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 Unterstütztung 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.

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.

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

Getting into the world of penetrating testing is a big challenge, especially when it comes to applying what was learned theoretically to an actual machine.
New learners should not apply their knowledge to real targets, and setting up a testing environment can be a daunting and time-consuming task, but a program that automates this process can lift this obstacle, and with the power of Python and Bash scripts, configuring a virtual machine for pen testing can be turned into a straightforward and effortless process.

1.2 Objective

This work aims to create a simpler way to set up a virtual machine for penetration testing, in addition, it is intended to enable the user to pass a particular set of configurations through metadata.
This could be achieved by implementing a python script that installs a user-defined list of modules to a specific virtual machine.

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[7].

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.

2.2 Penetration Testing

A penetration test, or a pen test, is a simulated cyberattack against a computer system for the purpose of checking for security vulnerabilities. Pen testing can expose various types of security weaknesses in an application system (e.g. APIs and servers), it can also identify unsanitized inputs that are vulnerable to code injection attacks.
The pen testing process can be broken down into five stages:

1. **Planning and reconnaissance:** Defining the goals and scope of a test, including the testing methods to be used and the systems to be addressed.
2. **Scanning:** Understanding the target application and how will it respond to several intrusion attempts. This can be done by inspecting an application's code to estimate the way it behaves while running.
3. **Gaining Access:** This stage consists of exploitation techniques that expose web application vulnerability, such as cross-site scripting and SQL injection. Attackers then use these techniques to escalate privileges and steal data to understand the scope of damage they can create.
4. **Maintaining access:** This aims to see if an exploit can be used to gain a persistent presence in an exploited system, long enough for a bad actor to gain in-depth access.
5. **Analysis:** Results of the pen test are then compiled into a report containing the vulnerabilities that were exploited, sensitive data that was accessed, and the amount of time the pen tester was able to remain in the system undetected[8].

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[9].
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 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[10].

Example of YAML syntax:

```yaml
---
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.

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[6]. 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:

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

The Metadata file is the core of the module, it consists of a YAML file that defines all of the features of the corresponding module, and especially the provided and/or the needed dependencies. The contents and rules of a metadata file will be covered later in the implementation 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.

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.
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.
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 figure:
TODO images

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, and with the help of python package "cerberus"[5], 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, under "tech" are the dependencies that this module provides, and 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 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

```yaml
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, and that's 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, and 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."[4]

This can be done by using an internal package of python called "string", this package has a helpful function, namely "Validation"[3].

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 xxx, 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 Metadata

Designing the metadata schema was an essential task considering the future potential complex modules that could be created, and plenty of thought went into implementing it.
Metadata is a meta.yml file inside the module, which has all crucial information for this module to be automated in this workflow.

4.3 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[5]. 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, and lastly, a condition is set to validate this metadata to the formerly loaded schema, returning "True" when all schema rules match the metadata, and "False" when they do not.

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

4.4 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 Template engine, 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, then the parser proceeds to the mentioned file and checks for the variable, and replaces it with the given values.

```
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

4.5 Interacting with VirtualBox

The convenient "pythonic" way to communicate with Virtualbox was supposed to be a python package called "virtualbox-python"[2], but unfortunately, the package has been not 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].

```
def replace(module, filename, replacements):
def restore_snapshot(module_name, snapshot):
    process = subprocess.Popen(
        'vboxmanage snapshot {VMNAME} restore {STARTSNAPSHOT}'. \
            format(
            VMNAME=module_name,
            STARTSNAPSHOT=snapshot
        ),
        shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE
    )

    output = process.stdout.read().decode('UTF-8')
    error = process.stderr.read().decode('UTF-8')
    print(output)
    print("************")
    print(error)
    process.wait()
```

Code snippet 4.7: Restore snapshot

In the scope of this work, "VBoxManage" was mainly used for starting VMs and restoring snapshots, As observed in the snippet xxx, the function "restore_snapshot" simply takes the module and snapshot's names and passes them to the bash command to be executed, it then prints out the standard output and error to 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.

4.6 Start Script

Last but not least, the main script that automates all the previous components, in addition to that, 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

4.6.1 Multi-module command

```
python main.py
```

Code snippet 4.8: Multi-module mode

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.

```
def add_module(selected_module):
    if meta_validation(selected_module):
        print("[*] Metadata validation successful")
    else:
        print("[!] Metadata validation unsuccessful")
        return False

    if check_dependencies(selected_module):
        install_list.append(selected_module)
        return True
    else:
        return False
```

Code snippet 4.9: Adding Module

After the user selects the needed modules, the install process can then be started with the keyword “run”, the program then prompts the user to type the VM’s name and snapshot’s name, so the modules can be installed on the intended VM and the right snapshot.

4.6.2 One module command

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

Code snippet 4.10: One module mode

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 arguments after calling “main.py”, and there are 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

This function is showcased in the following snippet:

```
def one_module_mode():
    parser = argparse.ArgumentParser(description='Choose the module, and its VM and snapshot (CASE SENSITIVE)')
    parser.add_argument("-m", "--module", help="Name of the module")
    parser.add_argument("-v", "--vm", help="Name of virtual machine")
    parser.add_argument("-s", "--snapshot", help="Name of snapshot")
    args = parser.parse_args()
    print(args.module,args.vm, args.snapshot)
    if args.module not in all_modules:
        print("Module", args.module, "is not available")
    else:
        global vm_name
        vm_name = args.vm
        runner.restore_snapshot(args.vm, args.snapshot)
        runner.run_modules([args.module])
        pars.delete_module(args.module)

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

Code snippet 4.11: One module function

4.7 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.

4.7.1 ssh

sh is an essential module to this software, as it initiates a 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 4.12: ssh metadata

As shown in the snippet above, this represents the meta.yml file of 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, and needless to say, the module does not need any additional dependencies, therefore it passes the dependency management.

```
!/usr/bin/env bash

export TMPDIR=modules/ssh/temp/


vboxmanage modifyvm ${VMNAME} --natpf1 "SSH,tcp,,${PORT_FORWARDING},,22
    "
vboxmanage startvm ${VMNAME} --type headless
echo "Waiting for VM to come up..."
sleep 8

rm -f ${TMPDIR}/mariokey*
ssh-keygen -b 2048 -t rsa -f ${TMPDIR}/mariokey -q -N ""
sshpass -p 1234 ssh-copy-id -i ${TMPDIR}/mariokey.pub -p ${
    PORT_FORWARDING} mario@127.0.0.1
chmod 700 ${TMPDIR}/mariokey*
ssh -p ${PORT_FORWARDING} -i ${TMPDIR}/mariokey mario@127.0.0.1 "ls"

rm -f ${TMPDIR}/rootkey*
ssh-keygen -b 2048 -t rsa -f ${TMPDIR}/rootkey -q -N ""
sshpass -p 1234 ssh-copy-id -i ${TMPDIR}/rootkey.pub -p ${
    PORT_FORWARDING} root@127.0.0.1
chmod 700 ${TMPDIR}/rootkey*
ssh -p ${PORT_FORWARDING} -i ${TMPDIR}/rootkey root@127.0.0.1 "ls"

vboxmanage controlvm ${VMNAME} acpipowerbutton
sleep 5
```

Code snippet 4.13: ssh main.sh

The main script of this module, presented by snippet xxx, manages to set the port forwarding rule with the help of the flag "–natpf1", this is done by executing direct commands using VirtualBox's CLI "vboxmanage". It then proceeds to start the VM and generate an ssh root key, afterward, the script copies the public key to the VM using the ssh and the previously defined port. Finally, shutting down the VM, and eventually, getting ready for the next module.

4.7.2 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.

```yaml
---
name: ssh-userpass

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


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

Code snippet 4.14: ssh metadata

For this module to work, the ssh port forwarding rule is needed, and as shown in the meta.yml file, under "needs" entity there is the port forwarding placeholder (PORT_FORWARDING), the files in which the placeholder is going to be replaced (main.sh), and the port value (2200), the same configuration that module "ssh" provides.
However, the provided configuration from the module "ssh-userpass" is a unique case, that is specially handled by the parser when it marks a colon (:), in this case "username:password", and this keyword triggers the function that handles multiple values and writes them in a script (userpass_script.sh) using a loop.

```
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 4.15: ssh-userpass parsing

Nonetheless, this module adds a simple functionality no top of "ssh" module, which is secure copying the helper script to the VM, and executing it there with a root privilege. The helper script creates new users on Linux with a one-liner.

```

#!/usr/bin/env bash

export TMPDIR=modules/ssh/temp/
export RESOURCEDIR=modules/ssh-userpass/resources/

scp -P ${PORT_FORWARDING} -i ${TMPDIR}/rootkey ${RESOURCEDIR}/userpass_script.sh root@127.0.0.1:/root/
ssh -p ${PORT_FORWARDING} -i ${TMPDIR}/rootkey root@127.0.0.1 "bash /root/userpass_script.sh"
```

Code snippet 4.16: ssh-userpass main.sh

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

Code snippet 4.17: userpass_script.sh

Bibliography

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

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

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

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

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

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

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

[8] *What is Penetration Testing: Step-by-step process and Methods: Imperva*. Dec. 2019. URL: https://www.imperva.com/learn/application-security/penetration-testing.

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

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