Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Developing a Software for Automated Module-based Configuration of Virtual Machines for Penetration Testing

Abschlussarbeit

zur Erlangung des akademischen Grades:

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Angewandte Informatik*

1. Gutachter: Prof. Dr.-Ing. Piotr Wojciech Dabrowski
2. Gutachter: Dr. rer. nat. Tom Ritter

Eingereicht von Nader Alhalabi [561121]

Datum

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den

Nader Alhalabi

Danksagung

Ich bedanke mich bei allen Personen die mich während meines Studiums und besonders bei der Erstellung dieser Arbeit unterstützt haben. Ich bedanke mich bei meinen Betreuern Herrn Prof. Dr. Piotr Wojciech Dabrowski und Herrn Dr. Tom Ritter für die Unterstützung während der Arbeit und die Bereitstellung von technischen Ressourcen.

Zusammenfassung

Die Beherrschung von Penetrationstests kann eine schwierige Aufgabe sein, da so viele Informationen aufgenommen werden müssen und so viele Schwachstellen zu lernen und zu analysieren sind. Penetrationstester müssen dieses Wissen auch in der Praxis anwenden, und virtuelle Maschinen können eine der besten Umgebungen sein, um die Fähigkeiten und das Verständnis eines Penetrationstesters anzuwenden, aber das Einrichten einer VM, um viele Schwachstellen zu üben, kann eine schwierige und zeitraubende Aufgabe sein.
Diese Arbeit zielt darauf ab, die Hindernisse bei der Einrichtung einer VM zu beseitigen, indem eine Software implementiert wird, die diese Aufgabe automatisiert. Diese Software sollte in der Lage sein, eine Liste von benutzerdefinierten Modulen zu installieren, die frei konfigurierbar sind, um eine bestimmte Schwachstelle auf der VM einzurichten, wodurch diese VM zu einer einsatzbereiten Sandbox für Penetrationstests und Hacking wird.

Die Implementierung dieser Arbeit - genannt "auto-hackbox" - ist in folgendem Repository verfügbar:

`https://github.com/nader-alhalabi/auto-hackbox`

Abstract

Penetration Testing can be a difficult skill to master, with so much information to absorb and so many vulnerabilities to learn and analyze. Penetration testers also need to practice this knowledge, and virtual machines can be one of the best environments to apply the skills and understanding of a penetration tester, but setting up a VM to exercise many vulnerabilities can be a tough and time-consuming job.
This work aims to remove the obstacles of setting up a VM, by implementing a software that automates this task, this software should be able to install a list of user-defined modules, which are freely configurable for the purpose of setting up a specific vulnerability on the VM, making this VM a ready-to-go sandbox for penetration testing and hacking.

The implementation of this work - named "auto-hackbox" - is available in the following repository:

`https://github.com/nader-alhalabi/auto-hackbox`

Contents

List of Figures

Listings

Chapter 1

Introduction

This introductory chapter provides a summary of the motivation, the desired aim, and the structure of this work.

1.1 Motivation

Penetration Testing (Pen-testing) is an important and high demanded skill in the cyber security world. It is also an essential task to do when developing critical applications. Getting into the world of penetration testing, on the other hand, is not a trivial quest. There is a lot of information to process, plenty of vulnerabilities to learn, and a big responsibility to be had.
Learning pen-testing is a big challenge, especially when it comes to applying what was learned theoretically to an actual machine. A learner can test his knowledge and skills on real targets or machines, but that can lead to harm on both sides.
For this reason, beginners tend to test their newly learned skills in artificial environments, and one of the best environments for this purpose is virtual machines.
However, setting up a virtual machine for pen-testing can be a daunting and time-consuming task, particularly, when a lot of vulnerabilities are involved.
This work aims to automate this difficult task by developing a software that installs multiple modules. These modules act as a package that performs a specific task to a virtual machine, which can include dependencies or configurations. A user can also pass a set of parameters to configure these modules suiting the intended use. The modules then install these specific dependencies and configurations to form a hackable or penetrable virtual machine.

1.2 Objective

The goal of this project aims to create a simpler way to set up a virtual machine for penetration testing, this also involves giving the user the ability to pass various configurations and parameters to the VM.

The implemented software must be capable to install several modules in an automated process, that ensures the compatibility of the dependencies between them. Modules should have an organized way to handle dependencies, whether they are provided or needed.
Metadata for the modules must also be designed to hold all the necessary information about a module (name, dependencies, configurations).
The automated process will have a validation measurement for the metadata to guarantee the success of the process. In addition to that, the ability to set pre-defined values as configurations for the dependencies.
The program must be able to communicate with VirtualBox reliably to install modules in the right manner.

1.3 Approach and Structure

This thesis can be divided into five main chapters. In the beginning, the challenges that led and inspired this work are introduced and illustrated. Chapter 2 gives an overview of the basics to understand the methods and techniques of the work, Then, in Chapter 3, the conception and design of the intended software are established.
Afterward, a detailed explanation of the implementation and the structure of the designed program is provided in Chapter 4. Lastly, chapter 5 gives a brief rundown on the tests and the evaluation of the development process, this gets concluded with a summary and potential future development.

Chapter 2

Fundamentals

In this chapter, the technical basics of this thesis are presented, initially, an introduction to virtual machines and penetration testing is given, followed by some explanation on python and YAML files.

2.1 Virtual Machines

A virtual machine (VM) is a virtual environment that works like a computer system with its own resources, like CPU, memory, and storage, created on an actual physical hardware system. With the help of a software called "hypervisor", the machine's resources get separated from the hardware so they can be provided in the right manner to be used by the VM.
The physical machines, ones equipped with a hypervisor, are called host machines (host), while the many VMs that utilize its resources are guest machines (guest). The hypervisor treats the host's resources as a pool of resources that can be simply distributed and relocated between existing guests as well as new virtual machines. VMs are also isolated from the rest of the system, and multiple of them can co-exist on a single physical piece of hardware. They can be dynamically relocated between host servers depending on demand.
One of the advantages of virtual machines is allowing numerous operating systems to run on a single computer at the same time, and each operating system runs as if it's running on the host hardware, thus the user experience within the VM is almost identical to that of a real-time operating system experience running on a physical machine[15].

This allows penetration testers to apply their knowledge on disposable sandboxes that are as real as host systems, with no worry of potentially damaging hardware or harming people/organizations.

2.1.1 Snapshots

A VM snapshot saves the state and data of a virtual machine at a specific time. The state includes the VM power state (powered-off, running, aborted), and the data includes all data stored on the VM (memory, disk)[6].
The snapshot acts as a copy of the virtual machine and can be used to create multiple instances of the same VM or to restore the VM to a former state. Snapshots can be very useful in development and testing environments, where testing several code changes is needed to have a safe rollback point[2].
In the scope of this project, snapshots were used to roll back to a former VM state, in which the modules were not yet installed. It is not necessary to reinstall the operating system on the VM, instead, the VM can be restored to a fresh OS install before starting to run the modules install process.

2.2 Penetration Testing

A penetration test, or a pen test, is an attempt to safely expose and secure an environment by exposing known security issues. These issues can be found in operating systems, applications, or improper configurations. They are also used to validate the effectiveness of various security policies and defensive mechanisms.
Penetration testing is mostly performed using automated or manual technologies to systematically exploit servers, endpoints, web apps, and other possible exposures.
After successfully exposing a particular system, testers may try to launch other attacks on other internal resources in order to gain deeper access to that system via privilege escalation.
Penetration testing is a process utilized by various companies and organizations to identify and assess the security risks associated with various systems and networks. The results of the tests are typically presented to the organizations involved in the security operations and management to be resolved[16].
This work aims to make the learning of penetration testing simpler by automating the process of setting up a testing environment.

2.3 Python

Python is a high-level programming language that has a variety of object-oriented features. Its flexible and high-level structure makes it very attractive for developing rapid application development. Its simple and easy-to-learn syntax helps minimize program maintenance.
The rapid edit-test-debug cycle of Python makes it very easy to debug programs. When

an error occurs, the interpreter prints a stacktrace, which tells the program which of the available exceptions has been encountered.
The source level debugger simplifies the debugging process by allowing the program to inspect and evaluate the code at a time[17].
Python was the language of choice for this software, for the fact that it has most of the packages that are necessary for this work, in addition to its automation capabilities and ease of implementation.
Python 3.8 is the language and version of choice for the implementation of this work.

2.4 Metadata

Metadata is a type of structured reference data that contains the attributes of an information source. It is often referred to as data that describes other data.
Meta is a prefix that simply means än underlying description or definition of data.M̈etadata helps users easily find, use, reuse certain instances of data. For example, author, date created and file size are basic document file metadata. Metadata can be created in various ways, such as manual or automated. The former can likely be more accurate, allowing the user to input any information that they feel is relevant to the file[5].
Manual creation of metadata is the relevant type for this project. Modules creators must write a metadata YAML file, that describes the most crucial information to run this module. This metadata file can hold the name, version, and different dependencies, which this module provides or needs. Although, most importantly are the configurations details.
With metadata describing the relevant information in the module, it makes reading this module to run it more accurate and error-free.

2.5 YAML

YAML is a data serialization language that is often used to create configuration files, It stands for yet another markup language and evolved into ain't markup language, which highlights that YAML is for data and not for documents. It is also easy to understand and is human-readable.
YAML is a superset of JSON, so JSON files are valid in YAML, but it uses Python-style indentation to indicate nesting, as there are no usual format symbols, such as braces, square brackets, YAML files use a .yml or .yaml extension.
The structure of a YAML file is a map or a list. Mappings allow you to group key-value pairs into distinct values. Order is not relevant, and each key must be unique. A map needs to be resolved before it can be closed. A new map can then be created by either creating an adjacent map or increasing the indentation level.
A list sequence is a type of object that contains values in an order. It can contain

multiple items, and starts with a dash (-) and a space, while indentation separates it from the parent. Naturally, YAML also contains scalars that can be used as values such as strings, integers, or booleans[19].

Example of YAML syntax:

```
---
name: max
enrolled: True
languages:
  - english
  - german
marks:
  - programming: failed
  - math: 1.0
```

Code snippet 2.1: YAML example

YAML is the dominant file type for writing configuration files and metadata, it has the benefit of easier human readability, which helps module creators to step into writing metadata rapidly and comfortably.

2.6 Validation

In basic automated systems, data is entered with minimal or even no human supervision. Therefore, it is essential to ensure that the data that enters the system is correct and meets the needed values and standards[3].
Data validation is an automated check, which ensures that the values of the input data are logical and acceptable. However, it does not ensure the correctness of the data itself. Therefore, validation is only a method of trying to decrease the number of errors in the input data.
There are various types of data validation, for example, range check, which verifies whether input data falls within a predefined range[4].
For the purposes of this project, data type check is the type of validation that is necessary for validating the metadata. A data type check confirms that the data entered has the correct data type, such as a numeric field can not accept a string input.

2.6.1 YAML Validation

Validating a YAML file involves confirming the required key-value pairs. The module's metadata can contain maps and lists, and these need to be correctly formed, to ensure proper parsing and templating of the YAML file.
For example, a "name" key is required to be present, and the value is required to be a string. When either of both cases is not met, the module would not be parsed or templated properly.

2.7 Templating

emplating, or web templating, is an approach to represent data in various forms. It often involves the creation of a document (template) and the presentation of data in a form that is easily understood by a human audience.
The template generally looks like the final form, only with placeholders instead of actual data, which are typically in a simple form.
Data that is presented using that template can be also separated into two parts, data required to be rendered, and data required for the template itself (navigation elements if it is a site). Combining template and data creates the final output, which is usually a web page of some kind[11].
The modules in this project have also placeholders in their scripts (templates), which represent certain user-defined configurations written in the metadata file. The values of the configurations then replace the placeholders in the different scripts.

Chapter 3

Conception and Design

In this chapter, the main parts of this program will be discussed, and the design will be explained.
The main idea of this program is to take a specific list of modules, and after some operations on them to ensure their validity, it should start to install the modules on VirtualBox[14].
Installing the modules and transferring files is done through SSH, which means realistically, that SSH module must always be present or rather be installed first for other modules to function.

3.1 Modules

A module is a user-defined package, that does one or more sets of tasks and it has 3 main components that allow it to be automated:

1. **Metadata**
2. **Main script**
3. **Resources**

The metadata is a configuration file written in YAML, it defines all of the features of the corresponding module, and especially the provided and/or the needed dependencies. This file can also hold important configurations which are essential for the functionality of the module. Metadata has some rules that will be covered later in the design chapter.

The main script is a Bash script, which acts as a start point to the module, it usually has the initial SSH connecting and resources copying commands.

Resources are everything else that the module needs to perform its intended task. For example, it can be website static files, SQL dumps, or other helper scripts.

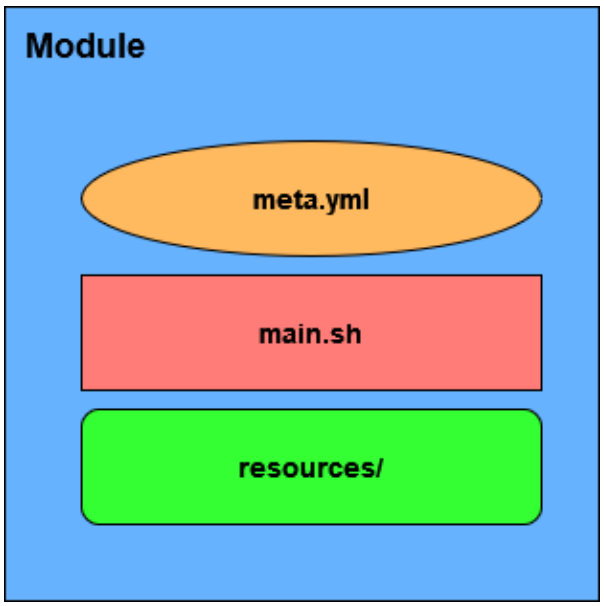

Figure 3.1: Module's content

3.2 Process

There are two considered ways to achieve the process to automate the install of modules: First by implementing a bash script that serves as an entry point to all modules, and then sending commands with a python script to it.

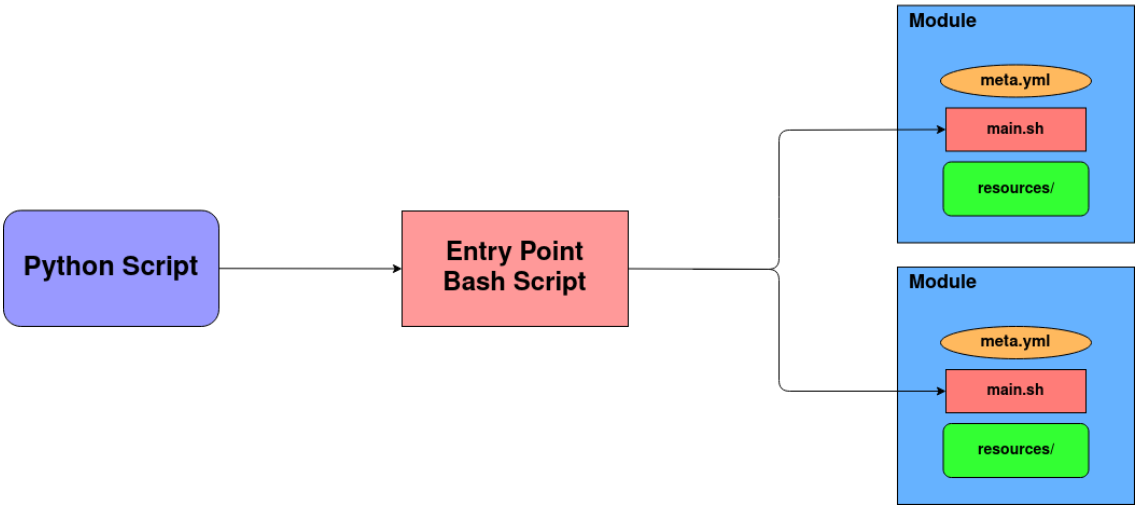

Figure 3.2: Process design 1

The second way is by controlling the modules from the main python script itself, this means that this python script is responsible for orchestrating the execution of bash scripts inside the modules.

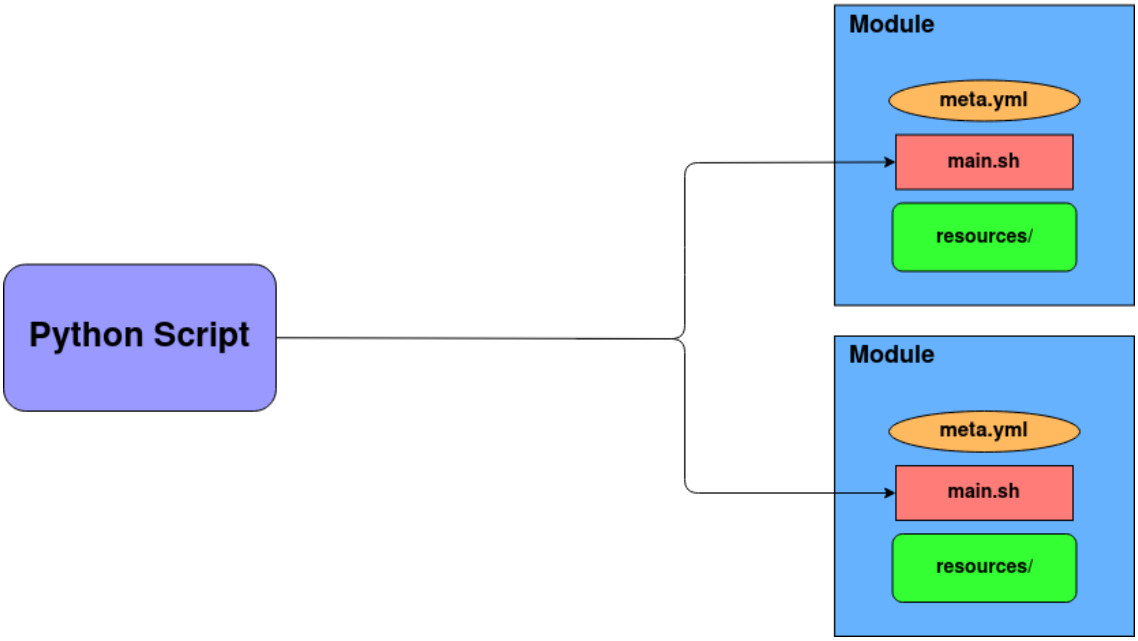

Figure 3.3: Process design 2

However, before the main python script executes the bash scripts in the modules, there need to be some necessary operations done.
First of all, metadata must be validated to ensure proper parsing of the YAML file, and afterward, the configurations available in the metadata must be parsed and then replaced with placeholders in the module's scripts.
The intended designed process will then look like the following figure3.4:

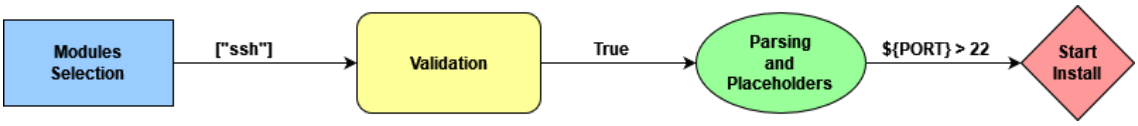

Figure 3.4: Full process

3.3 Validation

To ensure the success of the installation process, the metadata of the modules needs to be validated. A set of rules were declared to help creators of modules write a valid metadata file, this set of rules are written in a form of schema.
Furthermore, with the help of python package "cerberus"[13], this schema is validated against the metadata YAML file. In case of unsuccessful validation, the installing process will not begin.
These designed rules are explained in the following pseudo-code:

```
name:
provides:
    tech:  # list
      - entry:  # map
          name:  # string
          version:  # string
          config:  # list
    tech-config:
      - entry:
          name:
          version:
          config:
needs:
    tech:
      - entry:
          name:
          version:
          config:
```

Code snippet 3.1: YAML Schema

Each module has a "name" entity and provides one or more sets of dependencies and configurations. Entries under "tech" are the dependencies that this module provides. Lastly, the entries under "tech-config" are the configurations this module provides without providing the corresponding dependency for it.
Every "tech" entry has the name of dependency, the version of it, and all configurations it supplies. Same structure is for "tech-config".
For every configuration, the "name", "file", and "value" are listed.

1. **"name"** is the name of the placeholder that the parser will be replacing.
2. **"file"** is the name of the file, in which the placeholder should be replaced
3. **"value"** is a list of values that will be replacing the placeholder

```
config:  # list
    - name:  # string
      file:  # string
      value:  # list
```

Code snippet 3.2: YAML Schema

It could be noticed, that the “needs” entity does not have a “tech-config” entity of its own. That is for the reason that a module can not need a certain configuration without needing its dependency as well, thus deprecating the use of “tech-config” for it.

3.4 Template Engine

The metadata defines what dependencies the module has, and what configuration it provides and/or needs. This shows that parsing the metadata file must be done in an organized way.

When a module has a configuration, it consists of a placeholder in the main bash script or any other scripts in the resources folder. These placeholders are the locations where the values from the metadata should be replaced. This is where a template engine is used.

“A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.”[12]

This can be done by using an internal package of python called “string”, this package has a helpful function, namely “Template”[9]. It replaces placeholders with identifiers of the form ${PLACEHOLDER}.

Chapter 4

Implementation

The following section discusses how the requirements were implemented. First of all, the basic structure is explained, this is followed by a detailed presentation of the code. Every piece of component of the software will be implemented and explained on its own, and then the start script will be clarified. Last but not least, two examples of key modules are demonstrated.

4.1 Structure

Before the individual scripts in this project are explained, the general structure of the software should be overviewed.
As shown in figure 3.4, this work consists of multiple python components, with folders for module packages and test files.

- **validator.py** metadata validation for modules
- **schema.py** schema that validator.py uses to validate against
- **parser.py** loads metadata and replaces configuration with placeholders
- **runner.py** helper functions to run modules and restore snapshots on VirtualBox
- **main.py** main script that automates the previous steps into one workflow
- **modules/** a folder that houses all implemented modules
- **tests/** unit tests and workflow tests for the mentioned components

The implementation of the earlier mentioned components will be explained thoroughly in the next sections.

4.2 Validator

Before beginning any process in this automation workflow, a validation for the meta.yml file should be prioritized, for the reason that an invalid meta.yml leads to an invalid module, since there would be a conflict of dependencies and configurations.

For this task, a validation tool "called "cerberus" is used. "cerberus" provides simple and lightweight data validation functionality and is designed to be easily extensible, allowing for custom validation[13]. It takes a pre-defined schema and validates it against the passed module's metadata.

```
def valid_meta(module):
    schema = eval(open('./schema.py', 'r').read())
    v = Validator(schema)
    meta = load_meta(module)
    if v.validate(meta, schema):
        return True
    else:
        # uncheck below comment to debug False validation
        # print(v.errors)
        return False
```

Code snippet 4.1: Metadata validation

First and foremost, a schema python file is loaded up in "read" mode, after that, a "Validator" instance is initiated and the schema is passed on to it.
With a pre-defined function, the selected module's metadata file is also loaded up to a variable in "read" mode. Lastly, a condition is set to validate this metadata against the formerly loaded schema, returning "True" when all schema rules match the metadata, and "False" when they do not.

The schema file defines the rules of the metadata, with detailed information about the data type and requirements of every entity. A snippet from the long and very nested "schema.py" file can look like the following:

```
{
# name of the module, a simple string
    'name': { 'required': True, 'type': 'string'},

    # provides: a dictionary of provided modules
    'provides': { 'required': True, 'type': 'dict', 'schema':
    {
            # dictionary of technologies that the module provides,
            # dictionary type was chosen because every tech entity can
   have different options/configs,
            # can be nullabe when no technologies is provided
            'tech': { 'required': True, 'type': 'list', 'schema':
            {
```

```
                'type': 'dict', 'schema': {'entry': {'type': 'dict', '
    schema':
                {
                    'name': {'type': 'string'},
                    'version': {'type': 'string', 'nullable': True},
                    'config': {'type': 'list', 'nullable': True, '
    schema':
                    {
                            'type': 'dict', 'schema':
                            {
                                # value of type list to contain more
    complex types of values
                                'name': {'type': 'string'},
                                'file': {'type': 'string'},
                                'value': {'type': 'list'}
                            }
                    }
                }}
            }}}
            },
```

Code snippet 4.2: Validation schema

The validator can be executed by running the following command:

```
python validator.py [MODULE]
```

Code snippet 4.3: Validator command

4.3 Parser

After validating the metadata and ensuring its usability regarding the parser, the next step in the automated process is parsing and templating. Parsing is the extraction of configuration from the metadata file, particularly the provided, as well as, the needed configuration.
Templating, on the other hand, as was explained in section 3.4, is when the template engine replaces the placeholders in the module with the given configuration in the metadata.
However, before starting to replace placeholders, the module package acts as a template. This means the parser first replicates the module's folder and adds "-ready" after the name of the newly copied folder. For example, the module "ssh" becomes "ssh-ready".

```
def copy_module(module):
    source = './modules/{}'.format(module)
    target = './modules/{}-ready'.format(module)
    try:
        shutil.copytree(source, target)
    except FileExistsError:
        print("Module directory already exists")
```

Code snippet 4.4: Copying directories

This allows the reuse of modules since the original module's folder doesn't get tampered with.
Placeholders are variables in bash scripts, they look like "${VARIABLE}", where VARIABLE stands for the placeholder's name.
When the parser finds a configuration in the metadata, it contains the placeholder's name, value, and the file where it should be replaced. Afterward, the parser proceeds to the mentioned file and checks for the variable, and replaces it with the given values.
In the snippet below 4.5 , the parser navigates to file path, loads it in "read" mode, and starts replacing the placeholders with "safe_substitute()". Ultimately, the output is written in the same "ready" directory.

```
def replace(module, filename, replacements):
    script = open('./modules/{module}-ready/{filename}'.format(module=
   module, filename=filename), 'r')
    script_content = script.read()
    script.close()

    template = Template(script_content)
    sub = template.safe_substitute(replacements)
    outfile = open('./modules/{module}-ready/{filename}'.format(module=
   module, filename=filename), 'w')
    outfile.write(sub)
    outfile.close()
```

Code snippet 4.5: Templaing

The parser can be executed by running the following command:

```
python parser.py [MODULE]
```

Code snippet 4.6: Parser command

4.4 Interacting with VirtualBox

The convenient “pythonic” way to communicate with Virtualbox was supposed to be a python package called “virtualbox-python”[8]. Unfortunately, the package has not been updated for quite some time, and various problems were encountered during the use of “virtualbox-python” related to locking and unlocking virtual machine’s states and sessions.
For this reason, a more simple way for interacting with VirtualBox was opted for, namely by directly executing bash commands from python, using the VirtualBox’s own CLI “VBoxManage”[1]. The internal package ”subprocess”[10] provides the functionality of executing bash commands directly from python. Next, the process of restoring a snapshot using the formerly interduced tools will be explained:

```
1  def restore_snapshot(module_name, snapshot):
2      process = subprocess.Popen(
3          'vboxmanage snapshot {VMNAME} restore {STARTSNAPSHOT}'. \
4              format(
5              VMNAME=module_name,
6              STARTSNAPSHOT=snapshot
7          ),
8          shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE
9      )
10
11     output = process.stdout.read().decode('UTF-8')
12     error = process.stderr.read().decode('UTF-8')
13     print(output)
14     print("************")
15     print(error)
16     process.wait()
```

Code snippet 4.7: Restore snapshot

Initially, the function ”subprocess.Popen” takes the bash command as a parameter (line 3).
The requested VM name and the snapshot are passed on to the command to be executed. Also, this function captures the standard output and error which the command produced (line 8).
The output and error are then decoded and saved into variables to be printed after the command is done executing. The printed stars on line 14 are only to differentiate between the logs of the output and the error on the console.

However, one disadvantage to this method is the output and error logs are only printed out after the bash script finishes executing, which means modules with a big number of dependencies can take a decent amount of time without seeing the install log or progress in real-time on the console.

4.5 Start Script

Last but not least, the main script that automates all the previous components.
In addition to that, it has a simple dependency management to check that every module on the install queue has the needed dependencies to be able to install and/or function.
Nonetheless, the main script has two ways to execute depending on the user's use case

4.5.1 Multi-module mode

```
1 python main.py
```

Code snippet 4.8: Multi-module command

This mode is the standard multi-module automated and managed way to install a list of modules while undertaking dependency management.
It starts with listing all modules in the "modules/" folder on the console while also listing every provided and needed dependency for each listed module, this gives the user an overview and brief understanding of every module and can queue them in the proper order for the install process.
Nonetheless, the metadata validation check occurs first when the user adds a module to the install list. If the validation is valid, the program goes on to dependency checking, and if the validation is invalid, the process is stopped and a corresponding error message is printed.
Dependency management, on the other hand, kicks in when the user tries to install a module, which needs one or more dependencies that are not present in the general dependencies list. However, if that is not the case, the dependencies of the selected module are appended to the former mentioned dependencies list, to be checked again with the next module.
And as might be expected, when a module does not have any needed dependencies, it passes dependency management and gets appended directly to the install list.
After the user selects the desired modules, the install process can then be started with the keyword "run", the program then prompts the user to enter the VM's name and snapshot's name, so the modules can be installed on the intended VM and the right snapshot.
Eventually, when the whole list of modules has finished installing, the "ready" folders of the modules are then deleted.

4.5.2 One module mode

```
python main.py -m [MODULE] -v [VM] -s [SNAPSHOT]
```

Code snippet 4.9: One module command

This mode is intended for more advanced use cases, as the selected module does not go through dependency checking, for the reason that there is no dependency list to compare to.
One module mode lets the user install one module directly from the CLI, and it is the user's responsibility to be aware of the module's dependencies and what it needs and provides.
It is activated when python recognizes any arguments after calling "main.py", and this can be observed in upcoming snippet4.10:

```
if __name__ == '__main__':
    if len(sys.argv) > 1:
        one_module_mode()
    else:
        multi_module_mode()
```

Code snippet 4.10: One module function

One module mode accepts 3 arguments, that are all required and case sensitive:

- -m, –module : Name of the module
- -v, –vm : Name of virtual machine
- -s, –snapshot : Name of snapshot

It was noticed that entering an invalid snapshot name does not abort the install process, rather installs the module without restoring a snapshot.
However, an invalid module or VM name results in not starting the install process.

4.6 Module Examples

The last section in the implementation chapter will glimpse through some of the uncomplicated implemented modules, that will give a simple example of what modules could be potentially created.

4.6.1 ssh

"ssh" is an essential module to this software, as it initiates an ssh connection between the host system and the virtual machine for other modules to be able to execute scripts on the VM.

```yaml
---
name: ssh

provides:
  tech:
    - entry:
        name: ssh
        version:
        config:
            - name: PORT_FORWARDING
              file: main.sh
              value:
                - "2200"

needs:
  tech:
```

Code snippet 4.11: ssh metadata

As shown in the snippet above 4.11, this represents the meta.yml file of the module "ssh", it provides only one dependency with its only configuration.
This configuration is "PORT_FORWARDING", which forwards an ssh port to the host system, and the port value is "2200". In addition, the meta.yml file indicates that the ${PORT_FORWARDING} placeholder can be found in "main.sh" file and should only be replaced there.
Needless to say, the module does not need any additional dependencies, therefore it passes the dependency management process.

Next, the "main.sh" script of this module will be recapped and thoroughly explained:

```
1  !/usr/bin/env bash
2  export TMPDIR=modules/ssh/temp/
3
4  vboxmanage modifyvm ${VMNAME} --natpf1 "SSH,tcp,,${PORT_FORWARDING},,22
       "
5  vboxmanage startvm ${VMNAME} --type headless
6  echo "Waiting for VM to come up..."
7  sleep 8
8
9  rm -f ${TMPDIR}/rootkey*
10 ssh-keygen -b 2048 -t rsa -f ${TMPDIR}/rootkey -q -N ""
11 sshpass -p 1234 ssh-copy-id -i ${TMPDIR}/rootkey.pub -p ${
       PORT_FORWARDING} root@127.0.0.1
12 chmod 700 ${TMPDIR}/rootkey*
13 ssh -p ${PORT_FORWARDING} -i ${TMPDIR}/rootkey root@127.0.0.1 "ls"
14
15 vboxmanage controlvm ${VMNAME} acpipowerbutton
16 sleep 5
```

Code snippet 4.12: ssh main.sh

At the start, the temporary directory is defined to store the ssh keys that will be created. In line 4, the "vboxmanage" API modifies the NAT rules of the virtual machine with the argument "–natpf1". This makes the VM port forwards the value at the placeholder $PORT_FORWARDING, which will be replaced by the parser to include the desired value in the metadata.

The module then starts the virtual machine in headless mode and waits for the VM to fully boot up.
Starting at line 9, the script force removes any previously created root keys that are already created in the temporary directory. Afterward, an RSA key is generated using "ssh-keygen" command and stored again in the temporary directory.
Next, an ssh connection is made using the public key, and again, through the previously defined forwarded port. Following this, the access permissions for the keys are modified, so other modules can also initiate a connection as the same root user.
Last but not least, at line 15, a test ssh connection is made with the root key, and a "ls" command is executed, which lists all files in the current working directory.
Finally, the virtual machine is powered down to apply the changes and be ready for the next modules.

4.6.2 ssh-userpass

The purpose of this module is to automatically create users with their passwords on the Linux VM, this allows for quick generation of multiple users, for example, this can be handful for the training for privilege escalation.

```
---
name: ssh-userpass

provides:
  tech:
    - entry:
        name: ssh
        version:
        config:
          - name: PORT_FORWARDING
            file: main.sh
            value:
            - "2200"
          - name: username:password
            file: userpass_script.sh
            value:
            - nader:pass
            - max:muster

needs:
  tech:
```

Code snippet 4.13: ssh-userpass metadata

As observed in the "metadata.yml" file above, this module has the same provided features as "ssh" module, in addition to a "username:password" configuration.
This configuration provides a rapid way to add a user-defined list of users and passwords. It is specially handled by the parser, and it is triggered when a colon (:) is present in the configuration's name.
The special handler is to be seen in "parser.py" script and showcased in the following snippet:

```
def user_pass_parse(module, filename, conf):
    users = conf["value"]
    for user in users:
        username, password = user.split(":")
        script = open('./modules/{module}-ready/resources/{filename}'.format(module=module, filename=filename), 'a')
        script.write('sudo useradd -p $(openssl passwd -1 {password}) {username}\n'.format(username = username, password = password))

    script.write('rm /root/userpass_script.sh')
    script.close()
```

Code snippet 4.14: ssh-userpass parsing

As mentioned, when the special parser is triggered, the list of users from the metadata is passed on to the function "user_pass_parse()". The users are then put through a for loop to do several tasks.
Firstly, the usernames and passwords are split into different variables, since they are in the form of "username:password". Next, in line 5, a new helper script is created (userpass_script.sh).
Afterward, the usernames and passwords are passed to a bash script, that adds a user to a Linux system in one line6. Lastly, the helper script is deleted.

The produced helper script will look like the following after appending the list of usernames and passwords:

```
1 sudo useradd -p \$(openssl passwd -1 pass) nader
2 sudo useradd -p \$(openssl passwd -1 muster) max
3 rm /root/userpass_script.sh
```

Code snippet 4.15: userpass_script.sh

Chapter 5

Evaluation and Test

This chapter will provide the testing methodology and an evaluation of the development process, and the features of the resulted product will be overviewed and the limitations will be outlined.

5.1 Unit Tests

Testing is a crucial task of software development, as it confirms the intended designed functionality of the implemented tools and services. The testing of this software can be divided into two parts; unit tests and workflow tests.
Unit tests typically test the small units of a software. A unit can be a line of code, a method, or a class. Smaller tests give a much more granular view of how the code is performing[18].

In this project, only the "main.py" file was unit tested, as it has the main methods with return values that can be effectively tested. Next, snippets from the test files are outlined and explained.

```
1 def test_meta_validation():
2     assert main.meta_validation("ssh") == True
3     assert main.meta_validation("dre") == True
4     assert main.meta_validation("log-poison") == True
```

Code snippet 5.1: Validation unittest

In the snippet above is a simple test, that validates different modules for their metadata files. This validation uses the "meta_validation()" method, which reaches the "validator.py" script and tests the validation functionality on a unit level.

The next unit tests are the important part of this test script, and they should be run in the same order they are in.

```
def test_add_module_ssh():
    assert main.add_module("ssh") == True
    assert main.install_list == ["ssh"]
    assert main.dependencies_list == ["ssh"]
```

Code snippet 5.2: test_add_module_ssh

Initially, the "ssh" module is added, and asserted are the following:

1. If the module is gone through the addition process, which includes the metadata validation and dependency management.
2. If the module is present in the install list.
3. If the dependencies of the module are appended to the dependencies list.

As the "ssh" is a simple module with no needed dependencies, it passes properly through the full 3 tests and is added to the previously mentioned lists.

However, the same can not be said for the next test, as "dre" module does need missing dependencies which are not present in the dependencies list. Thus returning "False" from the first step.
This results in not appending the "dre" module to the dependencies nor the install list.

```
def test_add_module_dre():
    assert main.add_module("dre") == False
    assert main.install_list == ["ssh"]
    assert main.dependencies_list == ["ssh"]
```

Code snippet 5.3: test_add_module_dre

Next, the previously mentioned steps are tested on another module, namely "log-poison". "log-poison" is also a relatively simple module that needs only "ssh" as a dependency, thus passing through the first step "add_module" with "True" as return value. This means that the module is going to be appended successfully to the dependencies as well as the install list. This can be observed in the following snippet:

```
def test_add_module_log_poison():
    assert main.add_module("log-poison") == True
    assert main.install_list == ["ssh", "log-poison"]
    assert main.dependencies_list == ["ssh", "php"]
```

Code snippet 5.4: test_add_module_log-poison

Similarly, more different modules are added in the same way, and in the order, that lets them be appended properly to the corresponding lists.

5.2 Workflow Tests

Automation tools are considered as workflows since they have a set of components that work together to achieve the automated task. Therefore workflows should be tested in the right manner, "pytest-workflow" provides this functionality to python programs.
"pytest-workflow is a pytest plugin that aims to make pipeline/workflow testing easy by using yaml files for the test configuration."[7]
In this type of test, the validator.py and parser.py scripts and their functionality were tested.
The validation workflow tests are similar to the first unit tests; they simply test that the metadata for a certain module is valid.
However, this test operates on the end-user level, as it executes the same command that the user will execute when running this component.

```
- name: "Validate metadata for ssh"
  command: python validator.py ssh
  stdout:
      contains:
        - "True"
```

Code snippet 5.5: test_validator_ssh

In the snippet above, several YAML elements can be observed. "name" is for the test name that will be printed on the console. "Command" is the bash command that pytest will be executing. Lastly, "stdout" and "contains" mean that pytest will be expecting the values under "contains" to be present in the standard output of the test.
Regarding this test, it executes the command for validating "ssh" module and expects "True" as a return value in the output. The following tests have then conceptually the same behavior.

The parser tests, on the other hand, have quite more details in them. They also normally execute the command for parsing modules, however, they check for files rather than the output.

```
- name: "Parse metadata for ssh-userpass"
  command: python parser.py ssh-userpass
  files:
    - path: "modules/ssh-userpass/main.sh"
      contains:
        - "PORT_FORWARDING"
    - path: "modules/ssh-userpass-ready/main.sh"
      contains:
        - "2200"
      must_not_contain:
        - "PORT_FORWARDING"
```

Code snippet 5.6: test_parser_ssh-userpass

It can be noticed now that “files” element is present, and has a list of “path” and “contains” sub-elements. This case means that pytest navigates to the file in the “path” and looks for values in “contains”.
Relating to this test, pytest executes the command for parsing the module “ssh-userpass”. After that, it proceeds to the template folder for the module, and looks for the file ṁodules/ssh-userpass/main.sh: Then it checks if the placeholder “PORT_FORWARDING” is present.
Next, pytest advances to the “ready” folder of the module, and checks if the values of the placeholders are present. Additionally, it makes sure that the placeholders are no longer available using the element “most_not_contain”.

5.3 Evaluation

It has become apparent that this work focuses primarily on two aspects. These are the automation process and the module design.
The abstract goal of this work was to implement an automation tool for the purpose of modifying virtual machines. To evaluate the results of this implementation, the achieved objectives should be discussed and assessed.
Firstly, the process design was straightforward, running a python script should be able to execute different bash scripts in the modules. However, designing the initial prototype of this process was challenging, as the provided proof-of-concept examples were simple. Thus requiring a lot of pre-thinking to consider many potential module configurations and options.
Nonetheless, a reliable process was developed and future modules were considered along the way. The tasks which form the automation process had their own challenges as well. Mainly, template engines are usually implemented for web development to template HTML files with backend values.
Therefore, finding a template engine that is optimized for bash scripts was a difficult mission. Ultimately, a simple approach was opted for, considering the limitations it has. The parsing and validation components had good and well-founded libraries for each of them. As a result, the implementation of these parts was uncomplicated.
The passing of configurations requires little knowledge in editing YAML files, as it is not trivial as entering the parameters in the UI. Finally, the approach that was illustrated in the practical part of this thesis is suitable only for systems, which are based on a Linux distribution. In the end, the objectives were successfully achieved despite the faced challenges and obstacles.

Chapter 6

Conclusion

The last chapter in this work is going to briefly summarize the development process and some of the limitations of this implementation will be overviewed. Lastly, the potential future development of this project is shortly outlined.

6.1 Summary

This work dealt with the conception and the possible implementation of a software to configure virtual machines for penetration testing. The possibility to implement an automation tool for this task was researched. This tool must automate the process of installing user-defined modules to configure and modify virtual machines. It should also be possible to pass configurations and parameters at will to the modules to be installed along with the modules themselves. In the course of this work, a practical implementation was realized. For this implementation was the priority that it could be finished within the time frame of this work.
The objectives of this project were achieved, The modules were designed to act as a small automation system of their own. Every module has one or more tasks. Combining these modules produces a "penetrable" or "hackable" VM which introduces a certain vulnerability to pen test with.
The software consisted of multiple components, and every component was implemented as a standalone unit. A main script was then written to automate all of these components in a managed and organized approach.
Finally, workflow tests were implemented to confirm the functionality of each component.

6.2 Limitations

It is very difficult to create perfect software, and this software has its own imperfections regarding the implementation methods and design. Needless to say, the observed imperfections need to be addressed.

First, the software does not differentiate between the placeholder's identifiers. This means when the parser tries to replace a placeholder with values from the metadata, it can not recognize if the placeholder ${VARIABLE} is a configuration placeholder or a necessary variable for the functionality of the script itself. For this reason, the module creator should give attention to the naming of placeholders between the scripts and the metadata.
Another limitation is the naming rules of the scripts. The main script of the modules can only be named "main.sh", as it's hardcoded in the program. It could have been dynamically set according to its metadata, but it could have added a layer of unneeded complexity to the metadata structure as well as the install process. After all, most of the time the main script does only the initial functions such as the SSH connection, and it is good practice to have an organized module structure.
The last inconvenience is that after starting the install process of modules, the output and error logs are only printed on the console after a module install is finished, and not during the process. This is caused by the design of the "subprocess" library as it buffers all the output in variables, and prints it only after the execution of the bash command is finished. This can be annoying when installing large modules that provide a big number of dependencies since the output logs - and potentially the error logs - can not be observed immediately.

6.3 Future Development

The current state of this work only scratches the surface of the potential functionality of the project. The modules are particularly where the future development potential lies. As the structure of the modules allows for great expandability in the functionality of the scripts.
In addition to that, a web interface for the tool can be developed to provide the ability to select modules and pass configurations without editing the metadata. This will also make the tool more accessible to average users, thus making penetration testing more approachable.

Bibliography

[1] URL: https://www.virtualbox.org/manual/ch08.html#vboxmanage-intro.

[2] Nicolette Carklin. *Understanding the correct use of vm snapshots*. June 2021. URL: https://www.parallels.com/blogs/ras/vm-snapshot/.

[3] *Data Validation*. July 2021. URL: https://corporatefinanceinstitute.com/resources/knowledge/data-analysis/data-validation.

[4] *Data Validation | Techniques, Definition, How & What?* Feb. 2021. URL: https://teachcomputerscience.com/validation.

[5] Garry Kranz. "metadata". In: *WhatIs.com* (July 2021). URL: https://whatis.techtarget.com/definition/metadata.

[6] *Overview of virtual machine snapshots in vSphere*. URL: https://kb.vmware.com/s/article/1015180.

[7] *pytest-workflow*. Sept. 2021. URL: https://pypi.org/project/pytest-workflow.

[8] Sethmlarson. *Sethmlarson/Virtualbox-Python: Complete implementation of virtualbox's COM API with a pythonic interface*. URL: https://github.com/sethmlarson/virtualbox-python.

[9] *String - common string operations¶*. URL: https://docs.python.org/3/library/string.html#template-strings.

[10] *subprocess — Subprocess management — Python 3.9.7 documentation*. Sept. 2021. URL: https://docs.python.org/3/library/subprocess.html.

[11] *Templating - Python Wiki*. Sept. 2021. URL: https://wiki.python.org/moin/Templating.

[12] *Using template engines with Express*. URL: https://expressjs.com/en/guide/using-template-engines.html.

[13] *Welcome to cerberus¶*. URL: https://docs.python-cerberus.org/.

[14] *Welcome to VirtualBox.org!* URL: https://www.virtualbox.org/.

[15] *What is a virtual machine (VM)?* URL: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine.

[16] *What is Penetration Testing? | Core Security*. Sept. 2021. URL: https://www.coresecurity.com/penetration-testing.

[17] *What is python? Executive summary*. URL: https://www.python.org/doc/essays/blurb/.

[18] *What Is Unit Testing?* Sept. 2021. URL: https://smartbear.com/learn/automated-testing/what-is-unit-testing.

[19] *What is YAML?* URL: https://www.redhat.com/en/topics/automation/what-is-yaml.