Università
Ca'Foscari
Venezia

Master Degree
in Computer Science

Final thesis

Staresc - automatic and extendable vulnerability assessment over SSH

Supervisor
Prof. Stefano Calzavara

Co-supervisor
Dr. Riccardo Spampinato

Graduand
Davide Cecchini
862701

Academic Year
2021-2022

Abstract

Regardless of the technical level and the type of target, time is one of the major constraints during both defensive and offensive activities. To address this constraint, the cybersecurity community implemented many tools to automate repetitive tasks. Cybersecurity experts exploit these tools in order to have more time to spend on more tricky (and fun) activities.
In this work we present Staresc: a tool that automates command-line PoCs execution on multiple targets, relying on SSH or Telnet connections. Staresc is an easily extendable tool that performs tests on target machines, the tests are defined in YAML files (called plugins) that the tool can import at execution time.
Together with Staresc, we describe how to properly write its plugins and we outline a practical way to test, and validate, them.
Moreover, Staresc has been compared with the major competitors already available. We show which ideas introduced by other tools we adapt to our use case, which new features we introduced and which motivations led our technical choices.
Lastly, we discuss about possible improvements, covering the possible implementation challenges and their benefits.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

IT systems are becoming more and more complex, they are made up by many software and hardware components. These components are usually very different: they are written in different languages, have different tasks and are developed by different organizations.
This complexity increases the probability of these systems to be affected by vulnerabilities. In order to discover and fix these vulnerabilities, many IT systems are tested through a process called vulnerability assessment.
A vulnerability assessment is a systematic review of security weaknesses in an information system. It evaluates if the target system is affected by known vulnerability and, if needed, provides suggestions regarding the severity, the risk level and the possible mitigation. [30]
This process has changed a lot in the last decades, because the complexity of the IT systems and the number of newly discovered vulnerabilities increased a lot.
In 1999 the MITRE Corporation [11] launched the Common Vulnerability and Exposures (CVE) system [6], in order to track all the newly discovered vulnerabilities. In 2000 there were around 100 new registered CVEs every month.
In this context, the process for detection and remediation of vulnerabilities affecting IT systems was largely performed manually, due to the limited number of new vulnerabilities. The early vulnerability scanners were released in the late 90s (Nessus [13] first release in 1998), they were used to generate a report that, after a manual process of accuracy review and approval, would be passed to the system and network administrators, that would apply the necessary fixes. [7] [46]
With the increasing of the vulnerability discovery rate (more than 20000 new CVEs in 2021 [2]), a change of approach became necessary. It arose the concept of Vulnerability Management: the cyclical prac-

tice of identifying, classifying, prioritizing, remediating and mitigating software vulnerabilities. [43]
It became indispensable to automate as much as possible the Vulnerability Management process, in order to identify and fix as many vulnerabilities as possible. Thus, automatic vulnerability scanners became more and more important. It is not enough that they produce a report with the detected vulnerabilities. They must be able to assign a severity grade to the vulnerabilities, and suggest possible mitigation, keeping the configuration process as simple as possible.
Giving the huge amount of newly discovered vulnerabilities, it is essential for a vulnerability scanner to be extendable, in order to be easily enhanced with checks for new vulnerabilities.
Moreover, it is important that vulnerability scanners do not perform potentially destructive actions on the target systems. They should also provide reports, in a format that is easy to parse, especially for other automatic tools that are used in the Vulnerability Management process.
Cybersecurity community has developed many automatic vulnerability assessment tools. They usually perform a series of checks to identify common misconfigurations and vulnerable software versions.
Unfortunately, some of these tools are not free, some just support a restricted set of protocols and some others do not provide a simple way to develop new tests. To the best of our knowledge, there is no free and easily extendable tool that performs vulnerability assessments using interactive shell protocols.
This work presents Staresc [28], an easily extendable open-source tool that performs vulnerability assessment using interactive shells. This tool is written in Python [24], it supports SSH and Telnet connections and uses them to run parallel tests on target machines. It reads the tests to execute from YAML [35] files called "plugins", and provides reports in various formats (CSV, XLSX, JSON), that show the results of each test, including technical details to manually reproduce the tests.
Staresc's plugins are written in YAML language, their format is designed to make the plugins easy and fast to write. The goal of the project is to let people quickly design and publish new plugins, covering newly discovered vulnerabilities.
Given that it can run tests on parallel connections, and that it can be easily enhanced with additional YAML plugins, Staresc can be successfully applied to vulnerability assessment activities. It can be used to perform the most common checks in a faster way, especially when

multiple target machines are involved.
We present an effective methodology to build and test new plugins, showing an example of a plugin that has been used in real-world vulnerability assessment activities.
Moreover, we compare Staresc with its first version, with Nuclei [15] and with Nessus and OpenVAS [19]. In particular, we focus on how Staresc's plugin language evolved, from Python-based to YAML-based plugins. We show which concepts we have taken from the Nuclei's templating language, and why we consider our plugins easier to develop respect to those of Nessus and OpenVAS.
This work is structured as follow: Chapter 2 provides some technical background. Chapter 3 presents the structure of Staresc's code. Chapter 4 describes how to write and test new plugins. Chapter 5 contains the comparison between Staresc and the major competitors. Chapter 6 contains a discussion on possible improvements for the tool. And lastly, Chapter 7 wraps up the content of this work.

Chapter 2

Background

In this chapter, we give some background notions that will be useful in the rest of the work. In particular, we introduce the major competitors of Staresc, these are tools that either implement the same functionality in a different way, or applies some interesting concepts to different functionalities.
Moreover, we briefly describe the protocols that Staresc uses to perform its scans and the vulnerability that we use to write our example plugin in the next chapters.

2.1 Similar solutions

Staresc is not the first tool that performs automatic vulnerability assessment. There are other scanners that offer valuable features, but each of them has some characteristics that do not fit well our purposes. We wanted to have a light and open-source scanner, tailored for interactive shell protocols. Its plugins must be simple static files, easy to write and understand. Following, we present some of the vulnerability scanners from which we took inspiration, explaining which features fit well for our needs and which not.

2.1.1 Nessus and OpenVAS

Nessus [13] was created by Renaud Daraison in 1998, it was an open-source security scanner. After seven years, in 2005, Tenable Network Security made it a closed-source software and started to distribute it under license. OpenVAS [19] is an open-source [20] spinoff of Nessus, it was originally named GNessUs. OpenVAS and Nessus are similar tools, they share a part of the core engine and both run scans based on the content of their plugins. [49]

Moreover, plugins for both Nessus and OpenVAS are mostly written with the Nessus Attack Scripting Language (NASL), a scripting language, similar to the C and Perl languages. [41]
Nessus is considered better than OpenVAS since it provides a better user interface, supports more plugins (around 170000 Nessus' plugins vs. around 80000 OpenVAS' plugins), and performs scans with less false positives and false negatives.
Nessus and OpenVAS perform a wide variety of scans:

- They can check service versions in order to find Common Vulnerability Exposures (CVEs) that affect the target system.
- They can check for default and common credentials used in the exposed services, Nessus can also employ Hydra, an external login cracker that supports a wide variety of protocols.
- They can find common misconfiguration, for example not patched software or not trusted certificates.
- They can perform web application tests in order to find most common types of vulnerabilities, like path traversal and SQL injection.
- They can perform authenticated vulnerability scans, a type of scan that exploits known credentials to run checks on the target system (e.g. SSH credentials to run shell commands).

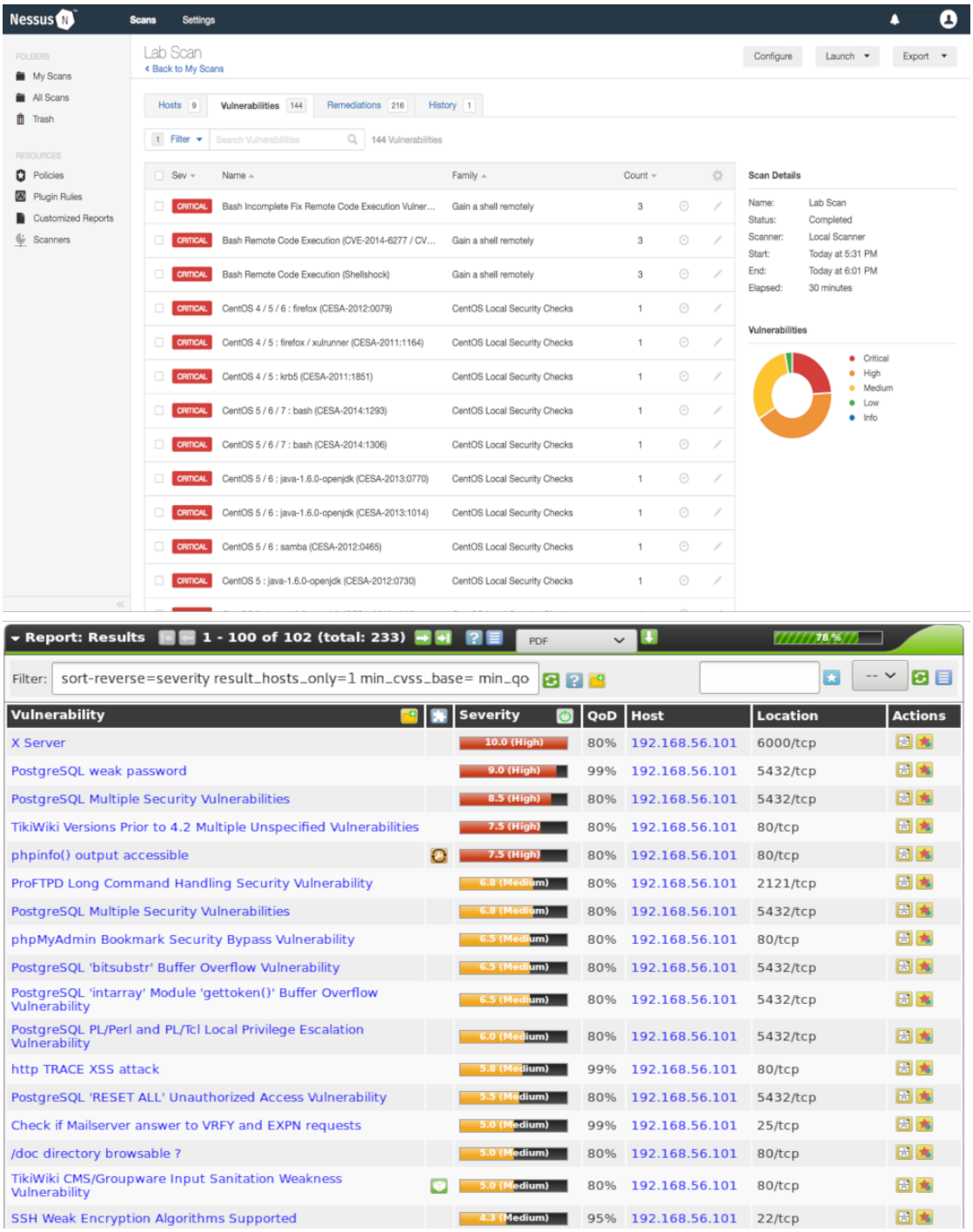

Figure 2.1: Security scan reports of Nessus (upper image) and OpenVAS (lower image).

Despite their useful features and the intuitiveness of their web interfaces (Figure 2.1), these tools do not fit well for our purposes. Nessus works well, but it is not open source and the NASL plugins are not static files. Even if NASL language makes plugins more flexible, it can make the development of them slower, since a person has to learn how the language works and know how to use the Nessus and OpenVAS' libraries.

2.1.2 Nuclei

Nuclei [15] is an open-source vulnerability scanner written in Go language [29] by ProjectDiscovery [23]. It supports a wide variety of protocols, including TCP, DNS, HTTP, etc.. This tool is based on the concept of templates: YAML files specifying how the tool should scan a system, in order to check for the existence of a given vulnerability.
Nuclei's templates do not simply describe which requests the scanner must send to the target system, but they also contain parsing rules to apply to the results of these requests (e.g. responses).
In their most basic form, templates are simple YAML files, easy to understand. However, Nuclei offers the possibility to define and use variables and functions in its templates. It is possible to use them through a Domain-Specific Language (DSL), an expression language that allows to build more flexible templates. [17]
For example, it is possible to specify to the scanner that a vulnerability is found if the base64 encoding of a given portion of the response is equal to a given variable, or it is possible to specify to the scanner that a given field must have a random value. These behaviours are hard to specify with strictly static templates.
From the Listing 2.1 it is possible to see a simple template that checks if an HTTP page contains a given regex.

```
1  id: amazon-mws-secret-token-value
2
3  info:
4    author: puzzlepeaches
5    name: Amazon MWS Secret Token
6    severity: medium
7
8  requests:
9    - method: GET
10     path:
11       - "{{BaseURL}}"
12
13     extractors:
14       - type: regex
15         part: body
16         regex:
17         - "amzn\\.mws\\.[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f
   ]{4}-[0-9a-f]{4}-[0-9a-f]{12}"
```

Listing 2.1: Simple Nuclei's template.

There is an "info" section that contains information about the template and the vulnerability that is checked. Then, there is a part that defines the web request that must be sent to the target URL, notice

the usage of the variable "BaseUrl" to build the path of the target. For the given request, the template developer defines one extractor that uses a regex to extract a portion of text from the body of the HTTP response.
Despite its open-source nature, its flexibility and its easy-to-develop YAML templates, Nuclei does not support interactive shell protocols, and so, it can not be used in authenticated SSH/Telnet based vulnerability assessments.

2.1.3 PEASS-ng

PEASS-ng (Privilege Escalation Awesome Scripts SUITE new generation) [21] is a suite of tools mainly maintained by Carlos Polop [3]. As the name suggests, the tools of PEASS-ng are enumeration tools used to perform privileged escalation. The ideal scenario in which they are applied is the one in which an attacker has access to a shell on the target machine. The attacker's goal is to gain horizontal or vertical privileged escalation. To achieve it, he executes the enumeration tool on the target machine and check its output. In this output, the tool shows a bunch of information that can be useful to identify privileged escalation vectors. [44]
The three main tools of this suite are WinPEAS, LinPEAS and MacPEAS, one for each main Desktop OS (Windows, Linux and MacOS). WinPEAS is an .exe or a .bat file, while LiPEAS and MacPEAS are the same .sh file that identifies automatically if it is run on a MacOS or a Linux machine, and performs the proper checks.
It seems that the tools of PEASS-ng are perfect for our needs: given an interactive shell on a target machine, we just need to upload the right tool, execute it and see which vulnerability it reports. So, why did not we just write a script that uploads WinPEAS or LinPEAS on the target machine and executes it?
The main problem is that WinPEAS and LinPEAS are not vulnerability assessment tools, they are enumeration tools. This means that they do not simply report the vulnerability found on the target machine, but also a bunch of information that is not directly connected to a specific vulnerability. For example Linpeas reports the executable files that have the SUID permission set. This is not directly a vulnerability, but it can become one, if one of the files is owned by a privileged user and can be exploited to run privileged commands. [40] From the Figure 2.2 it is possible to notice that a lot of not-vulnerable files are reported as potentially vulnerable.

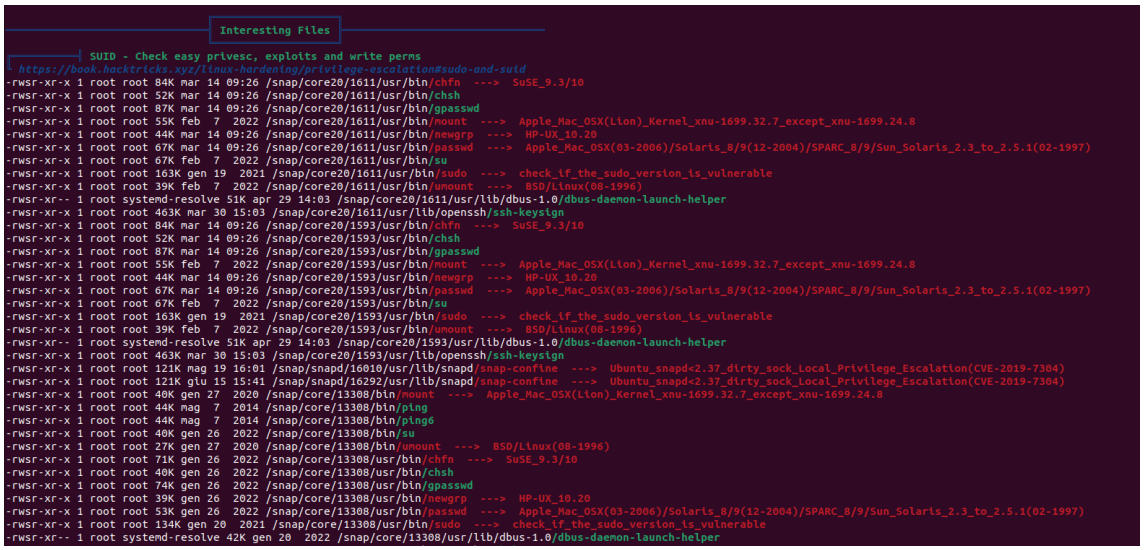

Figure 2.2: Linpeas report of SUID files.

This behaviour makes the output of WinPEAS and LinPEAS too verbose for our purpose, leaving to the user the repetitive task of cleaning the reports from the useless parts.

Table 2.1 contains a recap of pros and cons of the principal competitors of Staresc. We omitted PEASS-ng, since we do not consider it one of the principal competitors. Indeed, even if its tools have interesting features, they are not vulnerability assessment tools.

2.2 Interactive shell protocols

Our vulnerability scanner executes commands from a Command-Line Interface (CLI) on remote targets. To gain access to these CLIs, Staresc uses some client libraries that implement interactive shell protocols.
In this section we describe the two protocols that are supported by Staresc: Telnet [36] and SSH [50]. Even if they are the most popular interactive shell protocols in Unix-like environment, and cover a great number of use-cases, we will show that Staresc can be easily enhanced, adding support for other protocols.

2.2.1 Telnet

Telnet is an application level protocol developed on top of the TCP protocol in 1969. It can be used to provide a two-way, collaborative and text-based communication channel between two machines: a client and a server. This communication channel can be used by the client to perform a variety of activities on the server, like editing files, running programs, check emails and play simple games. [42]
Very often Telnet is used by the client to access to the server shell. Despite this is its major usage, it was not designed to be secure on public accessible networks; indeed, its communication channel is not encrypted and an attacker can perform a Man In The Middle (MITM) attack [10], sniffing password and other sensitive information. We have chosen to support it because it is still used in some internal networks, and many possible target machines do not support more secure protocols.

2.2.2 SSH

The Secure Shell Protocol (SSH) is a cryptographic network protocol designed in 1995 by Tatu Ylönen. Despite it is mainly used to provide a secure connection, that a client can use to open a shell on a server machine, SSH can be used to transfer or copy files to and from the remote machine, make tunneling and port forwarding, and establish X11 connections. SSH standard authentication requires username and password, but, for a more secure connection, it is possible to define a public-private key pair to use instead. [27]
SSH and its implementations (e.g. OpenSSH) have proven to be sufficiently reliable and versatile, and this made them a de facto standard in managing remote shell sessions.

2.3 Sudoedit vulnerability CVE-2021-3156

In the following chapters, we will often use an example plugin to show how Staresc works. This plugin has been designed to check if the target systems contain a version of the sudo binary [31] vulnerable to the CVE-2021-3156 [5] vulnerability.
This vulnerability has been discovered by the Qualys Research Team [26], that publicly disclosed it on 26/01/2021. [48]
It is a heap-based buffer overflow that can be exploited to achieve privileged code execution on the target machine.

Sudo is a utility for Unix-like computer Operating Systems that allows to execute programs with the privileges of another user, by default the root user.
On the majority of the systems, sudo is a binary with the SUID permission that is owned by the root user, this makes it a possible target for a privileged escalation attack.

2.3.1 Technical details

The vulnerability raises due to a wrong implementation of an array copy. Inside the `set_cmnd`() function, there is a loop that copies the command-line arguments into a buffer, that is placed in the heap memory (see Listing 2.2).

```
852         for (size = 0, av = NewArgv + 1; *av; av++)
853             size += strlen(*av) + 1;
854         if (size == 0 || (user_args = malloc(size)) == NULL) {
              ...
857         }
858         if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
              ...
864             for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
865                 while (*from) {
866                     if (from[0] == '\\' && !isspace((unsigned char)from[1]))
867                         from++;
868                     *to++ = *from++;
869                 }
870                 *to++ = ' ';
871             }
              ...
884         }
885           ...
```

Listing 2.2: Sudo code that contains the buffer overflow vulnerability.

This heap buffer is `user_args`, its size is calculated summing up the sizes of the command-line arguments (included the string terminator '\0') in the for loop at lines 852-853.
Then, if the binary is executed in the proper mode, the loop at the lines 864-871 copies the command-line arguments, char by char, into `user_args`. The variable `to` keeps a pointer to the destination buffer. In particular, it points to the position that will contain the next copied character.
For each argument, the variable `from` is initialized with the pointer to the first character. The single argument is copied with the while loop at the lines 865-869. At each iteration of the while loop, the character pointed by `from` is copied in the position pointed by `to`, and then, both `to` and `from` are incremented. At the end of the while loop, a

blank space is copied at the end of the copied argument.
The vulnerability is introduced with the if statement on the lines 866-867. If an escape character is found (a backslash), then `from` is incremented by one.
This behavior has been introduced because, at the start of the program, all the meta-characters are escaped with backslashes. With this if statement, sudo unescapes these meta-characters, skipping the backslashes.
The problem arises when an argument ends with a backslash character and the escaping phase is skipped. Indeed, in this scenario, we will be in the situation in which the `from`[0] content at line 866 is the '\' character, and the `from`[1] content is the null character (the string terminator).
The if condition is taken and `from` is incremented by one, making it pointing to the null character. Then the null character is copied into the variable pointed by the `to` variable. Eventually, `from` and `to` are incremented.
Since `from` will not point to the null character when it is checked in the while condition, the while loop does not stop at the string terminator of the argument, causing a buffer overflow that overwrites the memory that resides after the `user_args` buffer.
The researchers of Qualys found that, to set the right mode and reach the vulnerable loop, skipping the escaping phase, it is sufficient to execute sudo with the command "`sudoedit` -s".
So, to trigger this vulnerability it is sufficient to run the command:

```
sudoedit -s somestring\
```

Notice that, depending on the shell you are using, the trailing backslash may have to be escaped.
Even if the sudo developers have patched the binary rapidly, many vulnerable versions are present in many systems. The legacy versions from 1.8.2 to 1.8.31p2 and the stable versions from 1.9.0 to 1.9.5p1 are vulnerable in their default configuration.

	Pros	Cons
Nessus	• Intuitive GUI • Big pool of plugins • Supports authenticated scans over shell protocols	• Closed source • Complex plugin development
OpenVAS	• Intuitive GUI • Big pool of plugins • Supports authenticated scans over shell protocols	• Smaller pool of plugins • Complex plugin development
Nuclei	• Open Source • Big pool of plugins • Simpler plugin development	• Does not support shell protocols

Table 2.1: Pros and Cons of the main competitors of Staresc.

Chapter 3

Staresc Development

In this chapter, we will describe the structure of the code of Staresc, explaining the technical problems and the motivations behind our solutions. To make the code look neater, and easier to maintain, we divided Staresc into four parts:

- Core, containing the core engine; the part that implements the central logic and that uses all the other parts.
- Connection, that handles the connections to the target machines.
- Exporter, that contains the components used to export the results of the scan in different formats.
- Plugin parser, that contains all the logic used to parse the YAML plugin files.

All these parts are contained in the directory "staresc" of the repository. It contains all the Python source code of the tool, except for the file "staresc.py", that is contained in the main directory. In this file we can find the "main" function of the program, it parses the command-line arguments and sets up the execution environment based on them. Eventually, it passes the execution to the core part of the tool.

3.1 Core

The core part of Staresc is contained in the directory "staresc/core". It consists of two main classes: `Staresc` (defined in "staresc/core/staresc.py") and `StarescRunner` (defined in "staresc/core/runner.py").
A `StarescRunner` object is created in the `main`() function. First of all, it parses every YAML plugin, using the plugin parser part. Then

it creates a thread for each target connection and launches a scan on the target machine using all the parsed plugins.
For each scan, a `Staresc` object is created, it represents the opened shell session on the target machine. For each plugin, the thread calls the `do_check`() method of the Staresc object, giving as parameter the relative `Plugin` object. The `Staresc` object uses the `Plugin` object to perform the scan on the target machine, and then it returns the results of the scan using an `Output` object (see Subsection 3.3.1).
Moreover, the `Staresc` object parses the output of the tests, according to the plugin specification, in order to establish if the target machine is vulnerable or not.

3.2 Connection

The connection part is responsible to offer to the core part an interface to handle and use the connections to the target machines. It consists of a class hierarchy, with the class `Connection` at the top. The `Connection` class defines the scaffold of a `Connection` object and requires that subclasses implement methods like `connect`(), `run`() or `__init__`(). In this way, to add a new type of connection, it is enough to create a subclass of `Connection` that implements these three methods.
Currently, three types of connection are defined:

- `SSHConnection`
- `TNTConnection`
- `SSHSSConnection`

3.2.1 SSHConnection

`SSHConnection` handles SSH connections, using the library Paramiko. It creates a Paramiko `SSHClient` in the constructor, then, in the `connect`() method, it creates an SSH connection to the target host, performing the required authentication. Eventually, with the `run`() method, it executes the given command on the target machine; each command is executed in a dedicated SSH session.

3.2.2 TNTConnection

`TNTConnection` is the class that implements the connection interface for the Telnet connections, using the Telnetlib library. It uses, in the method `Connect`(), the given credentials to establish a Telnet connection with the target machine, then, in the `run`() method, it executes the given command in the opened Telnet session.
Given the interactive nature of the Telnet protocol, it is not easy to understand when the output of a command finishes. Telnetlib offers two types of functions to read command output from the communication channel: blocking and non-blocking ones. Since there is no way to know when the output of the given command is over, blocking reads lead to a deadlock. Indeed, when the command output is over, we do not know if it is over. So, we would perform a blocking read, that waits for some data that will never come.
Non-blocking ones usually do not return any output from the launched commands. They immediately read from the channel without waiting any output from the target machine, even if the channel is empty.
We identified and explored three different solutions, and eventually we choose to implement one of them:

- Blocking read interrupted by a timeout: it consists on the introduction of a timeout that terminates the blocking reads. In this way we should exit by the deadlock on the last read (of each command output). Unfortunately, this approach does not fit well for commands requiring more time to produce their output. For example, let's consider a situation in which we have a timeout of ten seconds, after which we kill the blocking read. If we execute on the target machine a typically fast command like "`ls`", this solution would work fine, since the output would be produced in less than ten seconds. We would call a series of blocking reads (each one consumes a portion of the output) and the last one, called after the output is over, would wait for ten seconds, until the timeout expires. But, for commands (or command series) that require more than ten seconds to produce their output (e.g. "`sleep` 15; `ls`"), our blocking read would be interrupted by the timeout, before the command can produce any output. In this way, we would miss the results of the command we run on the target machine. It becomes very complex to choose the right timeout. A too short one leads to the missing of the results of longer commands, while a too long one requires too much time to finish the read process of the output.

Moreover, with a fixed timeout of ten seconds, we would spend a lot of time waiting for the end of the last read, even for the commands that require less time to produce the output (e.g. "`ls`").

- Non-blocking read launched after a timeout: with this solution, we wait for a time and then we perform a non-blocking read. In this way we should avoid the problem of reading an empty channel, giving to the target machine enough time to produce the output. This solution does not work well in practice. In fact, it is difficult to find a right timeout that fits with both long and short commands. For example, a timeout of ten seconds would fit well with commands that require less than ten seconds to produce the output. On the other hand, it would not give enough time to the target machine to produce the output of the commands that require more than ten seconds (see the example before).
  Moreover, as the previous solution, with a fixed timeout we have a fixed overhead for each command.

- Blocking read with a canary: this is the solution we adopted. It consists on a blocking read that waits until we find a given string on the command output.
  We send the given command followed by a "`echo` <canary>", where <canary> is a random string. Calling the Telnetlib method `readuntil`(<canary>), we perform a blocking read that reads until it matches the <canary>.
  In this way, we get all the output of the given command followed by the <canary>. Removing the <canary> from it, we obtain the output of the command without wasting time using timeouts.
  There is a similar solution, used by softwares like Pexpect [22], that consists on the re-definition of the `PS1` environment variable (the one that defines the prompt), with a canary-like value. In this way, when the command output is over, the canary-prompt is printed and the client knows that no further read is needed. We choose to not implement this solution, since there are many custom shells that do not offer the possibility to customize the prompt string, while the `echo` command is almost a standard and it is available in many types of shell.

3.2.3 SSHSSConnection

`SSHSSConnection` stands for SSH Single Session Connection, it is the class used to handle SSH connections that can not spawn more than one session. During our tests on a real-world environment, we found some server machines that had, on the SSH configuration file, the following setting: "MaxSessions 1".
This setting forced us to implement an SSH client class, that executes all the commands on the same session. `SSHSSConnection` is very similar to `SSHConnection`, both use the library Paramiko. The difference is that `SSHSSConnection` directly uses the lower lever channels that are internally used by Paramiko.
The usage of these channels forced us to implement the same mechanism that we used in `TNTConnection`, the blocking read with a canary.

3.2.4 Connection string

All the information that Staresc uses to connect to a target, are specified using the connection string. It is possible to specify a single connection string as a command-line argument, or multiple connection strings specifying a file with the flag "-f/--file" (one connection string for each line). A connection string has the following format:

$$scheme : //user : secret@host : port/$$

Where:

- *scheme* is a string that identifies the connection class to use. ssh for `SSHConnections`, tnt for `TNTConnection` and sshss for `SSHSSConnection`.
- *user* is the username used to establish the connection.
- *secret* is the password of the given user.
- *host* is the hostname or the IP of the target machine.
- *port* is the port, on the hostname, listening for the connection.

3.3 Exporter

The exporter part of Staresc defines how the tool generates all the output regarding the scans, except for debugging messages and errors. This part is defined in the "staresc/exporter" directory. It consists on three main components:

- the `Output` class, defined in the file "staresc/exporter/output.py".
- the `StarescExporter` class, defined in the file "staresc/exporter/exporter.py".
- the `StarescHandlers` class and its subclasses, defined in the file "staresc/exporter/handlers".

3.3.1 Output

The objects of the class `Output` are used to keep the state of a single scan to a target machine. Each object contains: a reference to a target connection, a reference to a plugin defining the tests to do, and some fields that keep the state of the scan (e.g. if the vulnerability is found).
In the method `do_check`(), the `Staresc` object creates an `Output` object, to keep track of the state of the scan. When the method finishes, it returns the `Output` object to the thread that is executing the scan.
The last thing that a thread does before passing to the next plugin is to append the `Output` object (of the current plugin scan) to the `StarescExporter`'s queue.

3.3.2 StarescExporter

`StarescExporter` is a class that is never instantiated, it keeps: a queue of `Output` objects, a list of `StareschHandler` objects, and the relative methods that can be used to append objects to these lists.
During the startup, Staresc:

1. Parses the command-line parameters (that specify which handlers should be instantiated).
2. Instantiates the necessary `StarescHandler` objects.
3. Registers the handlers to the `StarescExporter`, appending them to the list of handlers.

Every time a scan of a plugin finishes on a target machine, an `Output` object is appended to the queue in the `StarescExporter`, using the method `import_output`(). From here, the `StarescExporter` calls the `import_handler`() methods of all the handlers. It passes the `Output` object as argument, in order to notify them the result of the scan.
`StarescExporter` implements also an `export`() method, it cycles through the registered handlers, and uses them (calling their `export_handler`() methods) to generate the reports passing the list of `Output` objects. The `export`() method is called when all the scans on the target machines are over.

3.3.3 StarescHandler and its subclasses

`StarescHandler` is the parent class of all the handler classes, it defines three methods: the constructor, `import_handler`() and `export_handler`(). For each type of report, an handler must be implemented. It must extend the `StarescHandler` class, implementing at least the methods `import_handler`() and `export_handler`().
`import_handler`() is used to "notify" to a handler that a new `Output` object has been created, it is called by the method `import_output`() of the `Exporter` class. `export_handler`() is called by the `export`() method of the class `StarescExporter`. It is called at the end of the whole scan, in order to generate the relative report.
Actually, we implemented four types of handlers: `StarescCSVHandler`, `StarescXLSXHandler`, `StarescJSONHandler` and `StarescStdoutHandler`. The first three handlers are pretty similar, they implement only the `export_handler`() method and generate, respectively, reports in the CVS, XLSX an JSON format. `StarescStdoutHandler` is a bit different, its `export_handler`() method generates a brief recap of the vulnerability assessment that is printed on the standard output. Moreover, it overrides the method `import_handler`(): every time an `Output` object is generated and passed to this method, this handler checks if the scan, relative to the `Output`, has found the vulnerability and, if so, logs it to the standard output.
A user can specify with the flags "-ocsv <filename>", "-oxlsx <filename>", "-ojson <filename>", which handler to use, and which name to use for the report file. With the flag "-oall <filename>", all the handlers are used, and one report for each format is created.
The `StarescStdoutHandler` can not be specified with a command-line flag, since it is used by default.

3.4 Plugin parser

The plugin parser part is used to parse and transform the data of the YAML plugin files into `Plugin` objects, one for each plugin file. This part is organized in a series of classes, whose objects are disposed in a pyramidal hierarchy.
In the upper position of the hierarchy, there is the `Plugin` class, its objects represent a single plugin that has been parsed from a YAML file. See the Chapter 4 to know the structure of the plugin files, how they work and how to write them. By now, it is enough to know that each file contains some metadata, like the name of the author or the description of the vulnerability found by the plugin, and a list of tests. The `Plugin` objects is constructed using a dict object build from the YAML file. It saves all the metadata using its fields, then it creates a list of `Test` objects, one for each test specified in the file.
The `Test` objects are constructed using the relative dict field, they contain three fields: the command to execute on the target machine, a list of `Parser` objects to use to parse the command output and a list called `plugin_tests`. The `plugin_tests` field is covered in Section 4.2.
The `Parser` class is never directly instantiated, it is the root of a class hierarchy comprising all the possible types of parser. The class `Parser` defines the scaffold that each `Parser` object should have, leaving the implementation to the classes that extend it. Each `Parser` contains some fields that specify its configuration, for example the field `parts` specifies on which parts of the command output (stdout or stderr) apply the parser, while the field `condition` specifies how to merge the results of parser's rules. An extensive explanation of the fields that can be used to configure a parser is given in the Section 4.1.
Currently, there are two types of parser: `Matcher` and `Extractor`. A `Matcher` is a `Parser` that checks if the given rules match with the command output, while an `Extractor` uses the rules to extract part of the command output and to pass it to the next parser in the pipeline (pipeline is covered in Subsection 4.1.1).
The central part of a `Parser` are the rules. They are applied on the output of the command to establish the presence of the vulnerability. There are two types of rules: "regex" or "word". The "regex" rules are used by the `Matcher` to match a given regex, and by the Extractor to extract the portion of text that matches the given regex. The "word" rules are simple strings, a `Matcher` checks if the string of the "word" rules are contained in the command output, while an `Extractor` ex-

tracts those strings from the command output. The Figure 3.1 gives a summary of the plugin parsing phases.

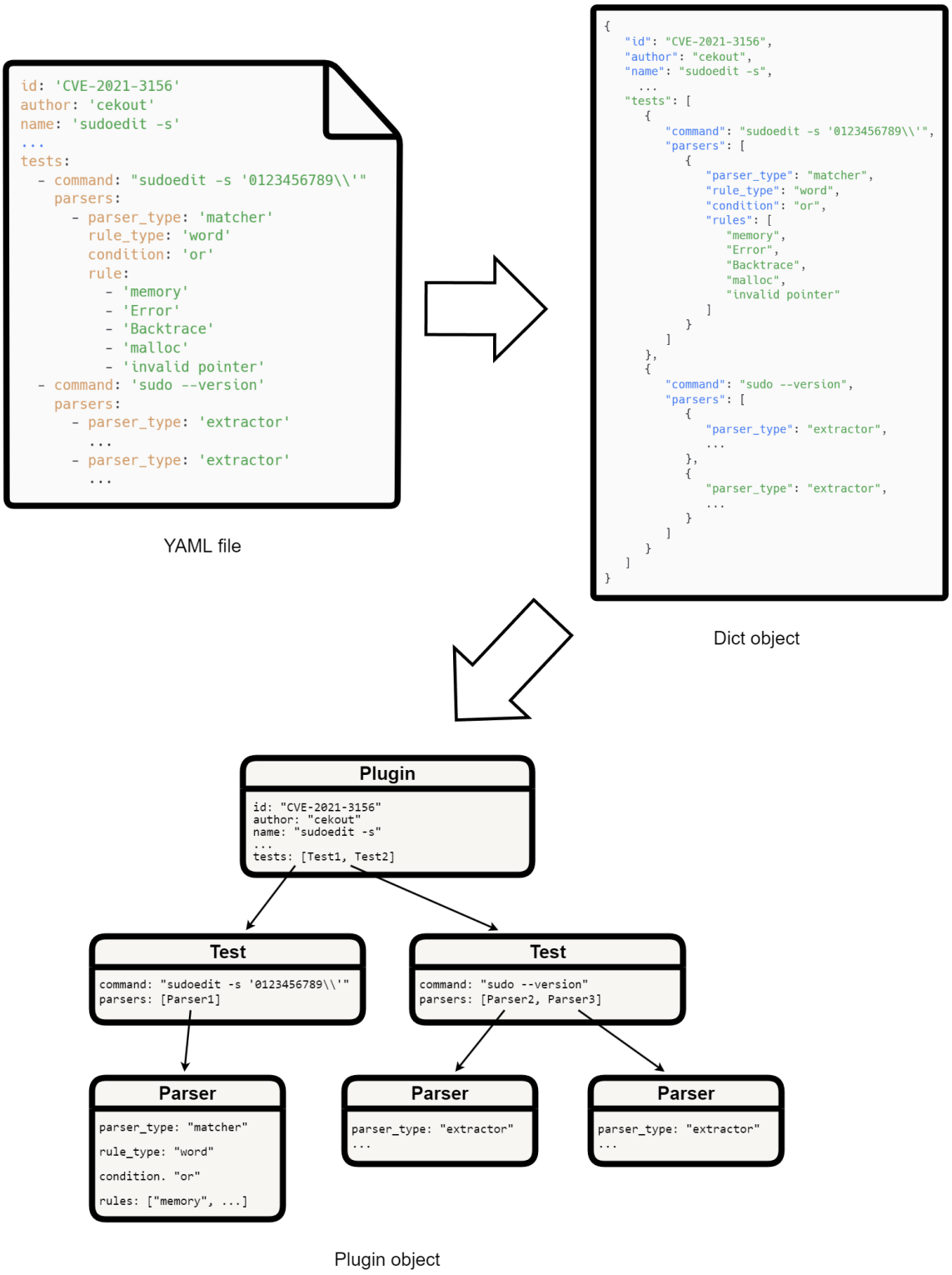

Figure 3.1: Plugin parsing phases.

During the scan of a plugin on a target machine (method `do_check`() of the `Staresc` object), the `Tests` are extracted from the `Plugin`. Then, the relative command is executed on the target shell, and the parsers of the given `Test` are applied to the output of the command. In the end, the `Output` relative to the current scan is updated.

Chapter 4

Plugin production

In this chapter, we explain in details how the plugins work and how to develop them. To make the process clearer, we will show an example plugin taken from the ones that have already used in real-world environments.
Moreover, we will explain how it is possible to debug a new plugin, showing an example with the plugin we introduced in the plugin development section.

4.1 Plugin development

The plugin files of Staresc are written in the YAML format. We choose this format following the example of the Nuclei scanner.
YAML is a simple human-readable format. It is easily understandable and it allows to quickly create new plugins, starting from a command-line PoC.
Plugin files can be divided into three sections:

- Metadata fields
- Plugin directives
- `tests` field

The metadata fields contain information about the plugin, the vulnerability that is checked, and the author.
`id` is the only mandatory metadata field, it is a string that acts as the unique identifier of the plugin.

The other metadata fields are:

- `author`: a string that identifies the author of the plugin.
- `name`: a string that identifies the name of the plugin or the name of the vulnerability.
- `description`: a string describing the plugin or the vulnerability.
- `remediation`: a string containing a brief explanation of how to patch the checked vulnerability.
- `cve`: a string containing the CVE code (if exists) of the checked vulnerability.
- `reference`: a string containing references (can be a URL or other) relative to the plugin or the vulnerability.
- `cvss`: a float number that identifies the CVSS V3 score of the checked vulnerability.
- `severity`: a string containing the severity rating (based on the cvss score) of the checked vulnerability.
- `cvss_vector`: a string containing the value of the CVSS V3 vector of the checked vulnerability.

Currently, the only supported plugin directive is `matching_condition`. It defines how the results of the tests must be merged. With an "and" value, the target machine is considered vulnerable if all the tests report that it is vulnerable. With an "or" value, it is sufficient that at least one of the tests reports the vulnerability to declare the target machine vulnerable. The default value of this field is "and".
To avoid useless tests, if the value of this field is set to "and", and one of the tests fails, the remaining tests are skipped. Similarly, if the value is set to "or" and one of the tests successes (confirms that the vulnerability is present), the remaining tests are skipped.
The field `tests` is a list containing the specification of the various tests. Each test contains a `command`, that is executed on the target machine, and a list of `parsers`, that are applied to the command output. The tests are executed according to the order in the YAML file.

4.1.1 Parsers

For each test we have a list of one or more parsers. The parsers are the components that parse the output of the command executed on the target machine. This parsing process is used to establish if the machine is affected by the given vulnerability.
Currently, Staresc supports two types of parser:

- matcher: given one or more rules, it checks if they match or not. The "ideal" result is a boolean value.
- extractor: given one or more rules, it extracts from the result of the command the portion of text that matches them.

The rules are the core part of a parser, defining what to match in/extract from the command output. They are defined as a list of strings in the `rules` field, inside each parser.
Currently, two types of rule are supported:

- word: a simple string, the matcher/extractor checks if it is contained in the text to parse.
- regex: a string that defines a regex, the matcher/extractor looks for text portions that match the regex.

The other fields of the parsers define how the engine should apply the rules, and how it should merge their results, they are:

- `parser_type`: a string defining the type of the parser, can be set to "matcher" or "extractor".
- `rule_type`: a string that defines the type of rules, can be "regex" or "word".
- `part`: a string that specifies which part of the command's output to parse, can be "stdout" or "stderr". By default the parsers examine both stdout and stderr.
- `condition`: a string that specifies how rules' results should be merged. An "or" implies that one matching rule is sufficient to report a positive test, while an "and" value requires that all rules should return a positive result, in order to have a positive test.

- `invert_match`: a boolean field supported only by the matchers. If set to "True", it reverses the behavior of the parser. If the matcher normally checks for the presence of some strings ("word" rule type), with a "True" `invert_match` field, it checks for the non-presence of the given strings.

To make Staresc's plugins more flexible, we introduced the possibility to apply more than one parser on the same command output.
The list of parsers is applied to the output in a pipeline-like process. In this pipeline, each parser receives the result of the parser before it. Then, it parses it, and eventually, it passes its result to the next parser.
The output (and input) of each parser has the shape showed in Listing 4.1.

```
{
    <matched_or_extracted>,
    {
        "stdout": <extracted_stdout>,
        "stderr": <extracted_stderr>
    }
}
```

Listing 4.1: The data structure that passes through the pipeline of parsers.

- `<matched_or_extracted>` is a boolean field that is true if all the matchers (or extractors), before the current one. have matched (or extracted) something, otherwise it is false. The initial value (at the start of the pipeline) of this field is "true". If a parser receives as input an object with this value set to false, it skips the parsing process.

- `<extracted_stdout>` and `<extracted_stderr>` are strings that represent the portions of text extracted by the extractors before, respectively from the stdout and stderr. The initial values (at the start of the pipeline) of these fields are the result (stdout/stderr) of the command executed in the target machine. The matchers affect only the boolean value (`<matched_or_extracted>`), they do not modify the value of `<extracted_stdout>`/`<extracted_stderr>`.
  The extractors, on the other hand, set the boolean to "true" if they extract something, or "false" if they do not. Moreover, they substitute `<extracted_stdout>`/`<extracted_stderr>` with the

portion of text they extract. If they can not extract anything, <extracted_stdout> and <extracted_stderr> are set to empty string.

4.1.2 Example plugin

To show how the various properties of the YAML file should be used, we show an example plugin.
This plugin has been implemented to detect the CVE-2021-3156 vulnerability. It is a heap-based buffer overflow that, if exploited, allows an attacker to obtain root privileges from a regular user.
We choose this vulnerability since it perfectly represents the type of vulnerability that we search with Staresc. It is a local vulnerability, so it can be found only by an authenticated user, that can open a shell session inside the machine.
Moreover, it is a privileged escalation vector that is exploitable by regular users. Indeed, it is sufficient to have the permission to run the sudoedit command in order to exploit it.
As we have seen in the Section 2.3, to trigger the vulnerability it is sufficient to run the command:

```
sudoedit -s "somestring\\"
```

The double backslash is interpreted by the bash shell as a single backslash (the first backslash is used as an escape character).
The ending backslash of the string triggers a one-byte out-of-bound write in a heap chunk. This write corrupts some heap control structure with a null byte (the ending byte of the parameter string), and eventually results in a crashing malloc() allocation (Listing 4.2).

```
cekout_test@test-vuln:~$ sudoedit -s "somestring\\"
malloc(): invalid next size (unsorted)
Aborted (core dumped)
cekout_test@test-vuln:~$ sudo --version
Sudo version 1.8.31p2
Sudoers policy plugin version 1.8.31p2
Sudoers file grammar version 46
Sudoers I/O plugin version 1.8.31p2
```

Listing 4.2: Sudo binary crashing using the PoC command.

Another indicator for this vulnerability, is the check on the version of the sudo binary: the legacy versions from 1.8.2 to 1.8.31p2 and the stable versions from 1.9.0 to 1.9.5p1 in their default configuration are vulnerable.
Our plugin contains both the checks: the crashing command and the

version verification. For the latter, we use a regex used by the Linpeas software. In Listing 4.3 you can see the content of the plugin. The fields `name`, `description`, `remediation`, `cve`, `reference`, `cvss`, `severity` and `cvss_vector` contain various information about the vulnerability.
The field `tests` contains a list of two tests, the first is the crashing command, while the second is the check on the version of the sudo binary.
The first test is made up two fields: the `command` field contains the command that triggers the crash, while the `parsers` field contains the specification for the matcher that parses the command output.
This matcher (`parser_type` set to "matcher") checks if at least one (`condition` set to "or") of the strings (`rule_type` set to "word") specified in the field `rules` is present in the output of the command (stdout or stderr, since the field `part` is not specified).
The second test runs the command "`sudo` --version", and pass its output to an extractor that extracts the portion of text that matches the regex "Sudo version .*\n". This is the line that contains the version of the sudo binary.
The extracted portion is then passed to a second extractor, that applies the version regex in order to see if the sudo binary has a vulnerable version. If the first extractor can not extract anything, the second extractor will not be applied.
Given that the field `match_condition` is not specified, both the tests must return a positive result (vulnerability found) in order to consider the target machine vulnerable. This behaviour is used to minimize the false positives.
Summarizing, the following are the actions that Staresc does when it performs a scan using the sudoedit plugin. Given an established connection to the target machine, Staresc runs the command of the first test (the crashing one), then it collects the output and passes it to the parser of the first test.
This parser checks if at least one of the given strings is present in the command output. If so, this test is considered positive, and Staresc proceeds with the second one. If the matcher can not find any string, the result of the test is considered negative, and Staresc does not proceed with the second test, considering the target machine not vulnerable.
For the second test, Staresc collects the output of the command and sends it to the first extractor. If it can extract a portion of text, Staresc sends the extracted part to the second extractor, otherwise it

considers the test negative. If also the second extractor can extract a portion of the test with its regex rule, the test is considered positive and the target machine is marked as vulnerable.
The final reports contain, for each target connection, both the information about the plugin/vulnerability and the results of the scan, reporting if the machine is vulnerable or if the scan requires too much time (triggers a timeout).

```
id: 'CVE-2021-3156-sudoedit'
author: 'cekout'
name: 'Component with known vulnerabilities: sudo before 1.9.5p2'
description: |
  Sudo before 1.9.5p2 contains an off-by-one error that can result in a
  heap-based buffer overflow, which allows privilege escalation to root via
  "sudoedit -s" and a command-line argument that ends with a single backslash
  character.
remediation: |
  Update the software to a more recent version
cve: 'CVE-2021-3156'
reference: 'https://nvd.nist.gov/vuln/detail/CVE-2021-3156'

cvss: 7.8
severity: 'High'
cvss_vector: 'CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H'

tests:
  - command: "sudoedit -s '0123456789\\'"
    parsers:
      - parser_type: 'matcher'
        rule_type: 'word'
        condition: 'or'
        rules:
          - 'memory'
          - 'Error'
          - 'Backtrace'
          - 'malloc'
          - 'invalid pointer'
  - command: 'sudo --version'
    parsers:
      - parser_type: 'extractor'
        rule_type: 'regex'
        rules:
          - 'Sudo version .*\n'
      - parser_type: 'extractor'
        rule_type: 'regex'
```

```
38          rules:
39            - '([01].[012345678].[0-9]+)|(1.9.[01234])|(1.9.5
    p1)'
```

Listing 4.3: Staresc's plugin for the CVE-2021-3156 vulnerability.

4.2 Plugin testing

To make plugin development easier, we implemented a "test_plugins" execution mode for Staresc. When a plugin is parsed, many errors can occur, especially with strings that represent regexes. These problems arise because the YAML language, even if it is a simple human-readable format, offers different ways to specify the same type of data. For example, string values can be expressed in five ways:

- plain scalar: the string does not need any quotes, but spaces at the start and at the end of the string are trimmed. Moreover, it can not contain escaped characters like '\n' and it can raise problems when this type of string contains special characters (like '&' or ':'), especially at the start of the string.
- single quoted: similar to plain scalar, the string can contain special characters, but it is not possible to specify escaped characters. To specify a quote character (''') in the string, it must be escaped with another quote character (e.g. ''').
- double quoted: these strings can contain escaped characters like '\n', double quote and backslash characters must be escaped (e.g. '\"' and '\\').
- literal block scalar: similar to plain scalar, but the string can extend on more than one line. The newline characters are kept in the resulting string.
- folded block scalar: similar to a literal block scalar, but the newline characters are not kept in the resulting string.

To test how a plugin is parsed by Staresc, it is sufficient to run Staresc specifying the plugin directory, and the flag "--test-plugins" instead of the connection string. In this mode, Staresc activates the debugging mode by default, and it prints all the information it parses from every plugin contained in the plugin directory.
This mode is also useful to test how the parsers (matcher or extractor)

of a plugin work. Indeed, the mechanism of the pipelined parsers can confuse the plugin developers. With this mode they can check if, given some testing results, these parsers work as expected.
To use this feature it is sufficient to add a field called `plugin_tests` to the elements of the list `tests` in the YAML file. This field is a list of "plugin_test" elements, each of which contains three fields: `stdout`, `stderr` and `expected`.
The idea behind these tests is the following, the command (field `command` in the test element) is executed on a hypothetical target machine, the results of the command are contained in the fields `stdout`, `stderr`. Naturally, we do not really execute the command on the target machine but `stdout` and `stderr` are used to create the "result object" that is passed to the list of parsers.
The field `expected` contains a boolean value. If it is true, Staresc assumes that the list of parsers must declare the test as positive, otherwise, it assumes that the parsers must declare the test as negative. A plugin developer can check if the parsers work as expected or not, given some testing snippets that could have been generated by a command on the target machines.
The following example shows some possible tests that can be applied to the sudoedit plugin. The Listing 4.4 contains the content of the field `tests` in the sudoedit plugin. This is similar to what we had in the plugin we showed before, but here, we specified some plugin tests. The first element of the `tests` list is the one that runs the command "sudoedit -s '0123456789\\'". From the resources about this vulnerability, we know that this command crashes the vulnerable sudo binaries, meanwhile it generates a "usage" message on patched sudo binaries.
Given these reasons, we added two tests for this `Test` object: the first contains the usage message in the `stderr` and it is expected to be marked as not-vulnerable (`expected`: false) by the relative parser. The second one contains the malloc crashing error message on the `stderr`, and it is expected to be marked as vulnerable by the parser. Similarly, we designed the two tests for the second `Test` object. The first `Test` contains, in the `stdout` field, the message printed by the command "sudo --version", with a patched version (the version 1.9.9). It is expected to not be marked as vulnerable by the parsers. The second one contains the stdout given by a patched sudo binary.

```yaml
tests:
  - command: "sudoedit -s '0123456789\\'"
    parsers:
      - parser_type: 'matcher'
        rule_type: 'word'
        condition: 'or'
        rules:
          - 'memory'
          - 'Error'
          - 'Backtrace'
          - 'malloc'
          - 'invalid pointer'
    plugin_tests:
      - stderr: |
          sudoedit: invalid option -- 's'
          usage: sudoedit -h | -V
          usage: sudoedit [-ABknS] [-r role] [-t type] [-C num] [-D directory] [-g group] [-h host] [-p prompt] [-R directory] [-T timeout] [-u user] file ...

        expected: False
      - stderr: "malloc(): invalid next size (unsorted)\n"
        expected: True
  - command: 'sudo --version'
    parsers:
      - parser_type: 'extractor'
        rule_type: 'regex'
        rules:
          - 'Sudo version .*\n'
      - parser_type: 'extractor'
        rule_type: 'regex'
        rules:
          - '([01].[012345678].[0-9]+)|(1.9.[01234])|(1.9.5p1)'
    plugin_tests:
      - stdout: |
          Sudo version 1.9.9
          Sudoers policy plugin version 1.9.9
          Sudoers file grammar version 48
          Sudoers I/O plugin version 1.9.9
          Sudoers audit plugin version 1.9.9

        expected: False
      - stdout: |
          Sudo version 1.8.31p2
          Sudoers policy plugin version 1.8.31p2
          Sudoers file grammar version 46
          Sudoers I/O plugin version 1.8.31p2
```

```
47          expected: True
```

Listing 4.4: Portion of the CVE-2021-3156 plugin with plugin testing fields.

Chapter 5

Comparison with other tools

We decided to develop Staresc because the tools that are available do not have the features that we want. Nessus and OpenVAS can perform authenticated vulnerability assessments, but their plugin language (NASL) is too complex. Nuclei has a flexible and intuitive YAML based language for its templates, but it can not perform vulnerability assessments over interactive shell protocols.
In this chapter, we analyse the differences and the common features between the actual Staresc implementation and four tools from which we took inspiration.The first is the old implementation of Staresc, that has a completely different mechanism to handle the plugins. The second is Nuclei, from which we took a lot of ideas for our YAML based plugin language. The third and the fourth are Nessus and OpenVAS, we focus a lot on the NASL scripting language analyzing its strengths and weaknesses.

5.1 Staresc Evolution

Staresc was originally ideated and developed by Bruno Colella. Then Valerio Casalino [1] improved the tool, and built the scaffold of the tool that we have now. In the last months, we changed a lot the features and the structure of the tool, but the core structure and the ideas under it are the same. It must take some plugins and some targets, run the tests specified by the plugins on the targets and then, it must print a report showing the vulnerable targets.
Two big parts of this process have been totally rewritten: the report production part and the plugin parsing part. We are going to discuss

the plugin parsing part below.
Regarding the first part, the older version of Staresc takes, as input, the name of a JSON file, and prints in this file a JSON object that contains all the information about the scan. Even if it is simpler than the actual one, this reporting implementation is less flexible, and can not be easily extended with additional reporting formats (like XLSX, CSV, etc.).

5.1.1 Plugin parsing evolution

The most important part that changed radically, is the plugin parsing one. Even if Staresc was developed taking example from Nuclei, its plugins were not written as YAML files. Indeed, originally they were implemented as Python modules, that were imported during the scan. There are four main reasons that lead us to change the format for our plugins and to adopt the YAML language:

- Python plugins are less readable than newer YAML plugins. With its easily readable structure, YAML language helps on keeping information cleaner. Due to its intuitive syntax, it is easy to understand the structure of the plugin.
- Python plugins require the developer to know Python. For a person that does not know Python or YAML, it is faster and easier to learn our plugin format based on YAML, rather than learn a whole new programming language.
- Python plugins can require to be updated if their version of Python reaches End Of Life.
- YAML plugins are easier to debug since they do not require to write control flow constructs and other complex statements that can lead to the introduction of bugs.

We developed Staresc with the goal to have a vulnerability assessment tool that takes advantage of a big pool of plugins. We believe that the best way to achieve this goal is to make the plugin development a fast and easy process, in order to encourage other people to write their own plugins and share them with the community.
In Table 5.1, we show a comparison between an old Python plugin and a newer YAML plugin. Both are developed to detect the sudo vulnerability of the CVE-2021-3156. The new plugin is taken from the example plugin shown in the Subsection 4.1.2. We stripped some

field from it, since this information was not contained in the Python plugin.

```python
# Commands to be executed on the target
COMMANDS = [
    'sudoedit -s "\\" 2>&1
    ',

    "sudo --version 2>&1"
]
# Matcher string for distribution/*nix
# https://docs.python.org/3.4/library/re.html
MATCHER = ".*"

def get_commands() -> list:
    return COMMANDS


def get_matcher() -> str:
    return MATCHER


def parse(output: list) -> str:
    if "memory" in output[0] or "Error" in
    output[0] or "Backtrace" in output[0]:

        retVal = "sudo -V\n"
        retVal += f"{output[1]}\n"
        retVal += "Is vulnerable to CVE
    -2021-3156"

        return retVal
    return "Not vulnerable"
```

```yaml
id: 'CVE-2021-3156-sudoedit'
author: 'cekout'
name: 'Component with known vulnerabilities: sudo
      before 1.9.5p2'

description: |
  Sudo before 1.9.5p2 contains an off-by-one
     error that can result in a

  heap-based buffer overflow, which allows
     privilege escalation to root via

  "sudoedit -s" and a command-line argument that
     ends with a single backslash

  character.
remediation: |
  Update the software to a more recent version
cve: 'CVE-2021-3156'
reference: 'https://nvd.nist.gov/vuln/detail/CVE
    -2021-3156'


cvss: 7.8
severity: 'High'
cvss_vector: 'CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:
    H/I:H/A:H'


tests:
  - command: "sudoedit -s '0123456789\\'"
    parsers:
      - parser_type: 'matcher'
        rule_type: 'word'
        condition: 'or'
        rules:
          - 'memory'
          - 'Error'
          - 'Backtrace'
          - 'malloc'
          - 'invalid pointer'
  - command: 'sudo --version'
    parsers:
      - parser_type: 'extractor'
        rule_type: 'regex'
        rules:
          - 'Sudo version .*\n'
      - parser_type: 'extractor'
        rule_type: 'regex'
        rules:
          - '([01].[012345678].[0-9]+)|
    (1.9.[01234])|(1.9.5p1)'
```

Table 5.1: Comparison between an old Staresc Python plugin (left) and a new Staresc YAMl plugin (right).

From the Table 5.1, you can see that the Python plugin must implement three functions. These functions are called by the engine in order to perform the scans and to parse the results.
These functions are:

- `get_commands`(): this function must return a list of strings, each string is a command that will be executed on the target machine.
- `get_matcher`(): this function is called at the start of a scan, it returns a string that is used as a regex to check if the plugin is compatible with the target machine's OS distribution.
- `parse`(): this function takes a list of strings that are the results (stdout and stderr) of the relative commands executed on the

target machine. As you can see, this function is used to parse these results and to decide if the target machine is vulnerable. It returns a string that marks the machine as vulnerable or not.

As you can see from the Table 5.1, `get_commands`() and `get_matcher`() simply return static strings. Thus, they can be easily replaced by static fields.
Moreover, the functionality that checks if the plugin is compatible with the OS of the target machine is temporarily removed, so, the field `distr_matcher` of the YAML plugin can be ignored for now.
On the other hand, we notice by our experience that most of the times, the `parse`() function just needs to check if a given regex or text appears in the result of the commands. So, we found a way to replace it with entities that accept static fields: the matchers and the extractors.

5.2 Comparison with Nuclei

Nuclei is a recent but solid tool, it owes his success to a very active community of developers that adds commits to the core engine almost every day.
Currently, more than 300 security researchers and engineers wrote more than 3000 plugins [16].
Its structure and its solidity are exactly what we are searching for our tool. Unfortunately, Nuclei supports a wide variety of network protocols, but not the interactive shell ones.
Given that, we tried to develop a tool that has a structure similar to Nuclei, but that is tailored to perform vulnerability assessment using the interactive shell protocols.
Nuclei's templates are rich of features: they can be used to send HTTP or DNS requests, to handle raw TCP connections, to check the content of local files and also to automate the behaviour of an internal headless browser used by the tool.
Many of these features are useless for our purposes and we do not need to implement them in Staresc. We decided to focus on the concept of parsing operators. Nuclei supports two types of operators: matcher and extractor.
Matchers are used to check if the response fulfills some conditions. There are six types of matcher, the common ones are: status, word and regex.
status is used to check if the response matches some given status codes, for example, if an HTTP response has the status code 200.

word matchers are used to check if the response contains a given portion of text. It can also be hex-encoded. regex matchers are similar to the word ones, they check if the response content matches a given regex.
Extractors are similar to matchers, with the difference that they extract portions of text from the response.

5.2.1 Comparison between Nuclei templates and Staresc plugins

From the Table 5.2 it is possible to see a comparison between Nuclei's matchers and Staresc's matchers.

```
matchers-condition: and
matchers:
  - type: word
    words:
      - "X-Powered-By: PHP"
      - "PHPSESSID"
    condition: or
    part: header

  - type: word
    words:
      - "PHP"
    part: body
```

```
parsers:
  - parser_type: 'matcher'
    rule_type: 'word'
    invert_match: true
    rules:
      - 'XBIzDRvndZWMTAqCbQVQXg'
  - parser_type: 'matcher'
    rule_type: 'regex'
    invert_match: true
    condition: 'or'
    rules:
      - '\nssl_tlsv1=[Yy][Ee][Ss]'
      - '^ssl_tlsv1=[Yy][Ee][Ss]'
      - '\nssl_sslv2=[Yy][Ee][Ss]'
      - '^ssl_sslv2=[Yy][Ee][Ss]'
      - '\nssl_sslv3=[Yy][Ee][Ss]'
      - '^ssl_sslv3=[Yy][Ee][Ss]'
```

Table 5.2: Comparison between Nuclei matchers (left) and Staresc matchers (right).

As you can see, the two types of matcher are very similar except for some slight differences. Nuclei supports the field `matcher-condition` that specifies how the results of the matchers must be merged. With an "and" value, all the matchers must get a match. Staresc does not support this field since it merges the results of the parsers in a different way. As we have shown in the Subsection 4.1.1, Staresc's parsers are placed in a pipeline-like structure, the result of the first parser is sent as input to the second parser and so on.
By default, this structure leads to a configuration in which the results of multiple matcher are merged, as if we had the field `matcher-condition` set to "and".
We discussed about the possibility to add a field similar to `matcher-condition` that lets the plugin developer modify how the results of the matchers must be merged. Eventually, we agreed that currently it is not a useful feature, since we noticed that we never needed it.
Another difference that is caused by our pipeline parsers structure is

the name of the field that contains the list of parsers. Nuclei's plugins contain the matchers' specification in a list, assigned to the field `matchers` (`extractors` for the extractors).
Staresc's parsers are listed in the more generic field `parsers`, with the parser type specified in the field `parser_type`, inside each parser.
This is due to the fact that we can mix matchers and extractors in the pipeline. So, by our point of view, matcher and extractors are different variants of parsers, and, as parsers, they must be placed in the same list.
Some of the more powerful and useful features that characterize Nuclei's templates, are the possibility to insert scripting-like statements that are executed in the template context, and the possibility to define workflows of templates.
Nuclei's template scriptings consists mostly on three features: DSL operators, helper functions and variables.

- DSL operators are a special type of matcher/extractor whose behaviour is specified by statements wrote in a scripting-like language. An example of DSL matcher can be found in the Listing 5.1.

- Helper functions are predefined functions that can be used by the plugin to apply transformation or to perform various checks. Plugin developers can use these functions to transform data before sending a request (e.g. `base64`() to encode part of the request), to decode part of the response (e.g. `base64_decode`()), to perform checks on the response (e.g. `len`() on the response body), etc.

- Variables can be introduced to define values that can be reused later in the plugin. They can be assigned static values or DSL expressions; in the latter case the expression must be enclosed in double-curly brackets. Moreover, there are some standard variables that are defined by the engine according to the context of the scan. For example, the variable `BaseURL` contains a string with the base URL of the target.
  In the Listing 5.2 is shown a Nuclei's template that sets two variables and sends the string "PING" on a TCP connection with the specified hostname. Then it reads 8 characters from the connection and checks (with a matcher) if they are equal to the string "hello", encoded with base64.

```
matchers:
  - type: dsl
    dsl:
      - "len(body)<1024 && status_code==200" # Body length less than 1024 and 200 status code
      - "contains(toupper(body), md5(cookie))" # Check if the MD5 sum of cookies is contained in the uppercase body
```

Listing 5.1: A Nuclei's DSL matcher that performs dynamic checks on a response.

```
# Variable example for network requests
id: variables-example

info:
  name: Variables Example
  author: pdteam
  severity: info

variables:
  a1: "PING"
  a2: "{{base64('hello')}}"

network:
  - host:
      - "{{Hostname}}"
    inputs:
      - data: "{{a1}}"
    read-size: 8
    matchers:
      - type: word
        part: data
        words:
          - "{{a2}}"
```

Listing 5.2: Nuclei variables used to give a dynamic behaviour to the template.

Workflows allow to define groups of templates that can be executed in series. This is particularly useful when we are targeting a specific technology, and want to execute all the template related to that technology. For example, if we are testing a web app that runs Wordpress [33], we can specify a workflow that contains a list of all the templates that target Worpress applications.
Another interesting feature of the workflow is the possibility to specify dependencies between the templates. It is very useful when we want to execute a template only under some conditions given by another template. For example, we could execute a template for a specific

technology only when we are sure that the given technology is used by the target system. In Listing 5.3 you can see an example workflow that executes a template to detect the technology used by the target system. Then, if the matcher named "vbulletin" matches (in the run template), two vbulletin exploit templates are executed. On the other hand, if the matcher named "jboss" matches, other two exploit templates, specific for jboss technology, are executed.

```
workflows:
  - template: technologies/tech-detect.yaml
    matchers:
      - name: vbulletin
        subtemplates:
          - template: exploits/vbulletin-exp1.yaml
          - template: exploits/vbulletin-exp2.yaml
      - name: jboss
        subtemplates:
          - template: exploits/jboss-exp1.yaml
          - template: exploits/jboss-exp2.yaml
```

Listing 5.3: An example of a workflow in a Nuclei's template.

5.3 Comparison with Nessus and OpenVAS

As we explained in the Subsection 2.1.1, Nessus and OpenVAS are two tools that share the core engine and a lot of features.
For our purposes, the most interesting features, that these two tools offer, are: the authenticated scans and the possibility to define new scans with the NASL language.
Nessus supports many authentication mechanisms, from classic HTTP login pages, to FTP and POP3.
These authentication mechanisms can be used to target both Unix-like and Windows systems. On the other hand, Staresc currently supports only SSH and Telnet protocols, that are usually used by Unix-like systems.
However, the structure of Staresc keeps separated the core engine and the connection part. This implies that, if the right connection is implemented, Staresc can be used to target Windows machines.
This is the idea that led us to the current implementation of the connection part: making the tool easily extendable with new connection types.

5.3.1 NASL language

The other important feature of Nessus and OpenVAS is the NASL language. It is a scripting language similar to C and Perl languages. We will explore how it works and how to write NASL plugins using two example plugins, the ones that OpenVAS uses to check the presence of the CVE-2021-3156 vulnerability.
Even if our first competitor is Nessus, we must use OpenVAS' plugins because Nessus' ones are not open source. We couldn't get the permission from Tenable to show the content of one of their CVE-2021-3156 plugins. However, the plugins used by Nessus are easily readable when the tool is installed (even in the free version). For example, in the Linux systems they are placed in the directory "/opt/nessus/lib/nessus/plugins" [37].
OpenVAS has two plugins that check the presence of the sudoedit vulnerability: one checks the version of the sudo binary, and the second checks if sudo crashes with a specific command, similar to the one that we use in Staresc's plugin. The content of the two plugins is shown in Listing 5.4 and Listing 5.5. To explain the NASL syntax, we use, as example, the plugin in Listing 5.4, then we focus on the differences between the first and the second plugin.
NASL plugins are generally divided into three parts [34]:

- Configuration
- Post configuration
- Security check

Configuration is the part of the plugins specifying all the metadata about the plugin and the checked vulnerability. It contains a lot of function calls that set static fields, like the plugin ID or the vulnerability severity.
In the example plugin, the configuration part goes from line 1 to line 41. On the first line, the CPE value is set. CPE stands for Common Platform Enumeration, it is a naming scheme used to identify information technology systems, software, and packages [18], Nessus and OpenVAS use this naming scheme to identify the products that they check in their plugins [39]. Here, the CPE identifies the sudo binary.
Lines 5-21 are easily understandable, they simply set some fields that are used by the core engine to give information about the plugin to the user.

Lines 21-23 are very important, here the developer specifies the dependencies of the plugin, and in which situation the plugin must be executed.
The function `script_dependencies`() is used to specify the plugin that must be executed before the actual one. Here, the plugin "gb_sudo_ssh_login_detect.nasl" checks, using SSH, the presence of the sudo binary.
To check if the plugin "gb_sudo_ssh_login_detect.nasl" has found the sudo binary, the sudoedit plugin uses the "keys".
These keys are similar to global variables that can be set or not. With the function `script_mandatory_keys`() the developer can ensure that some keys are set when the plugin runs. The function `script_exclude_keys`() does the opposite of the previous one, it checks if some keys are not set.
In lines 22-23 of the sudoedit plugin, the developer ensures that the key "sudo/ssh-login/detected" is set and the key "ssh/force/pty" is not set. "sudo/ssh-login/detected" key is set by the plugin "gb_sudo_ssh_login_detect.nasl", and it ensures that the sudo binary has been found. "ssh/force/pty", instead, is activated if the tool detects that a pseudoterminal, or a PTY, is used to send the commands.
Lines 25-41 contain the specification of other static fields.
Lines 44-46 contain the post configuration part. In this part the plugin developer includes the headers of the NASL libraries that the plugin uses. This mechanism is similar to the "include" mechanism used in the C language.
The library "host_details.inc" provides functions that can be used to retrieve information about the target host. For example, the function `get_app_version_and_location`() is provided by this library, and can be used to retrieve the location and the version of a given application in the target host.
The library "revision-lib.inc" provides helper functions for revision strings.
The library "version_func.inc" provides functions that can be used to handle version numbers. For example, it contains the function `version_is_less`().
The last part of the plugin is the security check one. It contains the real logic of the plugin: the security checks that the plugin makes, on the target systems.
On lines 48-52 the plugin retrieves the version and the location of the sudo binary. If no version is found, the function exits, since the `exit_no_version` parameter is set to "TRUE". It can be seen that

the CPE variable is used to uniquely identify the sudo binary.
In line 54-55 there is a first version check, in which the plugin checks if the binary version is less than the least vulnerable version. If the binary version is too low to be vulnerable, sudo is considered not vulnerable, and the plugin exits.
Notice that the exit value is 99, looking at the OpenVAS source code [12], we can affirm that this value corresponds to the constant "NASL_EXIT_NOTVULN", that is used by the OpenVAS engine to report a not vulnerable target.
On lines 57-60 there is a second check. Here the plugin checks that the version of the sudo binary is less than the lower patched version. If so, the binary is considered vulnerable, a security report is generated (lines 58-59) and the program exits.
On line 63 there is an additional exit call, that is reached if the version of the sudo binary is one of the patched ones. This call to the exit function marks the target as not vulnerable (exit code 99).
The active plugin has a similar configuration part. The only differences are the category of the plugin (line 18), the dependencies (lines 21) and the required keys (line 22).
The category of this plugin is "ACT_ATTACK", while the category of the previous plugin is "ACT_GATHER_INFO" (line 18).
These categories are used by the core engine to organize the plugins, this is useful when the user wants to restrict in some way the plugins to use, in a security scan. For example, assume to be in a situation in which you want to perform only version checks, because the target of the scan is a critical infrastructure. You would appreciate the possibility to not perform checks that can compromise the integrity of the target system.
The different dependencies are due to the PoC command that this plugin executes. Since it uses the Perl language, the plugin checks that the Perl binary is detected by the "gb_perl_ssh_login_detect.nasl" plugin.
From the post configuration part, you can notice that the active plugin includes the "ssh_func.inc" library. It provides functions that can be used to handle SSH connections to the target system.
The security check part is completely different from the one of the previous plugin.
On lines 46-47 the plugin checks if the sudo binary is present.
On lines 49-51 it retrieves a socket-like object that can be used to handle the SSH session on the target system.
On line 56-59 it checks that the SSH session has access to the bi-

nary. To do this, the plugin checks that the output of the command "sudoedit --help" is as expected.
On lines 63-65 the plugin checks that the sudo binary does not throw a segmentation fault in normal conditions. This check is performed to avoid false positives. In fact, without it, we could not know if the segmentation fault is generated by the CVE-2021-3156 PoC or not.
On lines 68-69 the official PoC is executed on the target system. Then, on line 70, the socket-like object is closed.
On lines 72-75 the result of the PoC command is checked, if it matched the segmentation fault pattern, the host is considered vulnerable.

5.3.2 Comparison between NASL plugins and Staresc plugins

The first difference between our example Staresc's plugin (Listing 4.3) and the OpenVAS' plugins (Listings 5.4 and 5.5) is the number of lines. The length of the Staresc's plugin is around 40 lines, while OpenVAS' version check plugin is around 70 lines long, and the active one is around 55 lines long (without blank lines and comments).
These numbers cannot be directly used to compare the two types of plugins, since OpenVAS' plugins consist mainly on configuration statements, that can be seen as the metadata fields (`name`, `description`, etc.) of Staresc's plugins.
So, if we compare only the security check parts, we can see that the active check of the Staresc's plugin takes around ten lines, while it takes around 20 lines in the OpenVAS' plugin.
This is basically the same check. We choose to not use the Perl PoC because it can not be used in target systems that do not support the Perl language. On the other hand, OpenVAS' plugin performs an additional check (lines 81-83) to establish if the sudo binary crashes without the PoC. We choose to not implement this check because we did not consider it necessary for our use cases.
However, this additional check adds only three lines. Thus, Staresc test takes less space than OpenVAS security check.
Looking at the version check plugin, its security check part is around 10 lines long, as the one of Staresc's plugin.
Regardless of the plugin length, it is clear that Staresc's plugin is easier to read and understand. The fact that it is written in YAML makes its structure clearer than that of OpenVAS' plugins. It is easy to notice how many tests, and which commands, will be executed. On the other hand, it requires a bit of knowledge to understand what the

parsers are and how they handle the command results.
However, it requires less effort to read how the parsers work instead of studying a new scripting language.
To understand how these two OpenVAS' plugins work, we had to read NASL documentation [38] [41] and sometimes the source code [12]. Moreover, things become harder if we want to read Nessus plugins, since Nessus source code is not readable and there is little or no documentation for its libraries.
If understanding existing NASL plugins is hard, write new plugins is even harder. It is a very time consuming activity, because it requires the developer to know which libraries and which function are available.
On the other hand, Staresc does not require the knowledge of libraries and functions to read and write plugins. This makes Staresc's plugins less, flexible because we can not handle the security checks directly in code, but we believe that this flexibility requires too much complexity, that makes plugins development a too much time consuming process. The right compromise could be represented by Nuclei's templates that support inline template scripting.
Another difference is given by the number of plugins used to check the same vulnerability. OpenVAS uses two plugins to perform two different checks, while we implemented a single Staresc's plugin for the same purpose.
This difference can be given by the fact that OpenVAS distinguishes the two plugins by their categories. In this way, it can choose which test to perform based on the risk level that the tests have.
Unfortunately Staresc does not have this feature, we could implement it introducing categories or tags in the plugins, but this would imply that the two tests should be split in two plugins, and this is not ideal. Another solution could be to introduce a `risk_level` field inside the test elements. In this way, Staresc could choose which plugins to run based on the maximum risk level of the scan.
Nessus plugins for the CVE-2021-3156 are even more complex, there are 76 plugins relative to this vulnerability [14]. Apparently, Nessus has one plugin for each OS distribution that it supports. Probably, this behaviour is given by the fact that its plugins check the version of the sudo binary using library functions that are specific for the target distribution. Thus, for each distribution, Nessus plugins have a different way to check the sudo binary version.
Furthermore, we found that Nessus plugins do not perform the active check, trusting only the version check. This can lead to an higher

number of false positives respect to OpenVAS and Staresc, since sometimes the sudo binary is patched, even if it has a vulnerable version. Given that, we can say that NASL plugins are more flexible and can lead to more accurate results, but this accuracy brings a not negligible complexity that plugin developers must face up. Nessus and OpenVAS are more user-friendly than developer-friendly.
On the other side, Staresc has less flexible and less precise plugins, that are easier to implement and are devised to be more developer-friendly than user-friendly.

```
CPE = "cpe:/a:sudo_project:sudo";

if(description)
{
  script_oid("1.3.6.1.4.1.25623.1.0.117186");
  script_version("2022-08-09T10:11:17+0000");
  script_xref(name:"CISA", value:"Known Exploited Vulnerability (KEV) catalog");
  script_xref(name:"URL", value:"https://www.cisa.gov/known-exploited-vulnerabilities-catalog");
  script_cve_id("CVE-2021-3156");
  script_tag(name:"cvss_base", value:"7.2");
  script_tag(name:"cvss_base_vector", value:"AV:L/AC:L/Au:N/C:C/I:C/A:C");
  script_tag(name:"last_modification", value:"2022-08-09 10:11:17 +0000 (Tue, 09 Aug 2022)");
  script_tag(name:"severity_vector", value:"CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H");
  script_tag(name:"severity_origin", value:"NVD");
  script_tag(name:"severity_date", value:"2021-07-20 23:15:00 +0000 (Tue, 20 Jul 2021)");
  script_tag(name:"creation_date", value:"2021-01-27 06:47:49 +0000 (Wed, 27 Jan 2021)");
  script_name("Sudo Heap-Based Buffer Overflow Vulnerability (Baron Samedit) - Version Check");
  script_category(ACT_GATHER_INFO);
  script_copyright("Copyright (C) 2021 Greenbone Networks GmbH");
  script_family("Buffer overflow");
  script_dependencies("gb_sudo_ssh_login_detect.nasl");
  script_mandatory_keys("sudo/ssh-login/detected");
  script_exclude_keys("ssh/force/pty");

  script_xref(name:"URL", value:"https://www.sudo.ws/stable.html#1.9.5p2");
  script_xref(name:"URL", value:"https://blog.qualys.com/vulnerabilities-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit");

  script_tag(name:"summary", value:"Sudo is prone to a heap-based buffer overflow dubbed 'Baron Samedit'.");

  script_tag(name:"vuldetect", value:"Checks if a vulnerable version is present on the target host.");
```

```
32    script_tag(name:"insight", value:"Sudo is allowing privilege
        escalation to root via 'sudoedit -s' and a command-line argument
        that ends with a single backslash character.");

34    script_tag(name:"affected", value:"All legacy versions from 1.8.2 to
        1.8.31p2 and all stable versions from 1.9.0 to 1.9.5p1 in their
        default configuration.");

36    script_tag(name:"solution", value:"Update to version 1.9.5p2 or later
        .");

38    script_tag(name:"qod_type", value:"executable_version_unreliable");
39    script_tag(name:"solution_type", value:"VendorFix");

41    exit(0);
42  }

44  include("host_details.inc");
45  include("revisions-lib.inc");
46  include("version_func.inc");

48  if( ! infos = get_app_version_and_location( cpe:CPE, exit_no_version:
        TRUE ) )
49    exit( 0 );

51  vers = infos["version"];
52  path = infos["location"];

54  if( version_is_less( version:vers, test_version:"1.8.2" ) )
55    exit( 99 ); # nb: Not affected

57  if( version_is_less( version:vers, test_version:"1.9.5p2" ) ) {
58    report = report_fixed_ver( installed_version:vers, fixed_version:"
        1.9.5p2", install_path:path );
59    security_message( port:0, data:report );
60    exit( 0 );
61  }

63  exit( 99 );
```

Listing 5.4: OpenVAS's CVE-2021-3156 plugin that checks the sudo version.

```
1   CPE = "cpe:/a:sudo_project:sudo";

3   if(description)
4   {
5     script_oid("1.3.6.1.4.1.25623.1.0.117187");
6     script_version("2022-08-09T10:11:17+0000");
7     script_xref(name:"CISA", value:"Known Exploited Vulnerability (KEV)
        catalog");
8     script_xref(name:"URL", value:"https://www.cisa.gov/known-exploited-
        vulnerabilities-catalog");
9     script_cve_id("CVE-2021-3156");
10    script_tag(name:"cvss_base", value:"7.2");
11    script_tag(name:"cvss_base_vector", value:"AV:L/AC:L/Au:N/C:C/I:C/A:C
        ");
```

```
12    script_tag(name:"last_modification", value:"2022-08-09 10:11:17 +0000
        (Tue, 09 Aug 2022)");
13    script_tag(name:"severity_vector", value:"CVSS:3.1/AV:L/AC:L/PR:L/UI:
        N/S:U/C:H/I:H/A:H");
14    script_tag(name:"severity_origin", value:"NVD");
15    script_tag(name:"severity_date", value:"2021-07-20 23:15:00 +0000 (
        Tue, 20 Jul 2021)");
16    script_tag(name:"creation_date", value:"2021-01-27 06:47:49 +0000 (
        Wed, 27 Jan 2021)");
17    script_name("Sudo Heap-Based Buffer Overflow Vulnerability (Baron
        Samedit) - Active Check");
18    script_category(ACT_ATTACK);
19    script_copyright("Copyright (C) 2021 Greenbone Networks GmbH");
20    script_family("Buffer overflow");
21    script_dependencies("gb_sudo_ssh_login_detect.nasl", "
        gb_perl_ssh_login_detect.nasl");
22    script_mandatory_keys("sudo/ssh-login/detected", "perl/ssh-login/
        detected"); # nb: PoC below requires perl to be installed on the
        target.

24    script_xref(name:"URL", value:"https://www.sudo.ws/stable.html#1.9.5
        p2");
25    script_xref(name:"URL", value:"https://blog.qualys.com/
        vulnerabilities-research /2021/01/26/cve-2021-3156-heap-based-
        buffer-overflow-in-sudo-baron-samedit");

27    script_tag(name:"summary", value:"Sudo is prone to a heap-based
        buffer overflow dubbed 'Baron Samedit'.");

29    script_tag(name:"vuldetect", value:"Runs a specific SSH command after
         the login to the target which is known to trigger an error message
         on affected versions of Sudo.");

31    script_tag(name:"insight", value:"Sudo is allowing privilege
        escalation to root via 'sudoedit -s' and a command-line argument
        that ends with a single backslash character.");

33    script_tag(name:"affected", value:"All legacy versions from 1.8.2 to
        1.8.31p2 and all stable versions from 1.9.0 to 1.9.5p1 in their
        default configuration.");

35    script_tag(name:"solution", value:"Update to version 1.9.5p2 or later
        .");

37    script_tag(name:"qod_type", value:"exploit");
38    script_tag(name:"solution_type", value:"VendorFix");

40    exit(0);
41  }

43  include("ssh_func.inc");
44  include("host_details.inc");

46  if( ! get_app_location( cpe:CPE, port:0, nofork:TRUE ) )
47    exit( 0 );

49  sock = ssh_login_or_reuse_connection();
50  if( ! sock )
```

```
51    exit( 0 );

53  # nb: We're only testing the "sudoedit" within the path as others might
        be not allowing to e.g. get root.

55  # just exit if we don't have access to the binary...
56  cmd = "sudoedit --help";
57  res = ssh_cmd( socket:sock, cmd:cmd, nosu:TRUE );
58  if( ! res || "usage: sudoedit" >!< res )
59    exit( 0 );

61  # or avoid any false positives if the binary itself is
62  # throwing a segmentation fault..
63  pattern = "(malloc\(\): corrupted top size|Segmentation fault)";
64  if( egrep( string:res, pattern:pattern, icase:FALSE ) )
65    exit( 0 );

67  # sudoedit -s '\' `perl -e 'print "A" x 65536'`
68  cmd = "sudoedit -s '\' `perl -e 'print " + '"A"' + " x 65536'`";
69  res = ssh_cmd( socket:sock, cmd:cmd, nosh:TRUE, nosu:TRUE,
        return_errors:TRUE, return_linux_errors_only:TRUE, pty:TRUE,
        clear_buffer:TRUE );
70  close( sock );

72  if( egrep( string:res, pattern:pattern, icase:FALSE ) ) {
73    report = "Used command: " + cmd + '\n\nResult: ' + res;
74    security_message( port:0, data:report );
75    exit( 0 );
76  }

78  exit( 99 );
```

Listing 5.5: OpenVAS's CVE-2021-3156 plugin that checks if sudo crashes with the PoC command.

Chapter 6

Future work

Even though it has passed its first release, Staresc is still a young project and is far from its definitive shape.
From the previous chapters, many possible improvements have emerged. In this section we discuss about the benefits of these improvements and how it would be possible to implement them.

6.0.1 Tags

The most simple, and quick, feature to implement is the addition of the tags to the plugins. The tags could be used to categorize the plugins.
With this grouping, it would be possible to configure more complex scans, specifying to use only the plugins that have certain tags and excluding plugins that have some other tags.
Till now, we never needed to implement this feature, since we run all the plugins at each scan. But, with the increasing number of plugins, we will eventually be in the situation in which we do not want to use all of them. Indeed, with a great number of plugins, it is likely that many of them may be useless or not suitable to the target systems that we want to scan.
Tags could be easily implemented, adding a field `tags` in the YAML plugins. It could be a list of strings, or a single string that contains all the tags separated by white spaces.
Moreover, we could implement some command-line flags to let the user specify how to filter the plugins to use, basing on the tags. The flag "--mandatory-tags" could be used to specify the tags that the plugins must contain, while the flag "--exclude-tags" could be used for the opposite purpose.

6.0.2 Support Windows connections

Another useful improvement is the support for more types of connections. Currently, only SSH and Telnet are supported. This limits our possible targets to mostly Unix-like systems.
OpenVAS and Nessus support a wide variety of protocols, that cover many different types of systems. For our purposes, it would be sufficient (at the moment) to support only the most common protocols used to execute commands on Windows remote hosts.
One of the most used protocols is the Server Message Block protocol (SMB) [52], that is a client-server message protocol. SMB is used by many Windows services to implement remote shells, one example is PsExec [51].
Given the easily extendable structure of the Staresc's connection part, and the presence of Python libraries that allow to communicate with a PsExec service on the target machines [25], it should not be difficult to implement the support for Windows connections.

6.0.3 Workflows

Looking at Nuclei (Subsection 2.1.2 and Section 5.2), we noticed a very useful feature, the workflows.
Workflows allow users to define an execution sequence of templates [32], they can be used to build group of template that must be executed sequentially. Moreover, they allow to define subtemplates.
Subtemplates are templates executed dependently on the result of other templates. In this way, Nuclei users can use workflow to specify in which cases these templates must be executed. Section 5.2 contains a more detailed explanation of Nuclei workflows.
Implementing workflows in Staresc could allow us to manage in a better way the plugins. We could use the workflows to define groups of plugins, that would be executed only in certain circumstances.
For example, consider a group of plugins that perform various checks on Apache (e.g. is it using a vulnerable version of Log4J? Does it use basic authentication over unencrypted HTTP?). To avoid useless checks that could lead to false positives, we would execute a preliminary plugin that checks if the Apache service is running. If this plugin reports that Apache is running, then we could execute the group of subplugins targeting it.
To implement workflows on Staresc, we should modify the plugin parsing part and the core part. The first should check if a field like `workflows` is contained in the YAML file that is loaded. Moreover,

it should parse all the structure contained by the field (included the dependencies between plugins and subplugins), and build a data structure that keeps the series of plugins and their relative subplugins.
The core part should run the plugins specified in the workflows and, if necessary, run the relative subplugins.
One related feature, that we should implement to support the workflows, is the named parsers. Indeed, Nuclei subtemplates are executed if a given matcher of the template returns a positive result. To refer to the matcher, they use its name (e.g. vbulletin in Listing 5.3). This feature should be easy to implement, just adding a field to the parsers.

6.0.4 Plugin scripting

Nuclei templates are not simple YAML files. They can contain snippets of a scripting-like language that the engine will execute during the scan.
This scripting language, together with variables and helper functions, allows to extends a lot the capabilities of templates, allowing them to specify a dynamic behavior that changes based on scan conditions.
Scripting-like code can be inserted into Nuclei templates as a string into the DSL type operators (matchers or extractor), or enclosed by double brackets.
Staresc's plugins can benefit a lot from the introduction of this feature, it could be possible to adjust the commands relying on the results of previous commands. For example, consider a plugin that checks if the configuration file of the SSH service contains any misconfiguration.
The default configuration file has the path "/etc/ssh/sshd_config", but it can happen that a custom configuration file is used. To see which configuration file is loaded by SSH service, we could check the command executed to launch the service (e.g. with the command "ps aux").
So, if we want to be sure to check the right configuration file, we could execute a command to extract the path of the configuration file and put it into a variable. Then, we could use the value, kept by the variable, in order to create the command that inspect the configuration file.
The implementation of this feature in Staresc can be challenging, one possible solution could be the use of libraries like Jinja [9]. These libraries are templating engines that take portion of files enclosed by special characters (e.g. double brackets), and interpret them as Python-like code.

Jinja can help us parsing the scripting code contained in the plugins. Moreover, it allows to define the environment in which the scripting code is executed, helping us to handle the variables and the helper functions that can be used by the plugin.

6.0.5 Porting in Go

Go is an open-source programming language developed by Google and released for the first time in 2009.
Go was designed by taking inspiration for the productivity and relative simplicity of Python, with the ability of C. [45] It is a compiled language and it is statically typed. Moreover, it uses the so-called "goroutines", lightweight threads managed by the Go runtime. Goroutines are generally faster than regular threads used by other languages, furthermore, Go's syntax makes it easier to write parallel code.
Regarding memory corruption vulnerabilities, Go is considered safer than C and C++. Since it is a compiled language, it does not require a Virtual Machine or an interpreter, making it faster than languages like Java or interpreted ones (Javascript, Ruby, etc.).
Another useful feature of Go is its building process. It can support a wide variety of platforms and architectures, and the building process is very straightforward. To specify the target platform and architecture, it is sufficient to set the environment variable "GOOS" "GOARCH" before running the command "go build -o <output_file>". [47]
Staresc could benefit a lot from a porting in Go.
First of all, it would perform faster scans, since it performs a lot of parallel work. Indeed, the scans run in parallel using Python futures [4], but Python does not have any built-in concurrency mechanism, while Go has been designed to support concurrency.
Furthermore, many times we had to deploy Staresc on machines that are not connected to the Internet (e.g. for scans on internal networks). We had a lot of trouble installing Python and all the libraries required by Staresc. It is possible to compile a Python program into a binary file and then copy the file to the machine, but it is a rather complex procedure, since Python is an interpreted language [8]. On the other hand, the Go building process can easily generate binary files for a lot of platforms and architectures, making the deployment of Staresc in these machines very straightforward.
One possible downside from the porting of Staresc in Go, is the library support. Python is much more widespread than Go, so, it can count on a larger number of libraries. This can be a problem, especially for

the libraries that we use to interact with the remote shells. However, we verified that libraries for SSH and Telnet are implemented, and we believe that, since Go is widely used for network tools, libraries for other interactive shell protocols are mostly available.

6.0.6 Web interface and integration with Nuclei

One of the pros of Nessus and OpenVAS is the web interface. It allows to configure the scans in an intuitive way and it provides a clear view of the reports of the scans.
Web interface would make the usage of Staresc easier, avoiding the user the burden of defining manually all the flags. Moreover, it can provide the saving of scanning configuration, making the usage of the tool less time-consuming.
The implementation of the web interface requires a lot of effort, it is a time-consuming task, especially for the frontend part. We plan to keep the engine separated from the web interface. In this way, users can choose to use either the original Staresc from command-line or the most intuitive web interface.
This structure permits to introduce the new interface without giving up the possibility to deploy Staresc on machines that can not expose web interfaces (e.g. machines reachable only with SSH).
In our ideal structure, web interface would be used to configure the scans and to render the templates. The configurations would contain three main parts: the plugins to use, the connection strings and the other flags of Staresc (e.g. debug). The web interface should use the command-line flags to launch the right scan, just executing Staresc from command-line.
For the report rendering part, the web interface should parse the generated report and show, on the web page, the various results. We could also add a new reporting format to the engine, to make this process easier.
We are also considering the support for Nuclei scans in the web interface. In this way, we could allow users to configure, launch and monitor scans for a wide variety of protocols from a single place. We would create a framework that benefits from the capabilities of Nuclei and Staresc, and that offers an intuitive web GUI. It could be a valuable competitor for Nessus and OpenVAS.

Chapter 7

Conclusion

This work presented Staresc, a new tool that performs vulnerability assessment over interactive shell protocols. It analyzed tools developed for the same purposes (Nessus and OpenVAS) and another tool that applies many interesting ideas to a different type of vulnerability assessment (Nuclei). Both pros and cons of these tools have been covered, showing which features and ideas could be applied to our implementation.
Staresc's software structure has been extensively analyzed, separating the various parts that compose the tool, and explaining the motivation under the principal implementation choices. This work showed how the four main parts of Staresc have been designed in a modular way, making it possible to edit one part, performing little to no changes on the other parts. In particular, it demonstrates that the implementation of new connection protocols, and reporting formats, can be made in a simple way, without modifying the core and the plugins parsing parts.
Plugins' syntax has been analyzed, showing which type of scan can be implemented, and discussing the technical issues and advantages that their development could bring. The plugins' analysis contained also a practical example of the development of a plugin that has been used in real vulnerability assessments.
Moreover, the work contains a comparison between Staresc and the main competitors. Our tool has been compared with its first version, covering the evolution of the plugin syntax. Then, Staresc's plugin syntax has been compared with Nuclei's templates syntax, showing which ideas lead our implementation and which factors lead to technical difference between the two syntaxes. Furthermore, we explained the basic syntax and structure of NASL plugins, comparing two OpenVAS plugins with the analogous Staresc' plugin.

Lastly, we covered the possible improvements that could make Staresc a faster, more flexible, precise and intuitive tool.
Staresc is an open-source project tailored to be easily extendable, and developed following many ideas introduced by recent tools. Even if it does not offer the same solidity of its major competitors, we hope that it could make vulnerability assessments, over interactive shell protocols, less time-consuming, allowing every security operator to rapidly develop and share its own plugins.

Bibliography

[1] 5amu. https://github.com/5amu. Accessed: 2022-09-05.

[2] Browse Vulnerabilities By Date. https://www.cvedetails.com/browse-by-date.php. Accessed: 2022-09-08.

[3] carlospolop. https://github.com/carlospolop. Accessed: 2022-08-30.

[4] concurrent.futures — Launching parallel tasks - Python. https://docs.python.org/3/library/concurrent.futures.html. Accessed: 2022-09-11.

[5] CVE-2021-3156. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156. Accessed: 2022-08-31.

[6] CVE system. https://cve.mitre.org/. Accessed: 2022-09-08.

[7] Exploring the Origins and Evolution of Vulnerability Management. https://blog.igicybersecurity.com/origins-and-evolution-of-vulnerability-management. Accessed: 2022-09-08.

[8] Is there a way to compile a python application into static binary? https://stackoverflow.com/questions/39913847/is-there-a-way-to-compile-a-python-application-into-static-binary. Accessed: 2022-09-11.

[9] Jinja - Jinja Documentation (3.1.x). https://jinja.palletsprojects.com/en/3.1.x/. Accessed: 2022-09-10.

[10] Main in the middle (MITM) attack. https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/. Accessed: 2022-08-31.

[11] MITRE. https://www.mitre.org/. Accessed: 2022-09-08.

[12] NASL_EXIT_NOTVULN definition. https://github.com/greenbone/openvas-scanner/blob/ca12c694e1fd162ac0386e6ec47b9a5925c8e01a/nasl/nasl_misc_funcs.c#L53. Accessed: 2022-09-04.

[13] Nessus. https://www.tenable.com/products/nessus. Accessed: 2022-08-30.

[14] Nessus CVE-2021-3156 plugins. https://www.tenable.com/cve/CVE-2021-3156/plugins. Accessed: 2022-09-04.

[15] Nuclei. https://github.com/projectdiscovery/nuclei. Accessed: 2022-08-30.

[16] Nuclei Templates. https://github.com/projectdiscovery/nuclei-templates. Accessed: 2022-09-05.

[17] Nuclei templating guide. https://nuclei.projectdiscovery.io/templating-guide/. Accessed: 2022-08-30.

[18] Official Common Platform Enumeration (CPE) Dictionary. https://nvd.nist.gov/Products/CPE. Accessed: 2022-09-03.

[19] OpenVAS. https://www.openvas.org/. Accessed: 2022-08-30.

[20] OpenVAS github repo. https://github.com/greenbone/openvas-scanner. Accessed: 2022-09-03.

[21] PEASS-ng. https://github.com/carlospolop/PEASS-ng. Accessed: 2022-08-30.

[22] Pexpect. https://pexpect.readthedocs.io/en/stable/. Accessed: 2022-09-05.

[23] ProjectDiscovery. https://github.com/projectdiscovery. Accessed: 2022-08-30.

[24] Python. https://www.python.org/. Accessed: 2022-09-11.

[25] Python PsExec Library. https://pypi.org/project/pypsexec/. Accessed: 2022-09-10.

[26] Security advisories. software flaws found by qualys. https://www.qualys.com/research/security-advisories/. Accessed: 2022-08-31.

[27] SSH Protocol – Secure Remote Login and File Transfer. https://www.ssh.com/academy/ssh/protocol. Accessed: 2022-08-31.

[28] Staresc source code. https://github.com/staresc/staresc. Accessed: 2022-09-05.

[29] The Go Programming Language. https://go.dev/. Accessed: 2022-09-11.

[30] Vulnerability Assessment. https://www.imperva.com/learn/application-security/vulnerability-assessment/. Accessed: 2022-09-08.

[31] What is Sudo? https://www.sudo.ws/. Accessed: 2022-08-31.

[32] Workflows. https://nuclei.projectdiscovery.io/templating-guide/workflows/. Accessed: 2022-09-10.

[33] Worpress. https://wordpress.com/it/. Accessed: 2022-09-05.

[34] Writing NASL scripts. https://dl.packetstormsecurity.net/papers/general/Writing_nasl_scripts.pdf. Accessed: 2022-09-03.

[35] YAML: YAML Ain't Markup Language™. https://yaml.org/. Accessed: 2022-09-05.

[36] Telnet Protocol Specification. RFC 854, May 1983.

[37] Location of Plugin Directory. https://community.tenable.com/s/article/Location-of-Plugin-Directory, October 2021.

[38] Michel Arboi. The NASL2 reference manual. http://michel.arboi.free.fr/nasl2ref/nasl2_reference.pdf, April 2005.

[39] Paul Asadoorian. Common Platform Enumeration (CPE) with Nessus. https://www.tenable.com/blog/common-platform-enumeration-cpe-with-nessus, May 2010.

[40] Raj Chandel. Linux Privilege Escalation using SUID Binaries. https://www.hackingarticles.in/linux-privilege-escalation-using-suid-binaries/, May 2018.

[41] Renaud Deraison. The Nessus Attack Scripting Language Reference Guide (incomplete). https://student.ing-steen.se/java/javacoding/toys/more_toys/nessus/txtfilez/nasl.html, September 1999.

[42] Laura Fitzgibbons. Telnet. https://www.techtarget.com/searchnetworking/definition/Telnet, September 2021.

[43] Park Foreman. *Vulnerability Management*. Auerbach Publications, USA, 1st edition, 2009.

[44] Kamil Gierach-Pacanek. Tools analysis: linPEAS obtaining, usage and alternatives. https://blog.cyberethical.me/linpeas, May 2021.

[45] Alexander S. Gillis. What is the Go Programming Language. https://www.techtarget.com/searchitoperations/definition/Go-programming-language, May 2020.

[46] Rhett Glauser. A History of the Vulnerability Management Lifecycle. https://vulcan.io/blog/a-history-of-vulnerability-management/, March 2019.

[47] Gaurav Kamathe. https://opensource.com/article/21/1/go-cross-compiling. , January 2021.

[48] Himanshu Kathpal. CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit). https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit, January 2021.

[49] Tim Keary. Nessus vs OpenVAS. https://www.comparitech.com/net-admin/nessus-vs-openvas/, April 2022.

[50] Chris M. Lonvick and Sami Lehtinen. The Secure Shell (SSH) Protocol Assigned Numbers. RFC 4250, January 2006.

[51] Mark Russinovich. PsExec v2.40. https://docs.microsoft.com/en-us/sysinternals/downloads/psexec, July 2022.

[52] Robert Sheldon and Jessica Scarpati. Server Message Block protocol (SMB protocol). https://www.techtarget.com/searchnetworking/definition/Server-Message-Block-Protocol, August 2021.