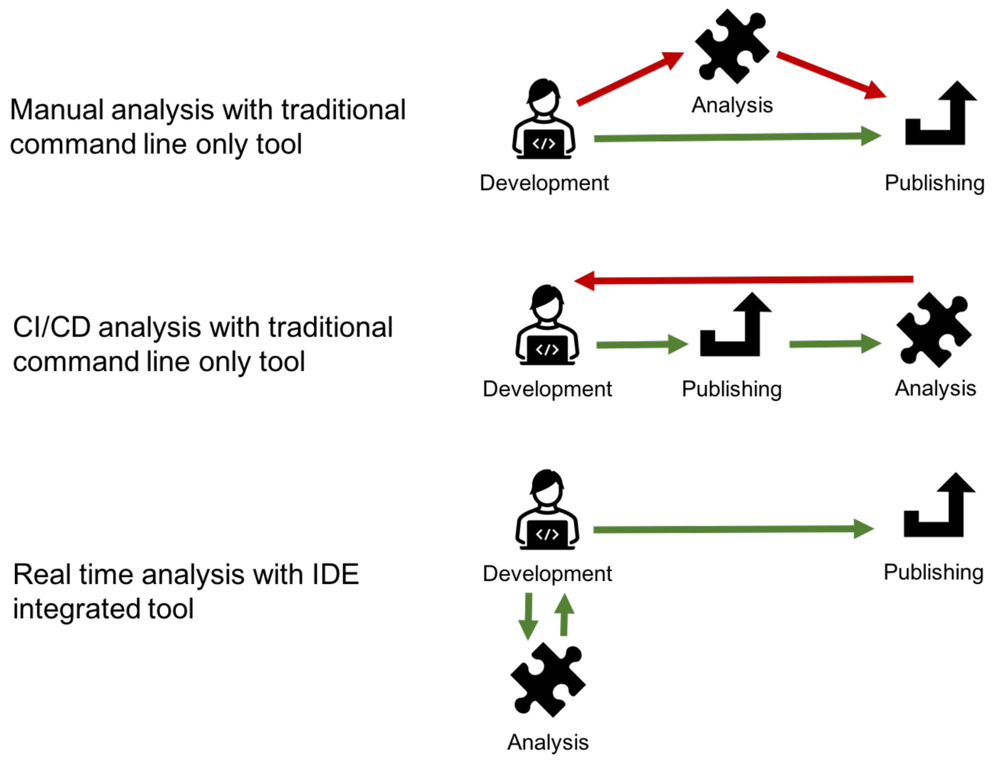

Manual analysis with traditional command line only tool
Analysis
Development
Publishing
CI/CD analysis with traditional command line only tool
Development
Publishing
Analysis
Real time analysis with IDE integrated tool
Development
Publishing
Analysis

Figure 6.2: Workflows for different analysis tools

6.2 Communication with the main core

As detailed previously in 4.3.4.2, the IDE communicates with the core of the linter through the LSP protocol. In practice, this means that they exchange JSON-RPC messages requesting and receiving information about the code to be analyzed and the detected vulnerabilities. The goal when choosing this kind of implementation was to take off as much logic as possible from the

IDE's responsibility, meaning that it acts as a thin client in the overall system architecture. The big advantage in this is that the same linter core (running as a local socket server) is compatible with any IDE prepared to deal with LSP messages. This can be easily achieved in a lot of cases by building a simple plugin, as detailed in 6.3.

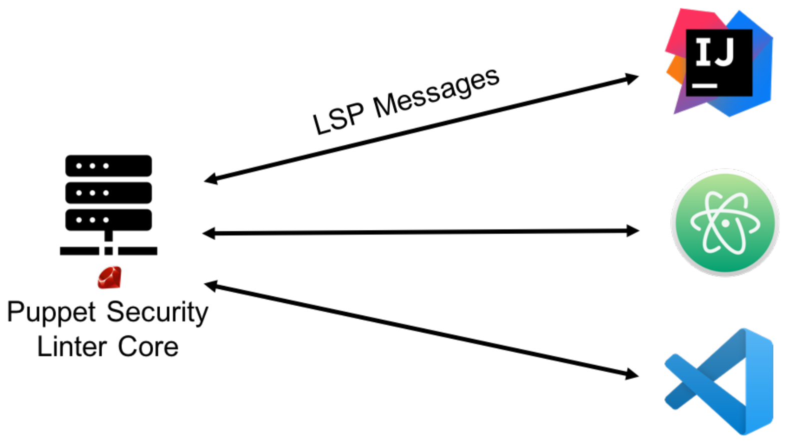


Figure 6.3: Core of the tool working with different IDEs from the market

This enables then a potential compatibility with most IDEs in the market, meaning that without a lot of effort the tool can reach to the vast majority of developers independently of the software they use. This is a crucial condition in the big goal of becoming the default security linter in the Puppet Community.

6.3 Basic requirements for an IDE extension

As explained in section 6.2, the architecture of the communication between the main core and IDE made it so the latter has the least amount of logic on its side. This was essential as there are many different IDEs available in the market and maintaining a plugin for each one of them is not an easy task.

Looking closely at exactly what is needed to integrate an IDE with this linter, it needs to be able to send and receive the JSON-RPC messages that are part of the Language Server Protocol. Several plugins like VSCode for example already offer integration out-of-the-box with this protocol, meaning that the plugin only needs to activate it when Puppet files are open. The IDE can automatically send the code for analysis to the core and receive/display the resulting vulnerabilities on the IDE's interface. Other IDE's might need an additional interpreter to be developed, able to do this task of sending and receiving LSP messages while the developer works on the code. Besides this, the plugin only needs to call the core server at startup, which is a simple command line tool. It runs in the background as a local socket server, receiving and sending messages to the IDE. No other functionality is required from the IDE plugin.

As such, this makes it fairly easy to develop plugins for new IDEs. As a bonus, changes can be performed to the linter core, such as adding and changing rules or configurations, while keeping full compatibility with the existing IDE plugins.

6.4 Implemented IDE: Visual Studio Code extension overview

As an implementation example for this thesis, it was decided to integrate the tool with Visual Studio Code, as it is a fairly easy IDE to work with (the author has already experience developing code for it) and also used by a considerable amount of developers, meaning that this initial plugin would already have a big impact. This integration, besides allowing for Visual Studio Code developers to use the new tool, has also the task of allowing for the practical demonstration inside this project of how the tool will work.

As mentioned previously in 6.3, Visual Studio code already has native support for the Language Server Protocol. As such, the plugin, developed in Typescript as it is standard across VS-Code extensions, only needs to activate this library (named Language Client) everytime a Puppet manifest is opened inside the IDE. This simplified development even more, as the logic of sending and receiving of the JSON-RPC messages was already done. Even the display of the warnings in the GUI is already taken care of. The only thing left to do was to activate the core of the linter, calling a simple command line process.

As mentioned in 5.2.2, the plugin was made available in the Visual Studio Code Marketplace.

Chapter 7

Tool's Validation: Results from User Study

7.1 Importance of conducting an User Study

As previously mentioned in section 3.2.2, one of the most important goals of this project was to significantly improve the user experience over the existing tools. This was demonstrated to be one of the main causes of complaints with the existing software and stated by developers as one of the main reasons why they avoid using static analysis tools.

As such, an User Study with actual developers from several backgrounds is the best way to evaluate how does the interface perform in a realistic scenario. The feedback from the participants corresponds to the real life experience of developers interested in adopting the software, providing an accurate evaluation of how well would the tool be received by the general public if released today.

7.1.1 Main goals

The main goal of the User Study was to provide an accurate and independent evaluation of the User Experience provided by the integration of the software with an IDE [1]. Important aspects like intuitiveness, workflow integration, and even the educational capability need to be tested by real developers who can give an independent opinion about the idea, implementation and possible improvements that could be made.

The integration with the developers' workflow, as mentioned previously in section 3.1.2, is critically important, as programmers will not stop their development process to run analysis tools. As shown, this often leads to the tools being seldom used, resulting in the accumulation of vulnerabilities that makes them much harder to be properly addressed, or even in the complete abandonment of the solution. One of the goals of this User Study is to understand how well does this tool fit into their workflows and if that would be enough to convince them to adopt it.

[1]Only Visual Studio Code was evaluated in this study, being the only IDE with an implemented plugin so far, but the basic User Experience is not expected to significantly change with other IDE's on the market.

It was also given a considerable amount of attention to the educational aspect of the software. As detailed in the section 4.4, the purpose of a software security tool should not only be in detecting problems but also teaching their users about the details and the importance of avoiding said problems in future occasions. That is arguably the best contribution given as it has the potential to considerably reduce the security vulnerabilities introduced in the future and the resulting software breaches and problems. To properly evaluate this, some of the participants chosen had less experience in the area and the goal of their contribution was to evaluate how well did the tool educate them on the vulnerabilities that they were previously unaware of.

7.2 Proposed structure

The User Study was conducted in remote interviews through video calls. The author interviewed all participants during these sessions and the feedback was recorded during the software utilization and in some questions done in the last part.

Most interviews started with the installation of the tool, being skipped if the candidate already had everything configured beforehand. After a small presentation talk and an introduction to the User Study itself, some Puppet Manifest Files with several security vulnerabilities were provided.

The main task consists in opening these files in the IDE and trying to find the vulnerabilities, learn about them and solve. At the same time, the interviewer took notes on the degree of difficulty felt by the participant.

Finished the search for all vulnerabilities, some questions were presented where the participant could rate their experience with the tool and also provide valuable feedback and suggestions for future improvement.

More details about the structure of the conducted User Study can be found on the original flyer distributed to all candidates of the study in appendix A.

7.3 Candidate Selection

The main candidates for the study were IaC developers who were looking for new tools to improve the quality and security of their code. Users of existing Static Analysis tools were without doubt the most fitted candidates for the job as their understand exactly what needed in a tool like this to be able to reach the market.

Besides them, students and more junior developers with a lower awareness for Cybersecurity were also very welcomed. Due to the importance of the educational potential detailed in section 4.4, another important goal of the study was to evaluate how well would the tool teach more naive users about how could they improve their code quality. As such, having these kinds of candidates allowed for a more complete evaluation and feedback of how well the tool was able to help them developed their skills further.

7.3.1 Participants' background

As mentioned before, the participants of the study have a fairly diverse background experience with IaC, Static Code Analysis Tools and programming in general.

Number of participants	Occupation	Years of experience	IaC Technologies
2	IT Professional	2	Puppet, Ansible, Terraform, Kubernettes
3	IT Professional	0	Ansible,Terraform
1	Student	0	-

Table 7.1: Study participants grouped into background experience groups

This made it possible to get different perspectives. The IT professionals with more years of experience had almost no problems in starting up the tools and doing the exercise. The vulnerabilities were already part of their day-to-day activities and their focus was completely directed at what the tool could do to improve their workflow and the results they obtained from it. Aspects like the user experience and customizability and how could they be improved in the future made the majority of the suggested improvements.

Professionals with less experience felt a little more difficulty in performing the task. Their experience definitely helped them understand how the linter worked, but some of the vulnerabilities were not immediately clear to them, forcing them to spend some time consulting the provided help pages detailed in section 4.4. This was of course essential to evaluate how well could the supplied materials help in educating users about the importance of solving these security vulnerabilities.

The students that participated still without any professional experience had to spend some time understanding what IaC was and also how to read and interpret the supplied Puppet Manifests. They also spent considerable amounts of time consulting the provided materials about the vulnerabilities, which was also essential to evaluate the educational potential of the software.

As shown then, this wide array of participants' backgrounds made the results of this study much more complete and allowed for the evaluation of how the tool could be received by different types of elements in the Puppet community, from absolute beginners to senior developers.

7.4 Results and User Feedback

The final User Study had the collaboration of 6 participants, ranging from professionals with some years of experience with Infrastructure as code to IT students still finishing up college with an interest in pursuing these kinds of technology. Each of them was subjected to the same online interview process described in the previous section 7.2. The result of that interview was a sheet of answers and feedback, which will now going to be compiled and analyzed in the study final results. The feedback obtained from participants will be the important building block for the future outlook for the tool described in section 7.5.

7.4.1 Vulnerability detection

As mentioned before, the main task of the User Study consisted in taking 5 Puppet manifests, opening them inside Visual Studio code with the Puppet Security Linter extension enabled, and try to detect and fix the vulnerabilities presented. The participant was also encouraged to use the configuration page when convenient.

The complete list of vulnerabilities is available in Appendix B, with also a small description and context of each one. Below are the results of the task for all participants. For each vulnerability, the type is presented alongside the percentage of participants who successfully identified the location of the vulnerability in the manifest and the percentage of participants who successfully solved it. The last column represents the distribution of the effort of everyone in solving the vulnerability, ranging from 1 (very easy - green) to 5 (very hard - red).

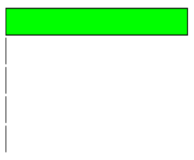
Vulnerability	**Type**	**Was identified?**	**Was solved?**	**Effort**
1.	HTTP without TLS	100%	100%	

Table 7.2: Analysis results for the first Puppet Manifest

In the first Manifest, only one vulnerability was present. This *HTTPS without TLS* vulnerability was very easily identified and solved by all participants, as it only consisted in switching the link protocol from HTTP to HTTPS.

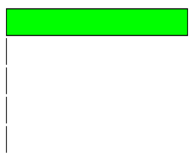
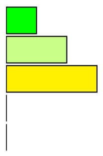
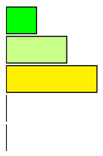
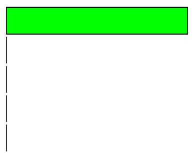
Vulnerability	Type	Was identified?	Was solved?	Effort
2.	HTTP without TLS	100%	100%	
3.	Hard-coded credentials	100%	100%	
4.	Admin by default	100%	100%	
5.	Hard-coded credentials	100%	100%	

Table 7.3: Analysis results for the second Puppet Manifest

On the second manifest, some participants had some trouble in removing the Hard-coded credentials from the manifest, as working those without any experience working with Puppet had to learn what Hierra is and how it works. In this study, only the function call was required to be written, meaning that participants did not need to build the actual secret file.

In general, the provided help page was effective in teaching how to use Hierra to remove the credentials from the file, meaning that the only effort was really opening and reading this help.

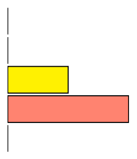
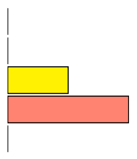
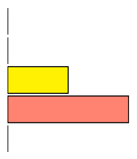
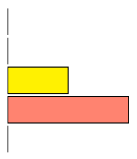
Vulnerability	Type	Was identified?	Was solved?	Effort
6.	HTTP without TLS	100%	0%	
7.	HTTP without TLS	100%	0%	
8.	HTTP without TLS	100%	0%	
9.	HTTP without TLS	100%	0%	

Table 7.4: Analysis results for the third Puppet Manifest

In the third manifest, things were much more complicated. The presented scenario was the use of HTTP inside a resource that downloaded and installed a package form an APT repository. It also validated the package itself with a GPG key. As presented in a previous study [17], in a lot of real life scenarios, these packages are retrieved from an unsecured HTTP connection precisely because the use of an GPG key guarantees the integrity of the received package.

On the previous SLIC tool, this caused a false positive because a warning was produced, but the developer had a good reason to keep it as it is. This scenario was included in the study because one of the main features of this new tool is the possibility of customizing rules and this this particular case, the participant of the study was incentivized to open the Configurations' Page and add this particular resource to the list of resources where the use of HTTP is allowed. This way, the warning was suppressed and the developer was free to keep using HTTP without a false positive.

Most participants did not know about this and instinctively tried to change the protocol from HTTP to HTTPS. This forced the intervention of the interviewer and after the explanation, most of them immediately recognized the need to add an exception to the rule there. This explains the higher effort.

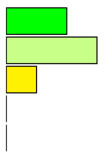
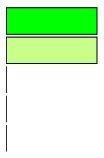
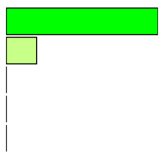
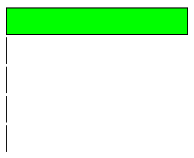
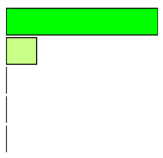
Vulnerability	Type	Was identified?	Was solved?	Effort
10.	Cyrillic Homograph Attack	100%	100%	
11.	Suspicious comment	100%	100%	
12.	Invalid IP address binding	100%	84%	
13.	Hard-coded credentials	100%	100%	
14.	Empty password	100%	100%	

Table 7.5: Analysis results for the fourth Puppet Manifest

In the fourth Puppet Manifest, the main difficulty was in understanding what an Homograph attack was, as it seemed that most participants did not know the existence of this kind of vulnerability. But after consulting the included help page, all participants understood that characters of different alphabets could have the same appearance and be used as a way of fooling the user to connect to a malicious address. As such, re-writing the address was the obvious choice to fix it.

The higher effort in solving the Suspicious comment is related with the wildly different approaches that all participants used. While some simply erased the comments, others tried to remove the suspicious words that triggered the tool, trying this way to "circumvent" the tool. One participant argued even that the suspicious comment could be left and as such, the rule could be disabled there. This caused some discussion and a little more effort in understanding exactly what vulnerability was described in the help page.

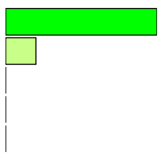
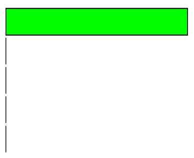
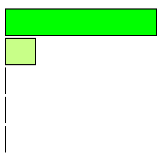
Vulnerability	Type	Was identified?	Was solved?	Effort
15.	Weak crypto algorithm	100%	100%	
16.	Hard-coded credentials	100%	100%	
17.	Admin by default	100%	100%	

Table 7.6: Analysis results for the fifth Puppet Manifest

Finally, in the last Puppet Manifest, there were no relevant Vulnerabilities that required an higher effort to resolve. Removing Hard-Coded credentials was very easy here as participants already learned about Hierra in a previous Manifest, thus demonstrating the effectiveness of the provided help pages in educating users.

7.4.2 Follow-up questions

After the vulnerability detection was completed, some questions were then made to participants so they could formally evaluate the tool, the study, and the feedback they got from this process.

All questions required a rating of 1 (very negative) to 5 (very positive). This allowed for the uniformization of all answers, making it possible to establish comparisons between different participants, which was practically impossible if only open answers were accepted. The answer distribution of all questions are presented below, divided up into three categories: *Pre-task* questions that are about the experience of using the tool before the study; *Task* questions about the performed vulnerability detection itself; and finally *Pos-Task* questions about the general impression obtained by the users after the completion of the task and feedback for the future.

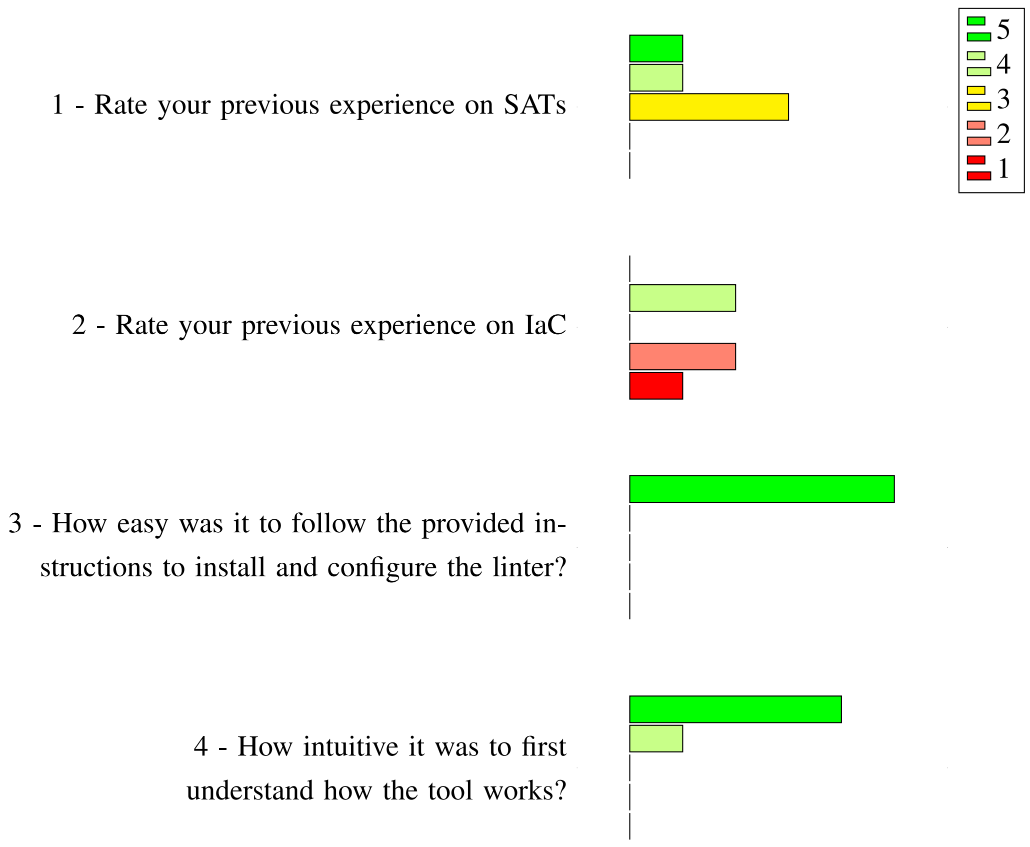


Figure 7.1: Pre-Task questions results

Most participants reveal here some experience with Static Analysis tools, although most admit that it is pretty basic and does not evolve more advanced tasks like deep customization or the development of new rules.

Experience with Infrastructure-as-Code was pretty poor across the board. Except for some of the most experienced professionals which work with tools like Ansible or Puppet, most participants reveal complete inexperience with this kind of software. This naturally explains some of the difficulties felt regarding interpreting the Puppet Manifest and also identifying some of the more complex vulnerabilities.

In a positive note, both the installation and first impressions of the Puppet Security Linter were considered very positive by nearly all participants, demonstrating the results of the work in simplifying these processes and integrating with Visual Studio Code warning system. On the negative side, some participants reported some difficulty in running the tool on Windows[2] and also in accessing the tool configurations' page[3].

[2]In one of the interviews where the participant was using the Microsoft Windows operating system, it was revealed that Visual Studio Code was unable to launch the core of the linter with the Ruby Windows installation, thus preventing the tool from working. This was since then mitigated by releasing a patch.

[3]Having to open a terminal window to run a command was considered by some participants as not very intuitive and a way of launching the configurations' page from the IDE was since incorporated in a later version.

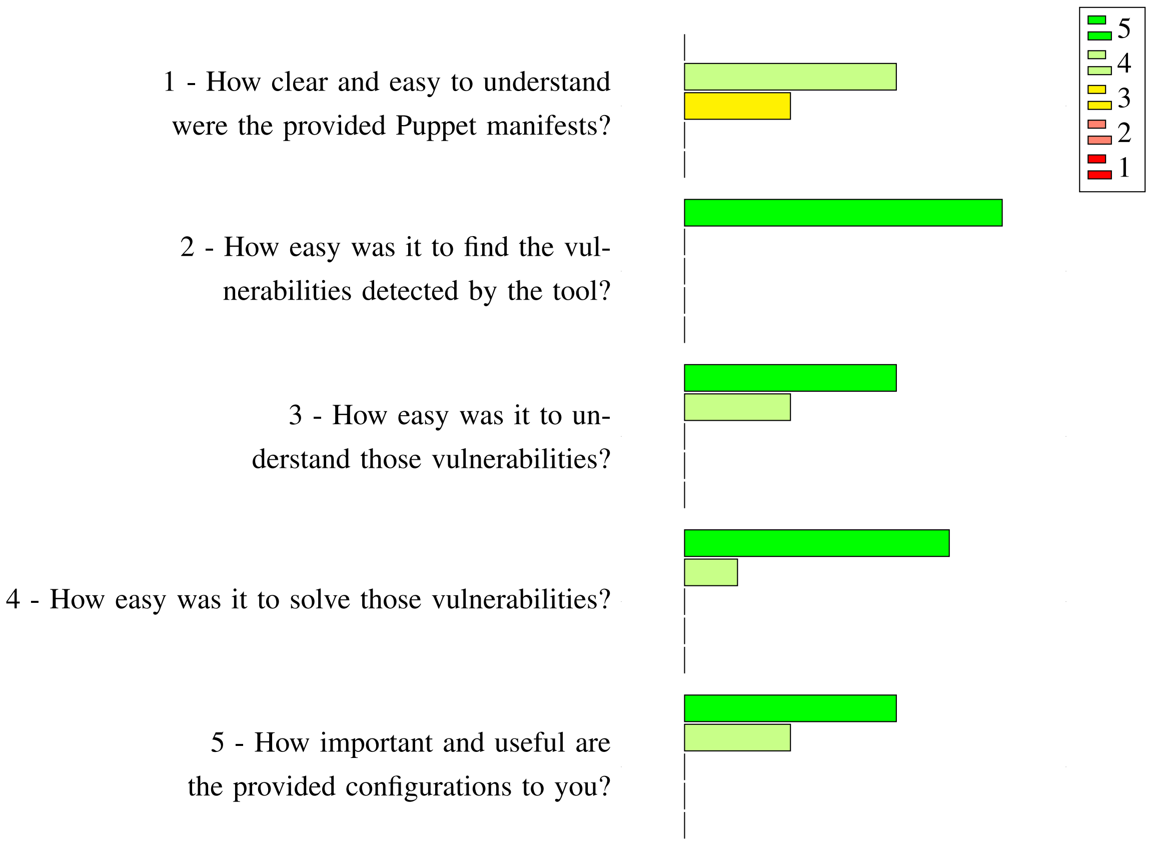


Figure 7.2: Task questions results

During the task, some participants had some trouble in understanding the syntax of the Puppet manifests, mainly due the previously mentioned lack of experience with this kind of IaC technologies. This required some effort form the interviewer to help them interpret the code and have the proper context for the vulnerabilities.

Finding and solving the vulnerabilities was universally considered very trivial by the fact that the warnings are displayed right in the code itself. The developer only needs to rewrite the selected code to solve the problem. This is precisely the result that motivated the integration of the tool with IDE's, mentioned in section 3.2.2. Participants gave very positive feedback on this integration and how it could really fit into their development workflows.

Understating vulnerabilities, in the opinion of participants, was made easy by the existence of the help pages, described in section 4.4. These provided critical insight into why the vulnerabilities are a problem and on how to actually solve them in the code.

Finally, the configurations' page, detailed previously in section 4.3.3.1, revealed to be a very useful resource in the performed task. Participants were generally impressed by the degree of customization offered, meaning that they could adapt the tool to the different projects they could develop. By establishing parallels with previous experiences with others SAT's, they could see the usefulness in having this kind of customization available to prevent false positives and a generally better experience. The main complaints revolved around the poor design and intuitiveness of the page, which is still very basic in terms of appearance.

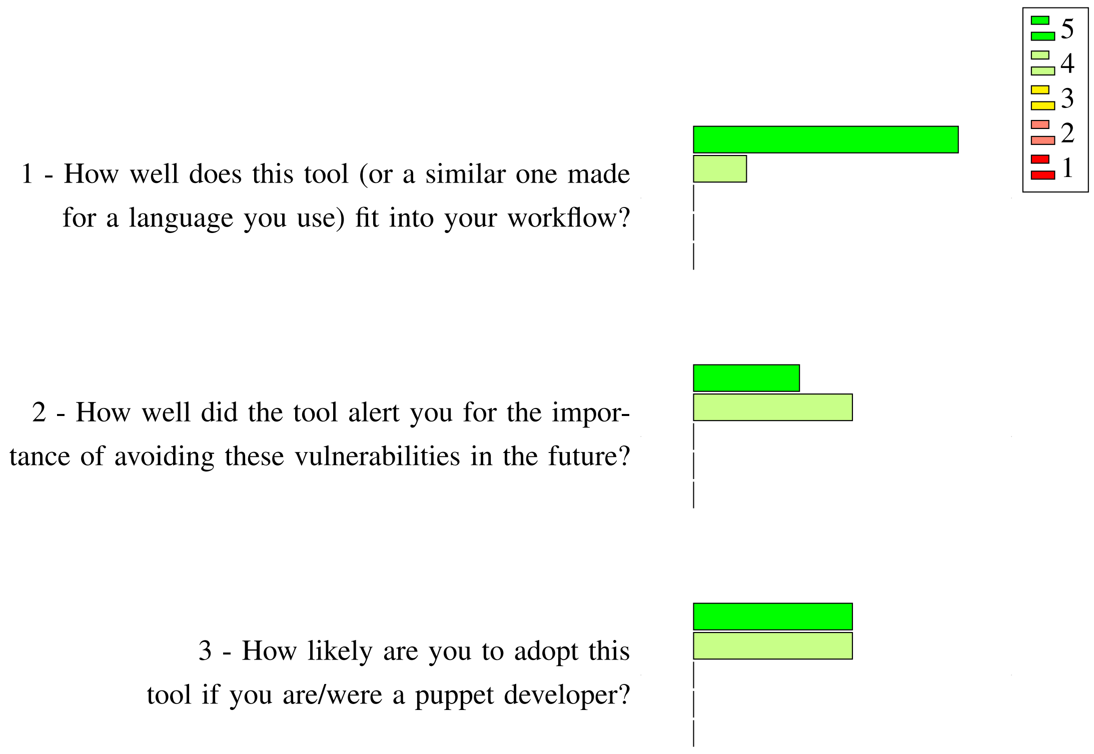


Figure 7.3: Pos-Task questions results

Finally, the last questions about the impressions after the realization of the task revealed that this tool fits very well into their workflows. The integration with the IDE ensures the minimal disruption into their development process, combined also with the good performance of the tool, capable of performing an almost real-time analysis. The only complaint was related to how the configuration's page is accessed through the terminal, which takes a little more time.

The educational purpose of the tool, as mentioned in section 4.4, was definitely recognized by the participants, who mentioned the importance of having this knowledge built into a tool like this. They agreed that this should be one of the main purposes of this kind of software. Someone also mentioned that with the further development of the tool and participation of more developers, the knowledge present in these pages could be very easily expanded, improving even more this educational potential.

Lastly, most participants agree that they would adopt this tool if they were Puppet developers, recognizing its usefulness in preventing costly security breaches and its easy of use. They also mentioned that this could clearly take some time as the tool still needs some more polishing and also a bigger and more active development community in the near future.

7.5 Looking into future improvements

As mentioned in the previous section, participants gave several improvement suggestions to the tool during the task and also at the end, during the feedback stage. Some of them were already mentioned in the analysis of the answers given, like the improvement of the configurations' page. There are still some important improvements to be done and some sections that need to be polished. Firstly, the access to the page itself is still considered very primitive. The page can be accessed by running the tool in a Terminal with a configurations flag, which breaks the immersion provided by integration with the IDE. With a later patch, it is now possible to click on a link inside the output window of Visual Studio Code, but this can still be improved and even integrated inside the IDE. Besides that, the page itself still has some rough aspects, like the design which lacks any CSS and the submit button, which could be placed in a more visible place floating at the bottom of the screen. Some configurations could also be more clear in their descriptions. Lastly, while this should not be the perfect behavior, in a lot of situations the participants where forced to restart Visual Studio Code to apply their configurations. This of course is not convenient and causes some disruption to a developer workflow. As such, it should be addressed in the future ensuring that changes are actually applied in real-time.

About the Vulnerabilities' explanation pages, although the feedback was generally positive, most participants agree that accessing them is currently a bit difficult. This is done by a hyperlink present in the warning message generated inside VSCode. This is the default behavior for most VSCode linters and the page can be accessed by Ctrl+clicking on the link. Admittedly, this is not very intuitive for users who are not used with VSCode and it is something that could definitely be addressed not only in this one but also in all future supported IDEs.

Regarding the rules, some suggestions were made like expanding the Homograph attack to include more alphabets besides Cyrillic, like the Greek alphabet which also has some problematic characters. Moreover, the suggestions were made to implement some kind of tag that could be written on the manifest to prevent a specific line form being analyzed by the linter. The IP binding tool could also recognize subnets besides individual IP addresses.

Finally, one user demonstrated a great interest in the fact that more rules and configurations could be easily added to the tool. To also improve on this feature, he suggested that in the future, another User Study similar to this one could be conducted, but this time focusing on developers trying to add new rules and configurations to the linter and how easily that could be achieved.

To conclude this section, all participants gave a general good feedback about the tool and the User Study conducted. They also took the time to give valuable feedback on what they think that could help the software to achieve even more success in the community. This feedback will definitely prove very valuable in the future development of the tool following the conclusion of this thesis.

Chapter 8

Conclusions and future work

Infrastructure-as-code is becoming much more relevant in a world increasingly connected to the cloud. As the complexity of IT infrastructures grows, the role of automation becomes much more prevalent to avoid ever increasing teams of expensive professionals and to reduce the prevalence of human errors and inconsistencies.

Security is vital to any kind of IT system and with the new paradigm of IaC comes along a completely new set of security vulnerabilities and challenges that need to be addressed. Automated static code analysis tools play an important role in dealing with these by providing developers with an important feedback on the quality of their code and giving companies an additional insurance that their code is secure.

Focusing only on the *Puppet* language, there is a considerable lack of options for developers who are looking for security linters. To address this, *Rahman et al.* built the SLIC tool, which promised to be the new standard tool for developers who want to ensure the security of their code. Although the author provided very impressive results in terms of precision of his tool, later independent studies revealed its countless flaws that caused an incredibly high number of false positives.

With this knowledge about the current state of IaC and the existing SLIC tool, a new tool was developed with a focus on increased precision and a better User Experience. This software, developed in Ruby, was designed from the ground up with the intention of being expanded in the future. By using a modular architecture, new rules and configurations can be added in the future with little to no effort. This will make it possible for new developers to integrate complex analysis technologies like Machine Learning or Natural Language Processing, enabling the tool to reach even higher levels of precision. The designed architecture enables the integration with virtually any IDE in the market. This is possible due to the centralization of all analysis logic into a central core that runs in a web socket server. By using the universal LSP protocol, developed by Microsoft, it can communicate with any supporting IDE. Plugins can be developed to aid in this process. This enables for a much better User Experience than the previous solutions on the market, allowing developers to detect and fix vulnerabilities easily without ever disrupting their workflow. This was also proved by the participants in the realized User Study, which independently evaluated how good and intuitive the usability of the tool really was. The possibility to customize the analysis rules was also one feature that developers considered in previous studies to be essential to any Static Analysis tool. As such, it was incorporated here, allowing for the full configuration of the behavior of the rules, although future improvements were requested during the User Study.

This Modular Architecture allows also for the potential adaptation of the tool to work with other IaC languages, like Ansible. By adapting the Tokenizer, the rest of the architecture can work without problems with the new language.

Another critical aspect addresses during the development process was the educational purpose of an analysis tool like this. Developers who still are not familiarized with security vulnerabilities need to be educated about them, as this is the only way that they will be able to actually address them. The final tool incorporates several tools, like help pages, to achieve this goal and the participants in the User Study agree. This is considered the most noble contribution to the Puppet community by the software.

To validate the work done and the usability of the software, a User Study was conducted, counting with the participation of not only professional developers in the IaC area, but also junior and student developers who are just now giving their first steps in this world. The individual interviews realized and the several questions answered allowed them to express their generally positive feedback of the developed tool. Both the intuitiveness of the IDE integration and the high degree of customization were praised by developers, although some aspects revealed their need to still be further polished. Arguably the most important result form this study was the long and insightful list of improvements and features suggested by the participants, which can definitely help the tool to achieve even better results and cement itself as the go-to solution in the Puppet community.

Gathering everything achieved in the Thesis, it is possible to affirm the the main objective of developing an improved Security Analysis tool for Puppet which could integrate with IDEs on the market was achieved with success. Not only the tool was developed, but was also subjected to an independent analysis via an User Study. All of this demonstrates its high potential and justifies the continuation of its development in the future after this project. With more time and a bigger development team, it can definitely become the standard in the Puppet community, which is the ultimate goal that inspired the entire project.

Appendix A

User Study Flyer

This flyer was distributed to all candidates who were interested in participating in the User Study. It explains what the study is about, what Infrastructure as Code is and exactly where does the evaluated tool fits in this scenario. Some information about the types of participants who were wanted is also present in the document.

It also contains instruction on how to install the software so that they could also try it out and get ready before the interview.

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

TQRG

Puppet Security Linter

for Visual Studio Code

User Study

Puppet Security Linter is a tool that analyses Puppet code in real-time, looking for security vulnerabilities that might compromise the deployed infrastructure in the future.

It works as a standalone console application and also as a Visual Studio Code extension, providing suggestions right in your code.

Part of a Master Thesis

The goal of this user study is to evaluate the User Experience of the Puppet Security Linter tool, developed as part of a Thesis for the Masters in Informatics and Computing Engineering course at FEUP.

The results will help to validate and improve the tool for a potential release to the public.

You will evaluate

- Ease of installation
- Intuitiveness of the UI
- Customizability
- Precision of detection

Any doubt?

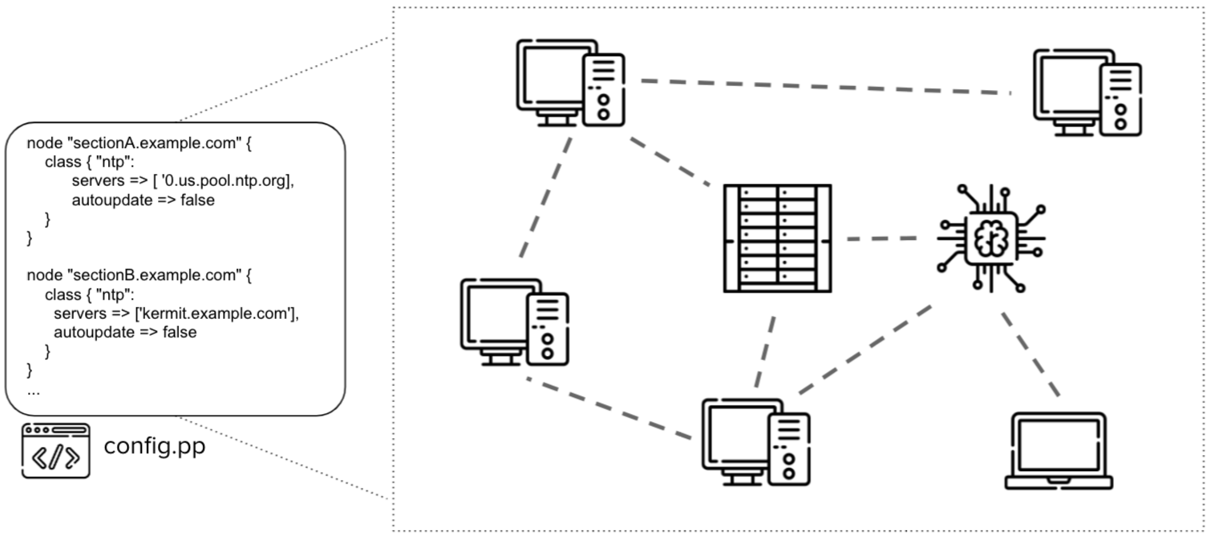
Tiago Ribeiro

up201605619@up.pt

30 ~ 45 mins

Some background info

Puppet is an infrastructure-as-code language, where IT infrastructure configuration and deployment is written in a programming language, allowing for the automatic and consistent deployment of applications and services.



```
node "sectionA.example.com" {
   class { "ntp":
        servers => [ '0.us.pool.ntp.org],
        autoupdate => false
   }
}

node "sectionB.example.com" {
   class { "ntp":
     servers => ['kermit.example.com'],
     autoupdate => false
   }
}
...
```

The entire infrastructure for an application can now be described in a script file (called manifest), making it possible to version and manage all environment configurations just like the rest of the code. But in the same way that a development team has a great view over all the configurations of a software environment, a malicious attacker can also have an easier time spotting server misconfigurations and vulnerabilities left unchecked that he might take advantage of to compromise the system.

As such, the need arises for a tool capable of spotting these potential vulnerabilities and then warning the developer about them, as soon as they are written. The Puppet Security Linter promises exactly that, displaying the vulnerabilities right inside the IDE, without disrupting the developer's usual workflow.

Who are we looking for?

We are looking for developers who are interested in using static analysis tools to improve the quality of their work or that already use them regularly. Students and junior developers motivated to start having more experience in the CyberSecurity world are also very welcomed!

Selection Criteria

- ✓ Developer or Student (experience in Puppet, IAC or scripting languages is a plus)

- ✓ Interested in contributing to build an accurate tool able to help them better secure their code.

- ✓ Knowledge about Static Analysis Tools (regular user or having past experiences, either positive or negative) is a good to have.

- ✓ Experience in software security is optional, as one of the goals of the study is to assess how well does the tool educate about good security principles.

How will the study be conducted?

At the end of this paper, we have available quick instructions on how to set up and use the software. You can start by downloading and then getting used to it.

We will schedule a call (maximum 45 minutes) to conduct the actual study and receive your valuable feedback.

The call will consist of two parts:

1. Hands-on experience

We will provide, in the beginning, a folder of simple Puppet Manifests. Some will be okay but others will contain a variety of security vulnerabilities.

You will use the linter to help you locate, identify and solve those vulnerabilities so that you can secure them and avoid malicious attackers who could exploit them in the future.

Don't worry if you have any doubt or problem. We will be present and help you with any technical issue you might encounter. We want to evaluate how well the tool works for you in a realistic scenario.

We are expecting this part to last from 20 to 30 minutes.

2. Feedback interview

After you get the chance to test the tool in a "real-world" scenario, we are interested to know everything you might have to say about it (good aspects, bad experiences, non-intuitive interfaces, etc.).

We will present a shortlist of questions where you will rate your experience on a scale of 1 to 5 regarding the pre-task (installing and using the tool before the call), task (the test you just performed on-call) and pos-task (what impression did you get from the tool for the future).

Afterward, we can also have a more informal talk about your feedback, suggestions and general impressions you got from the tool, according also to your experience and background.

We are eager to know what you will have to say!

Installing Puppet Security Linter

If you are getting ready for the call or just curious about the tool, here are the instructions on how to install and use it.

Installation

Prerequisites:

- ✓ Have Visual Studio Code installed;
- ✓ Have Ruby installed (version 2.3 and up);
- ✓ Have RubyGems installed;

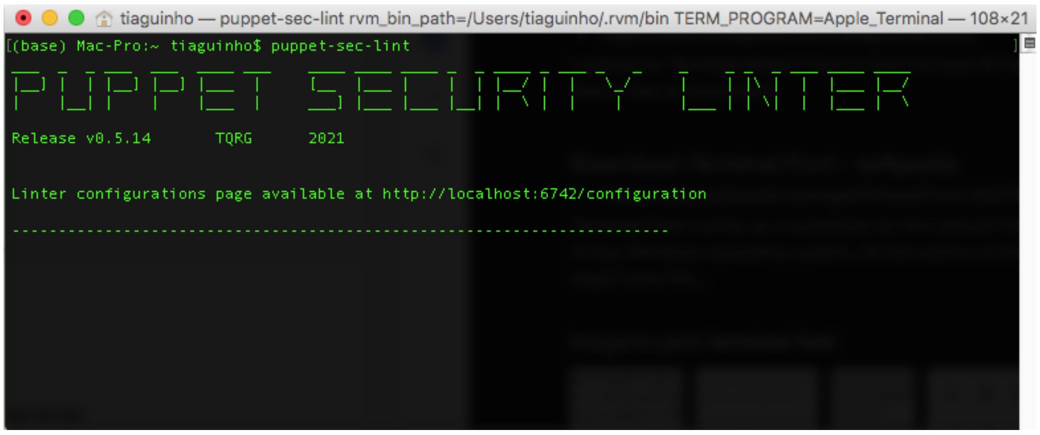
Firstly, install the Puppet Security Linter Ruby gem, responsible for all the analysis logic, by running the following command in a terminal window:

```
gem install puppet-sec-lint
```

To test if the gem is installed and working, run the following command on your terminal:

```
puppet-sec-lint
```

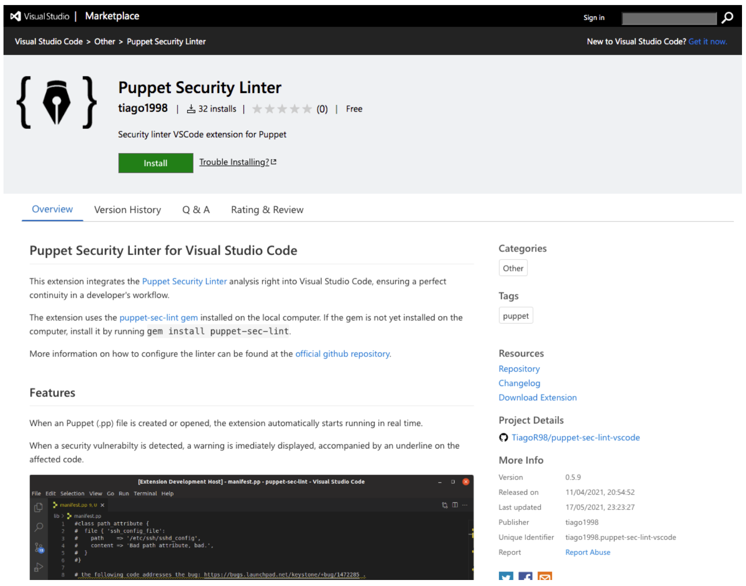
You should be able to see the following output:



```
tiaguinho — puppet-sec-lint rvm_bin_path=/Users/tiaguinho/.rvm/bin TERM_PROGRAM=Apple_Terminal — 108×21
[(base) Mac-Pro:~ tiaguinho$ puppet-sec-lint

Release v0.5.14        TQRG       2021

Linter configurations page available at http://localhost:6742/configuration

-------------------------------------------------------------------------
```

If everything is working correctly so far, it's then time to install the Visual Studio Code extension, so that we can detect the vulnerabilities in real-time, right inside the IDE (more IDEs will be supported in the future).

Install it from the marketplace or search for the extension "Puppet Security Linter" inside Visual Studio Code.



Visual Studio | Marketplace

Visual Studio Code > Other > Puppet Security Linter

New to Visual Studio Code? Get it now.

Puppet Security Linter

tiago1998 | 32 installs | (0) | Free

Security linter VSCode extension for Puppet

Install Trouble Installing?

Overview Version History Q & A Rating & Review

Puppet Security Linter for Visual Studio Code

This extension integrates the Puppet Security Linter analysis right into Visual Studio Code, ensuring a perfect continuity in a developer's workflow.

The extension uses the puppet-sec-lint gem installed on the local computer. If the gem is not yet installed on the computer, install it by running `gem install puppet-sec-lint`.

More information on how to configure the linter can be found at the official github repository.

Features

When an Puppet (.pp) file is created or opened, the extension automatically starts running in real time.

When a security vulnerabilty is detected, a warning is imediately displayed, accompanied by an underline on the affected code.

```
[Extension Development Host] - manifest.pp - puppet-sec-lint - Visual Studio Code
#class path_attribute {
#  file { 'ssh_config_file':
#    path    => '/etc/ssh/sshd_config',
#    content => 'Bad path attribute, bad.',
#  }
#}

# the following code addresses the bug: https://bugs.launchpad.net/keystone/+bug/1472285 .
```

Categories: Other

Tags: puppet

Resources: Repository, Changelog, Download Extension

Project Details: TiagoR98/puppet-sec-lint-vscode

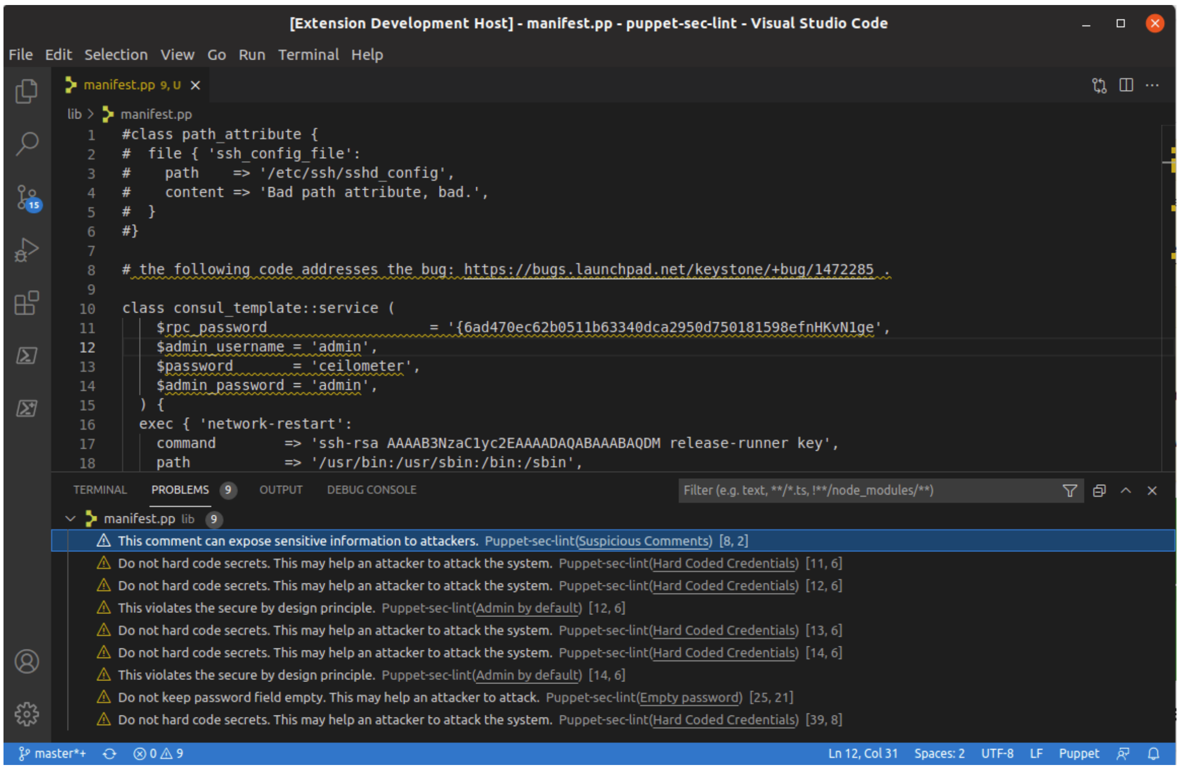
More Info

Version	0.5.9
Released on	11/04/2021, 20:54:52
Last updated	17/05/2021, 23:23:27
Publisher	tiago1998
Unique Identifier	tiago1998.puppet-sec-lint-vscode
Report	Report Abuse

Usage

After the extension is installed, it should be activated automatically every time a Puppet file (.pp extension) is opened.

As an example, a simple Puppet manifest was opened:



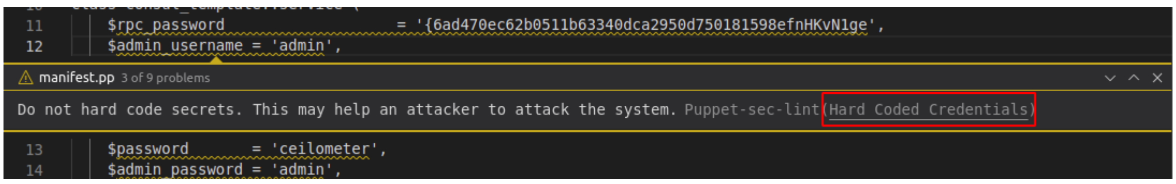
[Extension Development Host] - manifest.pp - puppet-sec-lint - Visual Studio Code
File Edit Selection View Go Run Terminal Help
manifest.pp 9, U
lib > manifest.pp

```
 1  #class path_attribute {
 2  #  file { 'ssh_config_file':
 3  #    path    => '/etc/ssh/sshd_config',
 4  #    content => 'Bad path attribute, bad.',
 5  #  }
 6  #}
 7
 8  # the following code addresses the bug: https://bugs.launchpad.net/keystone/+bug/1472285 .
 9
10  class consul_template::service (
11      $rpc_password                   = '{6ad470ec62b0511b63340dca2950d750181598efnHKvN1ge',
12      $admin_username = 'admin',
13      $password       = 'ceilometer',
14      $admin_password = 'admin',
15    ) {
16    exec { 'network-restart':
17      command         => 'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDM release-runner key',
18      path            => '/usr/bin:/usr/sbin:/bin:/sbin',
```

TERMINAL PROBLEMS 9 OUTPUT DEBUG CONSOLE
Filter (e.g. text, **/*.ts, !**/node_modules/**)
manifest.pp lib 9

```
⚠ This comment can expose sensitive information to attackers. Puppet-sec-lint(Suspicious Comments) [8, 2]
⚠ Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials) [11, 6]
⚠ Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials) [12, 6]
⚠ This violates the secure by design principle. Puppet-sec-lint(Admin by default) [12, 6]
⚠ Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials) [13, 6]
⚠ Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials) [14, 6]
⚠ This violates the secure by design principle. Puppet-sec-lint(Admin by default) [14, 6]
⚠ Do not keep password field empty. This may help an attacker to attack. Puppet-sec-lint(Empty password) [25, 21]
⚠ Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials) [39, 8]
```

All the vulnerabilities are immediately detected and displayed as regular Visual Studio Code warnings, underlining the relevant code.

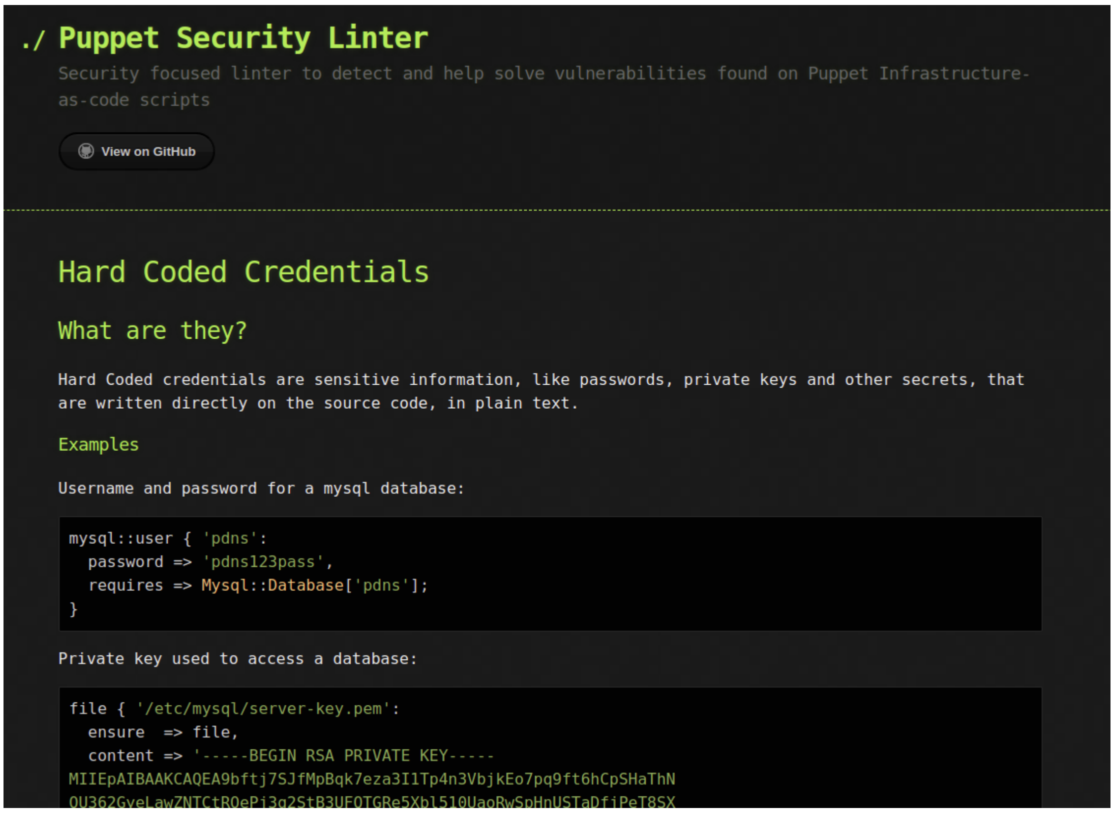


```
11      $rpc_password                   = '{6ad470ec62b0511b63340dca2950d750181598efnHKvN1ge',
12      $admin_username = 'admin',
```

```
Do not hard code secrets. This may help an attacker to attack the system. Puppet-sec-lint(Hard Coded Credentials)
```

```
13      $password       = 'ceilometer',
14      $admin_password = 'admin',
```

Each contains a small description of the problem and a link to a webpage where more information can be found about the vulnerability, how an attacker could potentially exploit it and suggestions on how to solve it.



./ Puppet Security Linter

Security focused linter to detect and help solve vulnerabilities found on Puppet Infrastructure-as-code scripts

Hard Coded Credentials

What are they?

Hard Coded credentials are sensitive information, like passwords, private keys and other secrets, that are written directly on the source code, in plain text.

Examples

Username and password for a mysql database:

```
mysql::user { 'pdns':
  password => 'pdns123pass',
  requires => Mysql::Database['pdns'];
}
```

Private key used to access a database:

```
file { '/etc/mysql/server-key.pem':
  ensure  => file,
  content => '-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA9bftj7SJfMpBqk7eza3I1Tp4n3VbjkEo7pq9ft6hCpSHaThN
OU362GveLawZNTCtROePj3q2StB3UFOTGRe5Xbl510UaoRwSpHnUSTaDfiPeT8SX
```

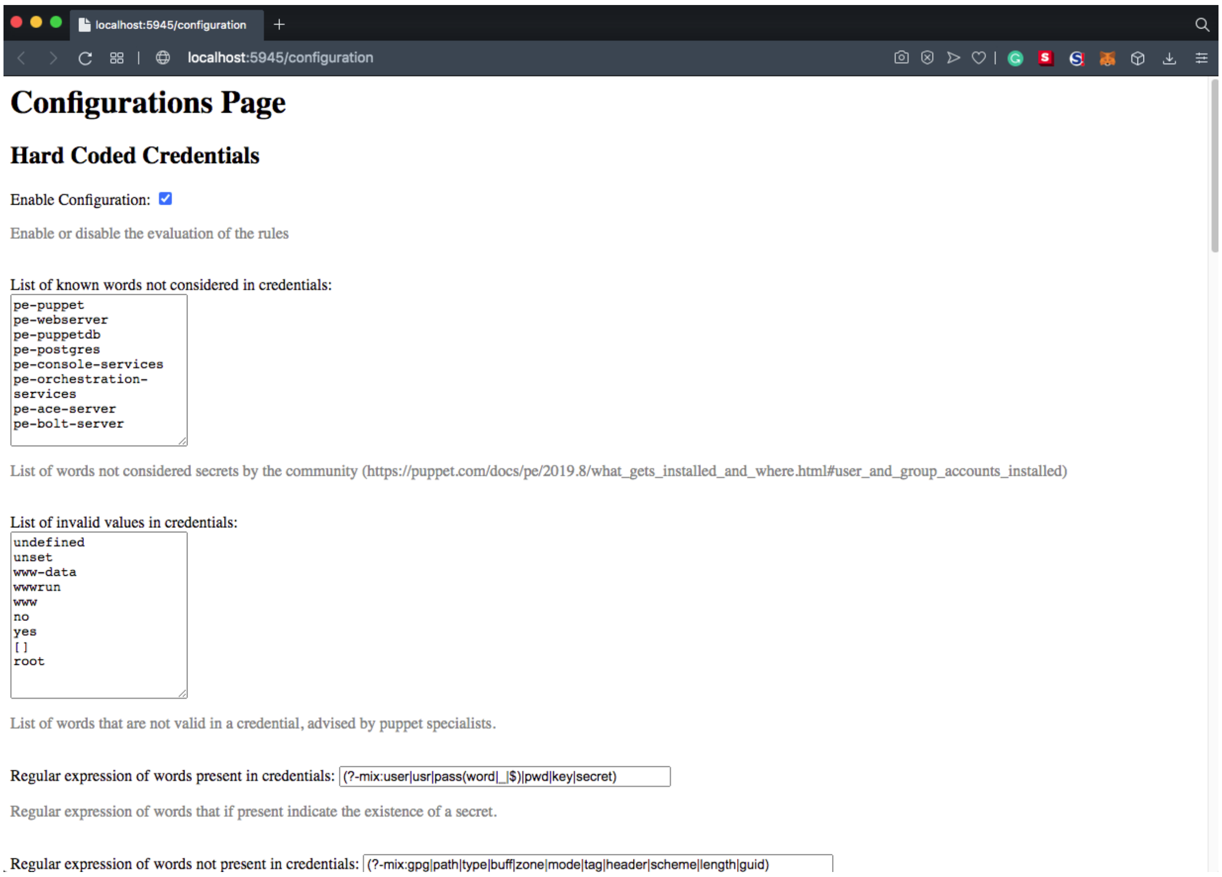
Customization

As every developer is different and different projects follow different conventions, a crucial part of any linter is the degree of customization of the rules applied. This ensures that the tool can be tuned up to any specific aspect of a software project and avoid unnecessary false positives.

The Puppet Security Linter was developed from the ground up with this in mind, allowing the user to customize several parameters of all the rules applied. If you need to configure how the linter detects the vulnerabilities, access the configurations page by running the following command in a terminal:

```
puppet-sec-lint --configuration
```

A window of your default web browser should open immediately with all the configurations available.



Configurations Page

Hard Coded Credentials

Enable Configuration: ☑

Enable or disable the evaluation of the rules

List of known words not considered in credentials:

```
pe-puppet
pe-webserver
pe-puppetdb
pe-postgres
pe-console-services
pe-orchestration-
services
pe-ace-server
pe-bolt-server
```

List of words not considered secrets by the community (https://puppet.com/docs/pe/2019.8/what_gets_installed_and_where.html#user_and_group_accounts_installed)

List of invalid values in credentials:

```
undefined
unset
www-data
wwwrun
www
no
yes
[]
root
```

List of words that are not valid in a credential, advised by puppet specialists.

Regular expression of words present in credentials: `(?-mix:user|usr|pass(word|_|$)|pwd|key|secret)`

Regular expression of words that if present indicate the existence of a secret.

Regular expression of words not present in credentials: `(?-mix:gpg|path|type|buff|zone|mode|tag|header|scheme|length|guid)`

Configure each vulnerability detection rule to your preferences and then click “Submit” to save your settings. All changes should be applied immediately.

Final Remarks

And that’s everything needed to perform the user study. Any questions can be answered by contacting me at up201605619@up.pt.Any questions can be answered by contacting me at up201605619@up.pt.

Appendix B

User Study Guide

This is the guide that supported the User Study interviews. For each participant, the information and results were filled in during the interview process.

The data collected ranged from some personal information and the background of the professional, which is important to contextualize the results of the interview, to the difficulty to address each vulnerability and finally, the feedback obtained from the experience and suggestions to improve the software.

The last section was used to fill in any additional information or feedback that came up during the call and that did not fit any of the previous sections.

Puppet Security Linter User Study Guide

Participant information

Name:

Background experience (years):

Current activity:

Experience and expectations about IaC, SAT’s and Puppet-Sec-Lint:

Interview Vulnerabilities Cheat Sheet:

Vulnerability			Information
File	Location	Type	
example1.pp	**Line:** 5 **Column:** 22	HTTP without TLS	This is a basic http vulnerability where the .jar file could be downloaded from an https address, ensuring its authenticity.
example2.pp	**Line:** 2 **Column:** 18	HTTP without TLS	This is a basic http vulnerability where the .jar file could be downloaded from an https address, ensuring its authenticity.
example2.pp	**Line:** 21 **Column:** 2	Hard-coded credentials	The username is hardcoded into the puppet manifest.
example2.pp	**Line:** 21 **Column:** 2	Admin by default	The username is an administrator of the node being created, having a high probability of having more privileges than needed, violating the “Least Privilege Principle”.
example2.pp	**Line:** 22 **Column:** 2	Hard-coded credentials	The password is hard-coded into the puppet manifest
example3.pp	**Line:** 4 **Column:** 20	HTTP without TLS	This is not actually a vulnerability, as the package authenticity is ensured by a gpg key. The user should be incentivized to use the tool configurations to add an exception here.
example3.pp	**Line:** 7 **Column:** 20	HTTP without TLS	This is not actually a vulnerability, as the gpg key authenticity is ensured by its signature. The user should be incentivized to use the tool configurations to add an exception here.

example3.pp	**Line:** 12 **Column:** 22	HTTP without TLS	This is not actually a vulnerability, as the package authenticity is ensured by a gpg key. The user should be incentivized to use the tool configurations to add an exception here.
example3.pp	**Line:** 15 **Column:** 22	HTTP without TLS	This is not actually a vulnerability, as the gpg key authenticity is ensured by its signature. The user should be incentivized to use the tool configurations to add an exception here.
example4.pp	**Line:** 59 **Column:** 18	Cyrillic HomographAttack	The url contains the letter 'a' from the Cyrillic alphabet which looks identical to the one in latin. This is a strong indication of a malicious website distributing a malware infected version of the 'yarn' package.
example4.pp	**Line:** 87 **Column:** 4	Suspicious comment	This is a suspicious comment as it indicates that the developer is not that confident about the solution used to fix the permission of the file.
example4.pp	**Line:** 102 **Column:** 5	Invalid IP address binding	This mail server has the 0.0.0.0 ip address binding, meaning that it doesn't filter connections from anywhere in the internet, which is usually a point of entry for malicious actors.
example4.pp	**Line:** 103 **Column:** 6	Hard-coded credentials	The username is hard-coded into the manifest.
example4.pp	**Line:** 104 **Column:** 15	Empty password	An empty password is very easy to be guessed.
example5.pp	**Line:** 15 **Column:** 26	Weak crypto algorithm	The md5 algorithm used to hash the password is not secure, meaning that's possible to generate the hash without the right password.
example5.pp	**Line:** 76 **Column:** 10	Hard-coded credentials	The username is hard-coded into the manifest.
example5.pp	**Line:** 76 **Column:** 10	Admin by default	The username is an administrator of the node being created, having a high probability of having more privileges than needed, violating the "Least Privilege Principle".

Interview vulnerabilities result:

Vulnerability			Was identified?	Was solved?	Effort? (1-5)	Notes
File	**Location**	**Type**				
example1.pp	**Line:** 5 **Column:** 22	HTTP without TLS				
example2.pp	**Line:** 2 **Column:** 18	HTTP without TLS				
example2.pp	**Line:** 21 **Column:** 2	Hard-coded credentials				
example2.pp	**Line:** 21 **Column:** 2	Admin by default				
example2.pp	**Line:** 22 **Column:** 2	Hard-coded credentials				
example3.pp	**Line:** 4 **Column:** 20	HTTP without TLS				
example3.pp	**Line:** 7 **Column:** 20	HTTP without TLS				
example3.pp	**Line:** 12 **Column:** 22	HTTP without TLS				

example3.pp	**Line:** 15 **Column:** 22	HTTP without TLS				
example4.pp	**Line:** 59 **Column:** 18	Cyrillic HomographAt tack				
example4.pp	**Line:** 87 **Column:** 4	Suspicious comment				
example4.pp	**Line:** 102 **Column:** 5	Invalid IP address binding				
example4.pp	**Line:** 103 **Column:** 6	Hard-coded credentials				
example4.pp	**Line:** 104 **Column:** 15	Empty password				
example5.pp	**Line:** 15 **Column:** 26	Weak crypto algorithm				
example5.pp	**Line:** 76 **Column:** 10	Hard-coded credentials				
example5.pp	**Line:** 76 **Column:** 10	Admin by default				

Pre-task questions

Question (On a scale of 1 to 5...)	1	2	3	4	5	Main thoughts
Rate your previous experience on SATs. What's your opinion on them?						
Rate your previous experience on IaC. What's your opinion on it?						
How easy was it to follow the provided instructions to install and configure the linter? Did you face any technical difficulty?						
How intuitive it was to first understand how the tool works? What could be improved?						

Task Questions

Question (On a scale of 1 to 5…)	1	2	3	4	5	Main thoughts
How clear and easy to understand were the provided Puppet manifests?						
How easy was it to find the vulnerabilities detected by the tool? Do you think that the tool overlooked vulnerabilities?						
How easy was it to understand those vulnerabilities? What are your thoughts on the explanation page?						
How easy was it to solve those vulnerabilities? What are your thoughts on the explanation page?						
How important and useful are the provided configurations to you?						

Pos-Task Questions

Question (On a scale of 1 to 5…)	1	2	3	4	5	Main thoughts
How well does this tool (or a similar one made for a language you use) fit into your workflow?						
How well did the tool alert you for the importance of avoiding these vulnerabilities in the future? Do you feel that the linter is also an educational tool?						
How likely are you to adopt this tool if you are/were a puppet developer?						

Additional considerations from the user:

References

[1] Vera Barstad, Morten Goodwin, and Terje Gjøsæter. Predicting source code quality with static analysis and machine learning. Oct 2014.

[2] Kent Beck. *Test-driven development: by exemple*. Addison-Wesley, 2015.

[3] Hendrik Bünder. Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages. *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, 2019.

[4] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys Tutorials*, 18(3):2027–2051, 2016.

[5] Toma Fernandez. What is infrastructure as code?, Jan 2020.

[6] Ryan Van Fleet. Basic rules engine design pattern, Nov 2019.

[7] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 2016.

[8] Michele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019.

[9] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why dont software developers use static analysis tools to find bugs? *2013 35th International Conference on Software Engineering (ICSE)*, 2013.

[10] Kaspersky Lab. Damage control: The cost of security breaches it security risks special report series.

[11] P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.

[12] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do windows users follow the principle of least privilege? *Proceedings of the Sixth Symposium on Usable Privacy and Security - SOUPS 10*, 2010.

[13] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[14] Mazedur Rahman, Samira Iqbal, and Jerry Gao. Load balancer as a service in cloud computing. *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, 2014.

[15] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[16] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 227–238, 2001.

[17] Marcelo D'Amorim Sofia Reis and Rui Abreu. A critical study on the precision of a state of the art infrastructure-as-code security linter. Submitted for publication 2021.