TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technology

Department of Software Science

ITC70LT

Artur Luik 163063

THE DESIGN AND IMPLEMENTATION OF AUTOMATED VULNERABILITY APPLICATION FRAMEWORK

Master's thesis

Tanel Tetlov

Master's Degree, Cyber Security

Researcher at NATO Cooperative Cyber Defence Center of Excellence

Tallinn 2018

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Artur Luik

21.04.2018

Abstract

The following paper is written in order to mitigate the problem of applying vulnerabilities to already existing machines for educational purposes. The author introduces the concept of automated vulnerability application framework which is intended for security experts, system administrators, vulnerability creation script developers, cyber security students and teachers to apply a known vulnerability to already existing machines. Vulnerable machines can be used to raise the awareness of the cyber security via demonstrating the vulnerabilities and their consequences. The goal of the paper is **to reduce the amount of time spent on preparing vulnerable machines.** The latter is achieved via designing and implementing a framework that allows to collaboratively gather a collection of reusable vulnerability creation scripts, find and detect collisions between vulnerabilities and apply them to a machine in a generic way. According to the author, this approach can save up to 40% of cyber security exercise preparation time. As a result of the paper, a prototype of the framework with abstract design guidelines is created.

This thesis is written in English and is 55 pages long, including 7 chapters, 23 figures and 3 tables.

Annotatsioon

Automatiseeritud turvanõrkuste rakendamise raamistiku disain ja implementatsioon

Järgnev dokument on kirjutatud, et leevendada õppeesmärgil turvanõrkuste tekitamise probleemi olemasolevatesse arvutitesse. Autor tutvustab automatiseeritud turvanõrkuste rakendamise raamistiku kontseptsiooni, mis võimaldab küberturbe ekspertidel, süsteemiadministraatoritel, turvanõrkuste loomise skriptide arendajatel, küberkaitse eriala õpilastel ja õpetajatel rakendada olemasolevaid turvanõrkuste loomise skripte, et luua turvanõrkuseid olemasolevasse arvutitesse. Dokumendis toodud ideede **eesmärk on vähendada turvanõrkuseid sisaldavate arvutite loomiseks kuluvat aega**. Eesmärki üritatakse saavutada läbi automatiseeritud turvanõrkuste rakendamise raamistiku, mis võimaldab turvanõrkust loovate skriptide kogukondlikku jagamist, kasutamist ja arendamist. Autori hinnangul on võimalik eelkirjeldatud meetodil säästa kuni 40% küberõppuse loomise ajast. Andmed põhinevad Eesti päritolu ettevõtte RangeForce kogemusel, mis väidavad, et ligi 20 protsenti ajast kulub planeerimisele, 40 protsenti loomisele ja 40 protsenti küberõppuse testimisele. Töö tulemusel valmib automatiseeritud turvanõrkuste rakendamise raamistiku disain ja prototüüp. Automatiseeritud turvanõrkuste rakendamise raamistik on loodud põhimõttel – ühe korra kirjuta, mitu korda kasuta. Raamistik võimaldab arendajatel mugaval viisil turvanõrkuste loomise skriptid kokku koguda ja jagada kogu maailmaga. Antud meetod tagab, et sarnaseid turvavigu vajavad situatsioonid ei peaks turvavigu loovaid skripte uuesti programmeerima, vaid kasutavad raamistiku poolt pakutud tööriista, et deklaratiivselt kirjeldada oma turvavigade vajadused arvutites. Lisaks turvavigade tekitamisele üritab raamistik vältida turvavigade vahelisi konflikte, kontrollida, kas turvaviga on võimalik üldse arvutisse tekitada ning pärast turvavea tekitamist kontrollida, kas arvuti on õige konfiguratsiooniga, et turvaviga ära kasutada.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 55 leheküljel, 7 peatükki, 23 joonist, 3 tabelit.

Table of abbreviations and terms

OS	Operating system – software that provides basic functionality for high level software.
DSL	Domain specific language, used to describe flows and actions in machine readable format. Actions and flows are related to one specific problem domain.
API	Application programming interface – functionality that is exposed to the developers to interact with a program.
YAML	Human readable data serialization language.
JSON	JavaScript Object Notation – Human readable data serialization language.
GCC	GNU Compiler Collection – includes, for example, C / C++ compilers.
XML	Extensible Markup Language – Human readable data serialization language.
UNIX	A family of multitasking, multiuser computer operating systems that derive from the original AT&T Unix [1].
Script	A program that is usually written in a scripting programming language to automate the execution of repetitive tasks.
Framework	A framework is a set of common practices and methods that allow you to achieve certain goals in a well-defined manner.
CLI	Command line interface – A program API that is usable via shell.
Service	A program that runs without user interaction in the background. Also known as a daemon.

Shell	A program that interprets commands from the user and returns their result back to the user (execution of a program).
Bug	A fundamental flaw in computer systems such as programmer's mistake or misconfiguration.
CVE	Common Vulnerabilities and Exposures is free to use and publicly available dictionary that provides definitions for publicly disclosed cybersecurity vulnerabilities and exposures [2].
IP address	A numerical label for a device attached to the network.
SSH	Secure Shell is a method for secure remote login from one computer to another [3].
Regressions	Situation in the software system where a change in functionality breaks already existing functionality not related to the change.

Table of contents

List of figures

List of tables

1. Introduction

The following paper is written in order to mitigate the problem of applying vulnerabilities to already existing machines for educational purposes. A vulnerability is a programmer's unintended mistake in a program's source code, misconfiguration or hardware design mistake that can lead to a malicious or unexpected behavior. The field of cyber security involves a lot of practical assignments, competitions, trainings and hands-on hacking demonstrations. All the latter require a significant amount of preparation work and expert knowledge in case one needs to demonstrate how you can exploit various programs and devices. One type of practical competition is called "capture the flag", where you need to exploit vulnerabilities to find a sequence of symbols called as "flag" in the device area which should be protected from unauthorized access. For instance, creating a machine for a capture the flag event needs infrastructure configuration (virtual machine deployments, network configuration), vulnerability creation, flag injections, testing vulnerabilities. All the steps can take a considerable amount of time. Furthermore, vulnerabilities, which are used for demonstrations are quickly getting out of date. The flag indicates that a person has found the vulnerability and exploited it correctly.

The purpose of this paper is to reduce the amount of time required for preparing demonstration machines for educational purposes via a common vulnerability application framework. Applying a vulnerability is a scalable problem - once you have a script for applying a vulnerability in a machine X with state requirements Y, you can use exactly the same script for other machines with the same state requirements Y. The script to create a vulnerability can be written by a community of people. One writes a new script - everybody benefits. The similar philosophy - "Knowledge is power, especially when its shared" is already implemented in the Metasploit Project [4]. Metasploit is meant for verifying that an exploit exists, this project applies already existing vulnerability to a machine (creates a security hole). The goal of this paper is **to reduce the amount of time spent on preparing vulnerable machines** by introducing a new framework architecture for applying vulnerabilities automatically in a generic manner (introduction of a new DSL - domain specific language). For the latter purpose, the author proposes the framework architecture, analyzes the conflict resolution/avoidance within the framework and demonstrates that this approach is possible and saves time by implementing the prototype of the framework that targets UNIX-like systems.

2. Related work

According to the author's best judgment, there seems to be no extensive scientific interest in vulnerability application automation at the moment. The following section will pinpoint some of the previous work done in this area, explain their working principles and compare them with the proposed framework.

2.1. SecGen

SecGen solves the issue of static vulnerable (the vulnerabilities stay the same) machines by creating vulnerable machines with randomized vulnerabilities and services, with constraints that ensure each scenario catered to specific skills or concepts [5]. It is meant to provide hands-on hacking experience for academic purposes by creating a fully operational virtual workspace using Vagrant and Virtualbox. Vagrant is a tool for building and managing virtual machines in an automated way [6]. VirtualBox is a cross-platform virtualization application that focuses on virtual machine creation [7]. SecGen solves the problem of static virtual machines - VulnHub [8], Metasploitable [9] websites already provide existing static vulnerable virtual machines. SecGen solves the problem by randomizing (or allowing a user to specify) the list of vulnerabilities that a target should have. SecGen uses the Puppet software [10], to automatically configure the created virtual machines according to predefined templates (Figure 1).

```
<?xml version="1.0"?>
<scenario
xmlns="http://www.github/cliffe/SecGen/scenario"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xsi:schemaLocation="http://www.github/cliffe/SecGen
/scenario">


  <system>
    <system_name>storage_server</system_name>
    <base platform="linux"/>


    <vulnerability
module_path="modules/vulnerabilities/unix/bash/shel
lshock"/>


    <network type="private_network" range="dhcp"/>
  </system>


</scenario>
```

Figure 1. SecGen DSL for describing vulnerable systems [11]

The example above (Figure 1) creates Linux storage server virtual machine with Bash Shellshock vulnerability [12]. SecGen is an invaluable tool in case you need to build the whole infrastructure. On another hand, if you want to modify an existing system, you are forced to use framework standards to implement the networking, virtual machine base contents, flag generators.

2.2. Metta

Uber Metta project uses Celery, Python, and Vagrant with Virtualbox to create adversarial simulation ecosystems. The main idea behind Metta is to simulate different known attack methods and phases in a virtualized environment. It allows system administrators to test their host / network-based detection mechanisms via simulated attacks. The attack scenarios are described using YAML documents (Figure 2).

```
enabled: true
meta:
  author: redcanary
  created: 2017-11-15
  decorations:
  - Purple Team
  description: load data to the Windows clipboard
  link: https://technet.microsoft.com/en-
us/library/cc754731(v=ws.11).aspx
  mitre_link:
https://attack.mitre.org/wiki/Technique/T1115
  mitre_attack_phase: Collection
  mitre_attack_technique: Clipboard Data
  purple_actions:
    1: cmd.exe /c whoami | clip
    2: cmd.exe /c powershell.exe echo Get-Process |
clip
os: Windows
name: load data to the Windows clipboard
uuid: ecdcb071-4763-4d97-9248-6790bc05586f
```

Figure 2: Uber Metta attack simulation – load data to Windows clipboard [13]

The example scenario tries to copy the user and process information to the clipboard in the targeted vagrant machine to simulate the attacker's behavior and test the system's attack detection mechanism. Metta is a considered useful in cases where you need to test your computer system/network defense plan.

2.3. AutoCTF

There is also a tool called AutoCTF, which addresses vulnerability creation on the program level - it enables users to create random memory corruption errors in C source code [14]. AutoCTF uses the LAVA bug injection system which adds bugs to the program. LAVA finds unused portions of the input and subverts them to introduce memory corruption errors into the program's source [14]. It addresses the same static virtual machine problem as SecGen. However, SecGen tries to create vulnerable machines while AutoCTF can create vulnerable programs.

2.4. Metasploit

Metasploit is the world's most used penetration testing framework [4]. It is a collaborative project between open source community and Rapid7 to implement automated scripts to exploit the known vulnerabilities in the systems. The main idea is to collect the vulnerability exploitation scripts into one framework and allow people to use them in a generic manner (Appendix 4). For example, in order to exploit Shellshock you can use the exploit/multi/http/apache_mod_cgi_bash_env_exec module and set the target host IP address and automatically get shell access on the machine (remote control the machine) [15]. In addition to vulnerability exploitation scripts, Metasploit has helper scripts that allow you to easily conduct lateral movement and gather data about the machines and network topology. The framework is presented as one toolkit for security teams to verify vulnerabilities.

2.5. Comparison with the current state of the art

We can compare existing ideas on purpose/abstraction level scale (Figure 3). Metta is completely in a class on its own. It is meant for adversarial simulation that can be done on a machine and network level. Metasploit is a widely used and known tool for exploiting vulnerabilities. For vulnerability creation, the proposed framework seems to compete with SecGen, however, SecGen has much wider area of responsibility - SecGen can create already existing targets and put them into the same network segment, the proposed framework will not create a target, yet it still works on a machine level, meaning, makes a machine vulnerable to attacks. The proposed framework will be a subset of SecGen‘s functionality focusing on defining vulnerabilities and conflict

resolution/avoidance. The latter will achieve separation of concerns – SecGen tries to solve all aspects of creating vulnerable machines (networking, flags insertions, virtual machines, vulnerability application). The proposed framework is designed to solve the problem of vulnerability application, which makes the tool more flexible and allows users to define their own environment around a vulnerability.

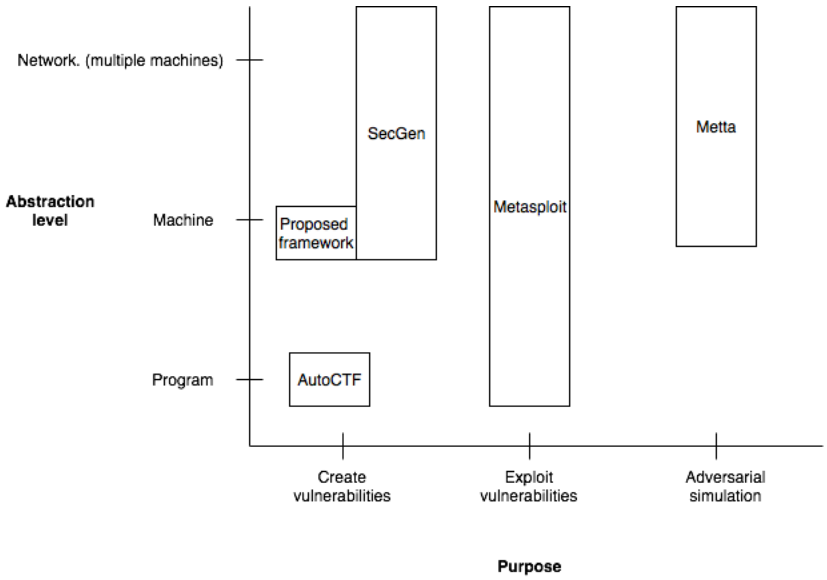


Figure 3: Comparison between existing solutions

3. Requirement analysis

The framework design has to support the users of the framework, therefore, before defining the requirements, the end users must be specified. The idea behind the framework is to collaborate with different parties, thus the developer (as a vulnerability provider) has an integral part in the ecosystem. The consumers of the vulnerabilities can vary - depending on the technical ability and needs, within the scope of this paper, the teacher, the system administrator (as a machine owner, maintainer, interested in breaking his/her own systems), the student and the security expert is taken into account in order to cover wide range of potential users.

3.1. User stories

The proposed framework must satisfy the following user stories to fulfill the potential needs of the users (Figure 4). The developer is an inevitable part of the system due to the nature of the framework, student, teacher, security expert and system administrator are most likely the framework's script collection users, therefore they should cover most of the use cases.

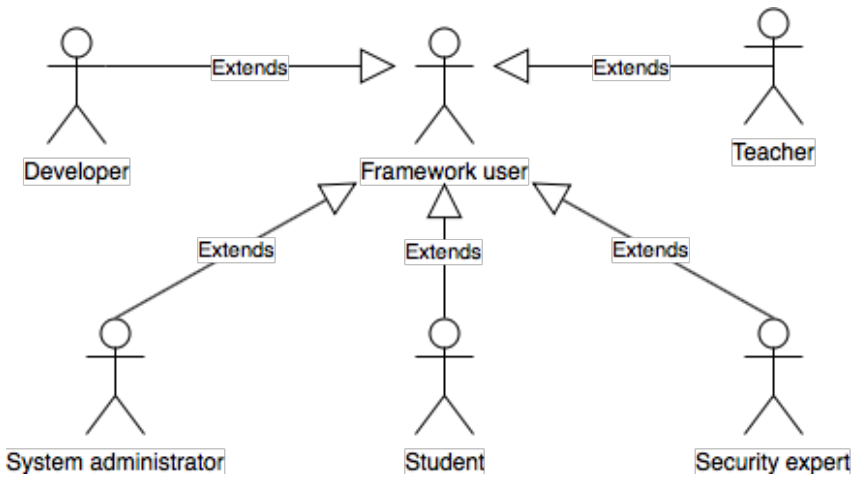


Figure 4: Framework actors

1. The developer would want to write new exploits without learning new scripting languages or frameworks. (Covered in OS abstraction layer)

2. The developer would want to test the vulnerabilities he/she wrote. (Covered in Framework lifecycle).

3. The developer would need a versioning system for vulnerabilities to keep track of the changes in the vulnerability creation scripts. (Covered in Vulnerability classification)

4. The system administrator would need to know whether the set of vulnerabilities work on target systems. (Covered in Framework lifecycle)

5. The system administrator would want to understand whether the applied changes work. (Covered in Framework lifecycle)

6. The system administrator would want to apply vulnerabilities automatically without user interaction. (Covered in The command-line interface)

7. The system administrator would need to apply vulnerabilities to multiple distributions and versions. (Covered in OS abstraction layer)

8. The student would want to have a hole in system X without knowing much about the technology. (Covered in The command-line interface)

9. The system administrator would want to modify existing machines, not to create new ones. (Covered in Framework lifecycle)

10. The security expert would want to search vulnerabilities by class, CVE identifier, description or software/system version. (Covered in Vulnerability classification)

11. The framework user would want to see which vulnerability scripts are available via CLI. (Covered in Framework vulnerability collection)

12. The framework user would want to keep program's size as minimal as possible (Covered in Framework vulnerability collection)

13. The system administrator would want to use the same tool for Windows and Linux servers. (The command-line interface)

3.2. OS abstraction layer

The vulnerability creation framework should be independent of OS, in other words, it must work in Windows, Linux, OSX. It does not mean that all the vulnerabilities will run on every platform, it means that the creation of a vulnerability has platform-independent implementation. For example, the standard procedure in vulnerability creation is file copying. Usually it already has abstraction on the programming language level (the file copying is implemented using underlying modules provided by the operating system kernel). However, it is reasonable to introduce an additional interface to protect from accidental platform changes in the future. Another example is checking whether a service is running. This is completely different for Windows and Linux and needs separate an implementation for each platform, even Linux distributions are not implementing all the features of service checking in a similar manner. The abstractions will also enable well-configured logging, which in turn leads to better debugging and error handling. The following example should demonstrate how OS-level abstraction should work.

Ubuntu 16.04 system: service apache2 start

Slackware systems (as of April, 2018): /etc/rc.d/rc.httpd start

Two different Linux distribution have completely different ways of starting the service. In order to satisfy the user story 8, the OS abstraction layer should handle the service lifecycle. Nowadays this can be done in multiple ways, for example with the following Puppet snippet (Figure 5) will do its best to guarantee that the service will be started

```
service { 'apache2':
  ensure => running,
  enable => true,
}
```

Figure 5: Puppet rule to ensure apache2 service is started

The same can be done with Chef and Ansible as well, however, the point of this section is to demonstrate the need for OS abstraction layer in order to satisfy the user story 8, the technical requirements will be analyzed afterward. In case the framework is directly depending on service apache2 start command the user story 8 will be violated once the system administrator wants to run the same scripts on Slackware systems. The OS

abstraction layer will also force developers to write high-level descriptions rather than implementing OS-specific functionality (cover user story 1).

3.3. Framework vulnerability collection

The most valuable asset in the framework is pre-written scripts that can be reused in similar environments to create a security hole. The collection should be visible for the public and searchable by meta information. The visibility and searchability should be available via CLI (as a part of user story 13). In other words, a framework user can open the program and search vulnerabilities using keywords that are compared against vulnerability classification parameters described in the next section - Vulnerability classification. The framework vulnerability collection should be stored in a third party data store in order to minimize the size of the program (the program should not contain all the vulnerability creation scripts).

3.4. Vulnerability classification

Classification of a vulnerability is an essential part of the framework in order to guarantee searchability and improve collision avoidance. According to user story 10, the vulnerability must have at least a CVE number (if applicable), a brief description and class. CVE provides identification for vulnerabilities, which can be used to for searching within the framework collection. In addition to the CVE classification, Common Vulnerability Scoring System v3.0 and The Common Weakness Enumeration should be used in order to give a user quick understanding of the severity of a vulnerability. The Common Vulnerability Scoring System (CVSS) is an open framework for communicating the characteristics and severity of software vulnerabilities [5]. CVSS also enables us to describe the characteristics of vulnerabilities, which in turn enables improved ability to search. The Common Weakness Enumeration (CWE) is a category classification for software weaknesses and vulnerabilities [16]. The CWE allows creating vulnerability trees, which act like behavior-based trees that are considered to be simple and flexible for human understanding [17]. Moreover, tags (related keywords, better searchability), vulnerability script version (semantic versioning [18] without patch version) and a reference to an external source for more information about the vulnerability would be useful for the framework user to quickly find the reference point. A similar approach (Figure 6) is used in SecGen as well with CVE, CVSS base score, CVSS vector

properties. From a technical point of view, in order to guarantee flexibility of the script implementation, the configuration parameters need to be defined (users can change the behavior of the vulnerability by changing the configuration).

```
<?xml version="1.0"?>
<vulnerability
xmlns="http://www.github/cliffe/SecGen/vulnerabilit
y"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.github/cliffe/SecGen
/vulnerability">
  <name>Bashbug / Shellshock</name>
  <author>Thomas Shaw</author>
  <module_license>MIT</module_license>
  <description>Installs GNU bash version 4.1 which
contains the bashbug / shellshock
vulnerability.</description>

  <type>bash</type>
  <privilege>none</privilege>
  <access>local</access>
  <platform>unix</platform>

  <difficulty>medium</difficulty>
  <cve>CVE-2014-6271</cve>
  <cvss_base_score>10</cvss_base_score>

<cvss_vector>AV:N/AC:L/Au:N/C:C/I:C/A:C</cvss_vecto
r>

<reference>https://web.nvd.nist.gov/view/vuln/detai
l?vulnId=CVE-2014-6271</reference>

<reference>http://www.symantec.com/connect/blogs/sh
ellshock-all-you-need-know-about-bash-bug-
vulnerability</reference>
  <software_name>bash</software_name>
  <software_license>GPLv3+</software_license>

</vulnerability>
```

Figure 6: SecGen Shellshock vulnerability metadata [19]

Another aspect of the vulnerability classification is the process description of how and what actually happens when a certain vulnerability is created. For instance, Shellshock is

a vulnerability that exists in GNU Bash (UNIX shell) version 4.3 or below [20]. The current Debian bash package [4.4-5], depends on

- dash (>= 0.5.5.1-2.2)
- libc6 (>= 2.15) [not arm64, mips, mipsel, ppc64el]
- libc6 (>= 2.17) [arm64, ppc64el]
- libc6 (>= 2.19) [mips, mipsel]
- libtinfo5 (>= 6)
- base-files (>= 2.1.12)
- debianutils (>= 2.15)

according to the Debian package repository information [21]. Now, a situation arises where a dependency conflict will be caused by the requirement of having both newest Debianutils and Shellshock present on the machine. Such a situation would have to be solved manually. For these purposes, an additional layer for describing vulnerability creation process is required.

3.5. Framework lifecycle

The goal of the user is to have one or more machines with the vulnerabilities. In order to achieve that, the user needs to specify the target machines and vulnerabilities that need to exist in the system. This combination of machines and vulnerabilities is named a scenario. The scenario concept is also used in SecGen to specify the vulnerability and base virtual machine [5]. The user must be able to run the scenario as well, which means that the selected vulnerabilities in the scenario are attempted to apply to the machines. In order to

know which vulnerabilities are available to use, one must be able to search for the vulnerabilities. The use cases are illustrated in Figure 7.

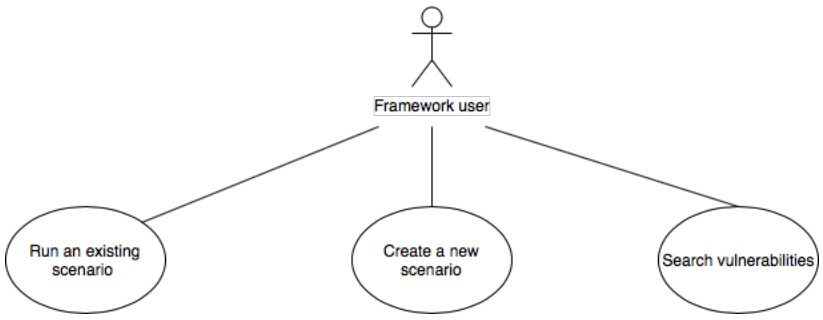


Figure 7: Framework's API use cases

Running an existing scenario (Figure 8) is the action where all the machines specified in the scenario are reconfigured to contain vulnerabilities listed in the scenario. The process should consist of three major steps: precondition check, configuring the vulnerability step and postcondition check in order to improve the reliability of the system (Design by contract principle) [22]. The precondition – postcondition logic is based on the assumption that every action has a cause and effect. The cause of the vulnerability is a broken program or configuration error, however, in order to create the cause, a certain set of preconditions need to be satisfied (a precondition for the existence of the configuration or program dependencies). In case the preconditions are met, the configuration step can begin. However, it doesn't mean that the developer, who gave an estimation for preconditions did not make a mistake. Therefore, postconditions to verify that the created changes actually appeared to need to be in place as well. The postconditions step is useful in spotting unexpected consequences of enormously interconnected systems, in other words, find regressions and unexpected changes.

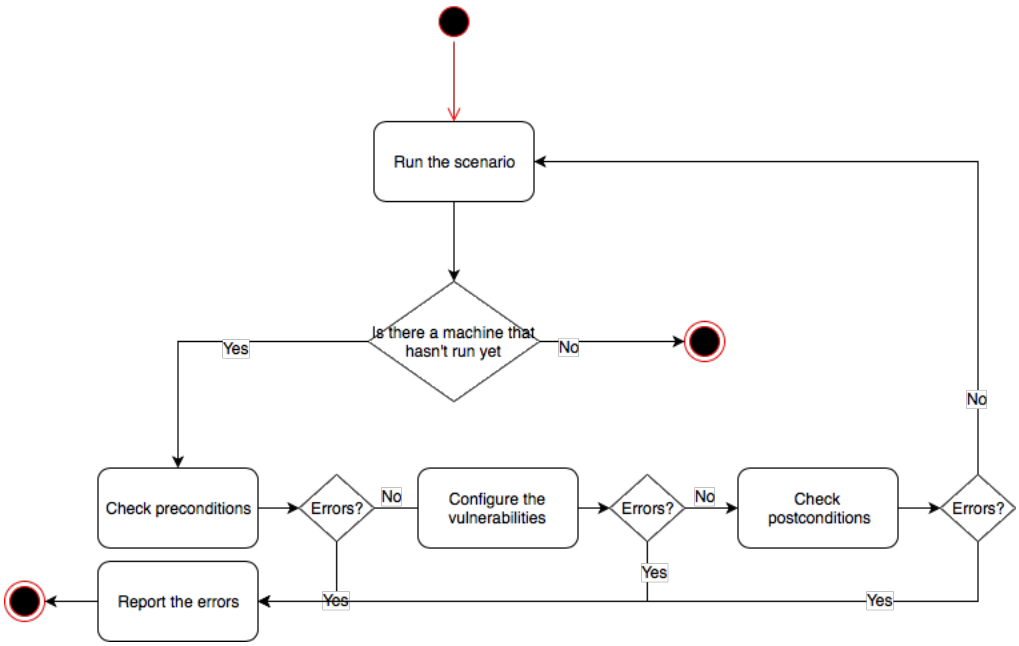


Figure 8: Running an existing scenario

3.5.1. Scenario preconditions phase

A vulnerability can be created either by misconfiguring the existing software or installing software that has a fundamental flaw. Both of the operations can cause conflicting changes. According to the recent research, about 31% of the Debian packages are conflicting because of conflicts on files and similar shared resources, 35% because of conflicts on shared data, configuration information, or the information flow between programs, 15% - uncommon, previously untested combinations of packages cause a conflict [23]. The first issue about conflicting file changes is usually mitigated on package manager levels with correct package metadata [24]. Introducing exactly the same metadata layer for the framework is not an option - direct competition with package managers is not needed, rather a fix for the package metadata is needed. However, with the OS abstraction layer, you can bypass the package managers completely and copy/edit files with different methods. Let us define OS abstraction layer as a set of functions F.

$$f_1(x_{11}, x_{12}, \dots x_{1m}), f_2(x_{21}, x_{22}, \dots x_{2l}) \dots f_n(x_{n1}, x_{n2}, \dots, x_{nk}) \in F, m, l, k, n \in \mathbb{N}$$

m, l, k is the number of arguments for the corresponding function f_i and n is a number of functions provided by the abstraction layer. The conflicting change between the vulnerabilities can happen if the vulnerability creation scripts modify the same resource, in other words, uses the same abstraction layer functions with conflicting arguments. By specifying the conflicting set of rules, the conflicts can be detected in the first step during

the precondition check. For example, through dry-running (running the scenario without actually modifying anything) we get set of operations $O \in F$ needed for the scenario. If the file copying function $f_1 = file\_copy(source, destination)$ is used twice with the same destination, it is a strong indicator that a conflicting change can occur. Considering the example, the following table (Table 1) can be built.

Conflicting operation pair	**Conflicting arguments**
$(file\_copy, file\_copy)$	$(destination, destination)$
$(file\_delete, file\_copy)$	$(file, destination)$

Table 1: Conflict table

Conflicting operation name indicates the pair of function that can possibly conflict in case conflicting arguments have the same value. In the case of $(file\_copy, file\_copy)$ and $(destination, destination)$, we assume that $file\_copy$ will conflict another $file\_copy$ function when the first $file\_copy\ destination$ equals to the second $file\_copy\ destination$. However, there can be pairs of functions that conflict in one case, but not in another. For instance, let's assume that a developer uses $file\_copy$ for creating a temporary directory and already another vulnerability creation script uses exactly the same directory to configure a vulnerability. In that case, the temporary folder does not influence the vulnerability itself, so, finding conflicting operations is rather a preventive measure and should be overridable by the user because the usage of those functions is completely up to a developer.

Therefore, during the precondition check, in order to detect possible conflicts between the vulnerabilities as early as possible, we need to

- Check that all the preconditions are met for each vulnerability (preconditions in Configure the vulnerabilities phase)
- Dry-run the scenario to get set of required operations for applying the vulnerabilities
- Check whether the set of operations have conflicting changes.

The approach above will improve the finding of conflicting vulnerabilities within the scenario. However, there is still chance that two vulnerabilities are using different

underlying resources due to the complexity of software dependencies (check Vulnerability classification, dependency conflict logic). The latter is highly dependent on the implementation of the software and developers are highly encouraged to use the package managers' features to encounter this problem.

3.5.2. Configure the vulnerabilities phase

This step configures each vulnerability in the scenario separately. For configuring a vulnerability X for the machine, the machine needs to have a state that is satisfactory for the vulnerability X. Therefore, before configuring the vulnerability, a certain set of preconditions for vulnerability need to be satisfied. The same check was already run in the preconditions step for the scenario, however, by the time the vulnerability configuring starts, the system may have changed its state (for example, creating another vulnerability that requires specific service version). Assuming that the service downgrade causes the vulnerability X, preconditions for service downgrade would be:

- Ensuring that service package is installed currently
- Ensuring that service is currently started

For postconditions, the developer can use the following assumptions:

- Service with new version package is installed
- Service is currently started

Postconditions are required in order to guarantee that the service or feature keeps working after modifications. This is an easy way to test the service/feature health/exploitability and find the regression (change that broke another part of the system). The model is abstract and fully customizable by the vulnerability configuration author and can be summarized as

Figure 9.

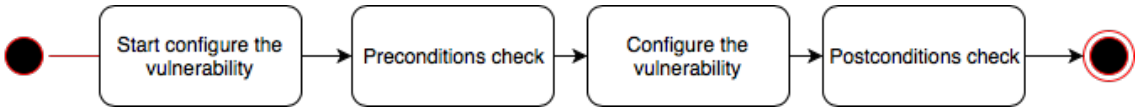


Figure 9: Configure a vulnerability

The following model (Figure 9) is required in order to satisfy the user story 2. The model will improve the framework's conflict avoidance (precondition check) and conflict detection (postcondition check) between the vulnerabilities.

3.5.3. Scenario postconditions phase

After the vulnerabilities have been configured, everything else is assumed to work in the usual way. In practice, this is really hard to achieve due to the nature of the software. during the preconditions phase the preconditions for vulnerabilities were checked, however, the fact whether or not the whole system keeps working was not checked. For the latter problem, the system is considered healthy if all the vulnerability postconditions pass.

3.5.4. Limitations and downsides of the proposed framework

The framework does not specify the OS abstraction layer. Therefore, the possible actions are limited to the OS abstraction layer functionality. Moreover, the vulnerability implementation rules are not strictly defined, which means that the quality of the script collection is defined by the developers; it is a well-known fact that developers make mistakes, the framework has no protection against the errors in scripts. The framework can be used after machines have been bootstrapped and have the network connection. In addition, the framework does not take into account the system reboots, a reboot needs to be handled by the developer during the configuration phase.

3.5.5. The command-line interface

The CLI is required to provide unified access to the framework and allow the users to perform a predefined set of actions to enable processes described in Framework lifecycle. The CLI has to be implemented in a platform-independent language (user story 14). The CLI must be able to create new scenario, run the existing scenario, search for existing vulnerabilities in non-interactive mode (the program exits immediately after the execution) or interactive mode. The non-interactive mode is meant for scripting, so

system administrators can use the tool without complications, interactive mode is meant for quickly trying out the possibilities of the tool and make the tool beginner friendly.

3.5.6. Domain Specific language

A domain specific language (DSL) is a generic term to refer to a standardized computer language markup to achieve a certain domain-specific task. In terms of vulnerability application framework, the DSL is used to describe the scenarios defined in the framework lifecycle. This paper defines three entities - scenario, vulnerability and target machine. The scenario is defined as a group of vulnerabilities and applies to a group of servers. The situation can be modeled with entity relation diagram (Figure 10).

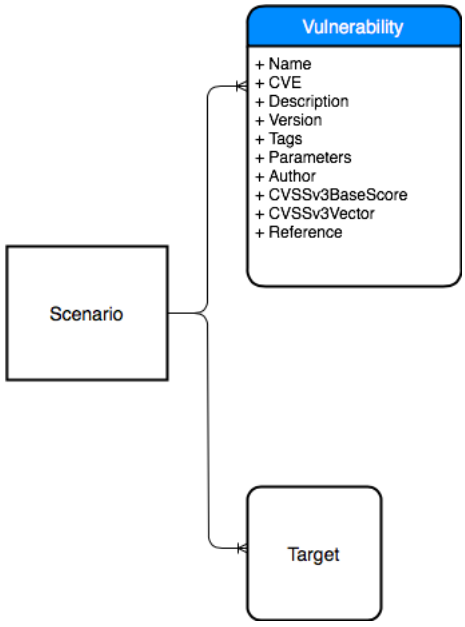


Figure 10: Scenario model

A vulnerability is defined by name, CVE number, description of the vulnerability, author, CVSS v3 base score, version, parameters, CWE classifications, CVSS v3 vector and reference to the vulnerability details. The vulnerability properties are explained in the Vulnerability classification section. A target is a machine that will become vulnerable after the framework has finished its actions. The implementation of entity relations can be done using one of the popular markup languages as XML, JSON or YAML. The markup language itself does not matter, the important aspect is that the information is presented in user readable format. For that purpose, YAML is considered to be more readable than JSON [25], which in turn is considered more readable than XML [26]. This is also complying with the Rule of Least Power, which suggests choosing the least

powerful language suitable for a given purpose [27]. The structure is simple (no deep hierarchy, properties needed) and doesn't need the flexibility of XML and JSON. Therefore, the example vulnerability and scenario would look like YAML documents (Figure 11, Figure 12).

```
name: Shellshock
description: Bashbug / shellshock vulnerability for
Debian
version: 1.0
tags:
    - shellshock
    - bashbug

parameters:
    -  param: Example parameter
author: Artur Luik
CVCCv3BaseScore: 4
CVCCv3Vector: AV:N/AC:L/Au:N/C:C/I:C/A:C
CVE: CVE-2014-6271
CWE:
   - OS Command Injections
Reference: https://nvd.nist.gov/vuln/detail/CVE-
2014-6271
```

Figure 11: DSL to describe a vulnerability

```
vulnerabilities:
- Shellshock:
   - version: 1
   - param: 4
targets:
- 192.168.0.10:
    - auth: ssh
- 192.168.1.10:
    - auth: ssh
```

Figure 12: DSL to describe a vulnerability application scenario

The vulnerability versioning (Shellshock=1.0) should be used in order to improve the overall consistency of the framework and avoid backward incompatible changes.

3.5.7. Ethical aspects

Building the described tool can have unacceptable consequences to society. For example, a malicious actor can take the collection of scripts and use it to create additional vulnerabilities, which makes the targets more vulnerable and causes increased hacking activity (low hanging fruit philosophy, more data breaches [28]). The malicious actor can also be the system administrator in a corporate environment, either deliberately or accidentally executing malicious scripts on a huge number of machines, which could potentially lead to huge data breaches. Moreover, the tool can be used by people without special training and accidental vulnerabilities can be created.

On the other hand, it can help system administrators to prepare the defense against vulnerabilities by creating example vulnerable machines and observing them; the framework can save thousands of hours to prepare and test the cyber exercises. It also enables teachers and lecturers to demonstrate and prepare their study material, which in turn leads to better education in cyber security field. Cyber security related studies and courses have been gaining popularity during recent years. According to the «The “Ethics” of Teaching Ethical Hacking» there is an ethical requirement for teachers to ensure that all the tools will be used for the right purpose [29]. Exposing this tool to the public can and will generate illegal use of it, however, the purpose of the tool is to enable people to learn and grow. Everything can be misused, it is up to people to decide how they should use it.

4. Technical implementation

The following section is meant to test the theoretical aspects presented in the analysis section in practice. As a result of the section, a working prototype of the framework will be implemented.

4.1. OS abstraction layer

Currently, there are four major competitors amongst the configuration management and orchestration tools – Puppet [10], Chef [30], SaltStack [31] and Ansible [32], Chef and Ansible are the most popular according to Google Trends (Appendix 5). All they are meant for automated configuration changes and software installations. The main idea behind the OS abstraction layer is to provide wrapper functions for actual OS modification calls in order to analyze the required changes for a vulnerability creation during the scenario preconditions phase. The orchestration tool should support dry-runs, which means that all the required operations are listed without actually changing anything.

Within the scope of this thesis, only UNIX-Like systems are targeted, which usually use SSH (Secure Shell) to establish a connection between a client and a server remotely. In order to achieve an extendable system, the underlying architecture should also support the Windows operating system. The abstraction layer should not require the huge overhead of management servers or agent installations to the targets. Management servers will require additional resources from the vulnerability framework service provider - maintenance and server costs, for running a light-weight vulnerability creation framework. This would require too much effort and complications. More complex systems tend to be less stable and maintainable, therefore, the software that provides OS abstraction layer should

- support SSH connectivity with clients
- have Windows support
- not require additional management servers
- simple to understand

4.2. Agent-less and agent-based software

The software that provides OS abstraction layer runs in the client as well as in the server (in case of Ansible, Puppet, Chef, SaltStack), which means that a client runs the software that communicates with a server and sends required changes to the server. The difference between agent-less and agent-based software is rather fundamental - the agent-less tools do not have specialized receiver on the server side (a target machine), in other words, agent-less tools will send all the required scripts from the client (machine used for initializing the configuration) to the server, while agent-based tools have already a receiver installed in the server. The agent-less - require nothing (no additional specialized software in a server) - philosophy is not entirely true. For example, Ansible needs working SSH daemon and up-to-date Python interpreter in the target machine (those requirements are already satisfied with most of the Linux distributions). In the case of the proposed framework, the agent-less software is preferred because the target machines may not have installed agents, therefore, in order to reduce the complexity of the tool, an agent installation is out of scope. Both Puppet and Chef, require additional agents (programs) installed in the server, meanwhile, Ansible and SaltStack (agentless mode) use no agents [32] [10] [30] [31]. Ansible is more popular according to Google Trends and the documentation of Ansible is more readable in the author's opinion, then Ansible is the preferred choice for OS abstraction layer.

4.3. Ansible

Ansible is a radically simple IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and many other IT needs [32]. Ansible connects to the machines using SSH and pushes and executes "Ansible modules", which are small programs that configure one or more resources in the target machine. Ansible uses declarative configuration files to represent the state of the machine. For example, the rule presented in Figure 13 will add config=1 to /etc/example.

```
- name: Copy the example configuration
  copy: content="config=1" dest=/etc/example
```

Figure 13: Ansible example

In general, Ansible will allow defining the server groups and roles. A server group is a list of servers that need to be configured and roles are steps that need to be taken in order to create the desired state of a machine. The combination of the group of servers and the roles creates an Ansible playbook. Ansible supports check mode ("dry run") that reports the changes that will be done without actually modifying target machines.

4.4. Developing vulnerability creation scripts

The declarative configuration files of Ansible described in the previous section can be used to describe each step in the framework lifecycle. For demonstration purposes, let's describe one vulnerability creation script implementation in Ansible.

4.4.1. Precondition phase

Shellshock vulnerability exists in GNU Bash version 4.3 or below [20], therefore the preconditions for the vulnerability creation script is intuitively the existence of the Bash program in the system, this can be implemented as described in Figure 14.

```
- name: Check bash existence
  command: bash --version
  register: bash_version
  tag: precondition
  failed_when: bash_version.rc != 0
```

Figure 14: Vulnerability preconditions phase in Ansible

4.4.2. Configuration phase

According to the National Vulnerability Database, the last bash version that was affected was 4.3 [20], therefore the Bash version 4.3 installation is required, this can be described with Ansible script (Figure 15).

```
- name: Create a temporary folder for bash files
  file: path=/tmp/bash state=directory

- name: Download Bash 4.3
  unarchive:
    src: https://ftp.gnu.org/gnu/bash/bash-
4.3.tar.gz
    dest: /tmp/bash
    creates: /tmp/bash/bash-4.3
    remote_src: yes

- name: Install bash 4.3
  shell:
    cmd: /bin/bash ./configure && /usr/bin/make &&
/usr/bin/make install
    chdir: /tmp/bash/bash-4.3
```

Figure 15: Vulnerability configuration phase in Ansible

4.4.3. Postconditions phase

After the configuration phase has finished, postconditions should verify that Bash 4.3 is actually installed. This can be achieved with

```
- name: Verify that Bash 4.3 is installed
  command: bash --version
  register: bash_version
  tags: postcondition
  failed_when: bash_version.stdout.find('version
4.3.0') == -1
```

Figure 16: Vulnerability postcondition phase in Ansible

4.4.4. Conclusion

It is possible to implement all three stages of vulnerability creation with Ansible; however, the implementation is completely up to the developer and can have multiple problems. First of all, even though the OS abstraction layer is used, the compatibility with other Linux distribution or versions is questionable, a developer cannot possibly foresee all the possible use cases of the script, moreover, the quality of the script can vary. For instance, the example above can be considered as poor quality for many reasons:

- Precondition step does not check whether the GCC compiler and build essentials have been installed
- An external data source is used, which leads to unexpected content modifications or deletion. Sometimes software hotfixes are getting backported, which means that the same version can contain a bugfix that fixes the expected vulnerability.
- Precondition and postcondition should not modify the existing state of the system, however, the developers can run arbitrary code during the precondition or postcondition step.

The code quality problems can be mitigated via implementing the vulnerability creation scoring system (to let users know about the quality), code review standards (the script will not end up in the collection unless somebody has approved it) or implementing automatic code linter (a program that checks and reports well-known mistakes). All the mitigation techniques are out of the scope of this thesis, yet, really important in the further development of the framework.

4.5. Vulnerability creation scripts within the framework

In the previous section, an example vulnerability creation script was created. In order to use it within the framework, a certain set of rules needs to be introduced. According to the framework lifecycle phases, the first phase needs to execute all the preconditions for the vulnerabilities defined in the scenario. The second step needs to execute for each vulnerability precondition, configure the vulnerability step and postconditions. After the scenario configuring phase, the postconditions for each vulnerability need to be executed once again. To achieve the latter, the structure to find the exact vulnerability and phases need to be introduced.

4.5.1. Tool structure

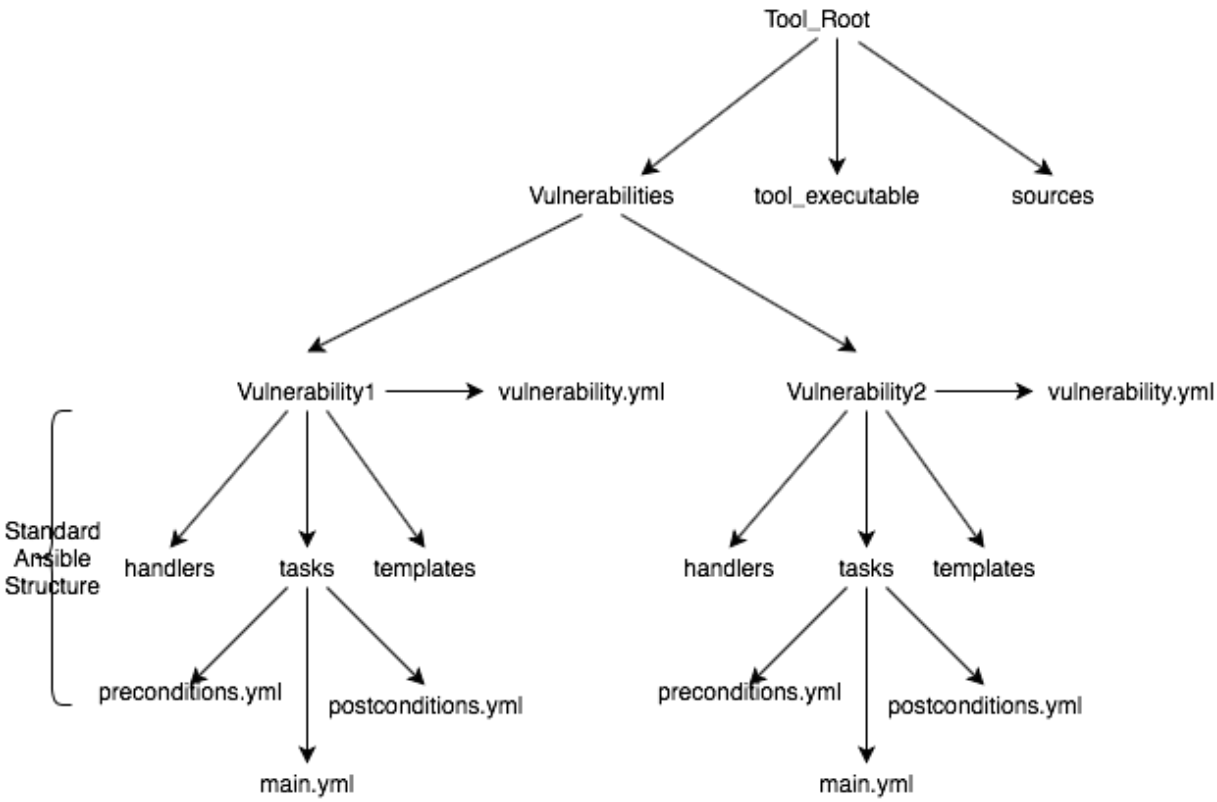


Figure 17: Tool structure

The following structure is being inspired by Ansible best practices, where a role (as a vulnerability name) is divided into handlers, tasks, templates, files, vars, defaults, meta, library, module_utils, lookup_plugins directories [33]. The idea is simple - in case the structure is fixed by Ansible standards, the developers can create new vulnerability creation scripts without known framework requirements in the first place, although, once the Ansible scripts are ready, the only requirements for the tool is that preconditions file

contains “precondition” tag for each action and postconditions file contains postcondition tag for each action. The structure in Figure 17 presents the directory structure of the tool in order to comply with Ansible best practices. The technical requirements are implemented as corresponding Ansible actions presented in Table 2.

Technical requirement	**Technical implementation in Ansible**
Check that all the preconditions are satisfied for each vulnerability	Run all Ansible roles that correspond to vulnerability script with the “precondition” tag, which executes only actions that are required checking whether or not the vulnerability is actually possible.
Dry-run the scenario to get a set of required operations for applying the vulnerabilities	Run all Ansible roles that correspond to vulnerability scripts with the check=True option and collect the actions that are about to be executed.
Check whether the set of operations have conflicting changes.	Use collected actions and compare them using custom logic in Python
Preconditions/Configure/Postconditions per each vulnerability	Supported out of the box by the developer (developers need to include precondition and postcondition to their Ansible role)
Check that all the postconditions are satisfied for each vulnerability	Run all Ansible roles that correspond to vulnerability script with the “postcondition”, tag which executes only actions that are required checking whether the vulnerability is actually created

Table 2: Technical requirements and corresponding Ansible implementation

In order to actually implement the behavior described in Table 2. Its role is to dynamically create the required Ansible runs and follow the Framework lifecycle principles.

4.6. Additional layer on the top of Ansible

The responsibility of the additional layer on top of Ansible is to actually interpret the designed DSL in Domain Specific language section, implement the flows described in the framework lifecycle section and provide the CLI as described in the command-line interface section. The programming language itself doesn’t really matter, however, recent

years have shown that Python is becoming more and more popular in the cyber security field. A lot of tools have been written in Python because of the vast variety of Python libraries, including Ansible library which provide application programmable interface, which can be utilized for implementing framework lifecycle.

4.6.1. Scenario preconditions phase

The Ansible Python API [34] with the help of Python programming can create the following logic flow (Figure 18).

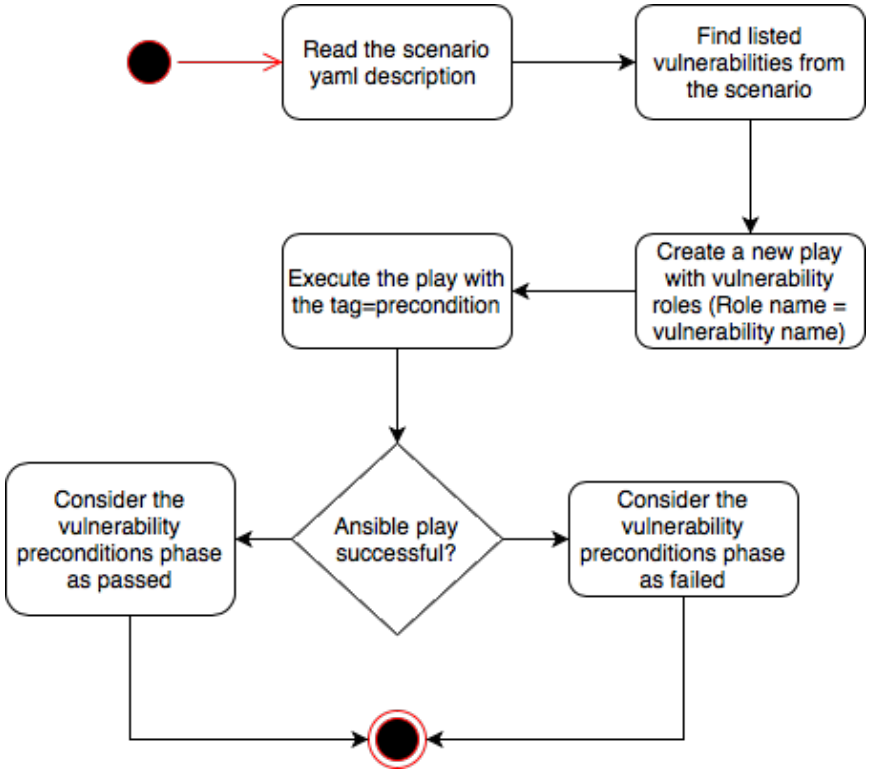


Figure 18: Precondition logic in Python

The flow described in Figure 18 will satisfy the criteria of the first step of a scenario precondition phase which checks if all the preconditions of the vulnerabilities are satisfied. The next step is to collect all the actions required for the vulnerability configuring phase. For that purpose, the Python Ansible library provides a callback 'v2_playbook_on_task_start', which can be used for getting actions and action arguments [35]. On the top of Python Ansible library, the following logic can be built.

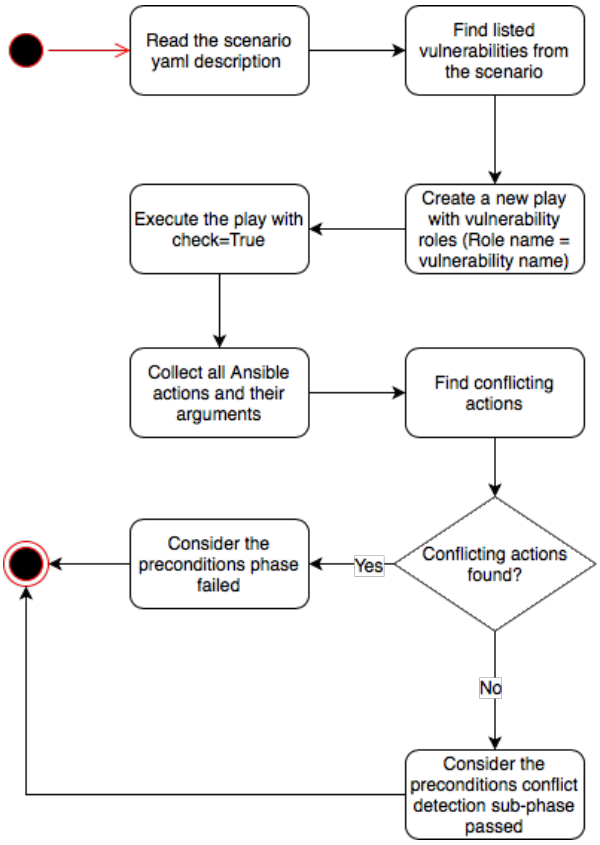


Figure 19: Preconditions - conflict detection

4.6.2. Scenario preconditions phase - conflict avoidance

As described in the Framework lifecycle section, OS abstraction layer functions (in this context Ansible actions) can be used for conflict avoidance. Ansible actions are defined by name and arguments.

```
- name: Create a temporary folder for bash files
  file: path=/tmp/bash state=directory
```

Figure 20: Temporary Ansible folder - possible conflict

The snippet above (Figure 20) defines an action ‘file’ with parameters path=’/tmp/bash’ and state=’directory’. In order to check whether this action conflicts with anything else, we need to know all the other actions that are needed to create the scenario and compare whether a pair of actions exists in the conflict table (Table 1: Conflict table).

Ansible has currently (17th of April, 2018) 1688 different modules that can be used [36], currently it is outside the scope of the paper to analyze each of them separately, the conflict table will improve once the tool gets more mature.

4.6.3. Scenario configuration phase

The proposed DSL has a really similar structure with Ansible playbooks, therefore to run the scenario configuration phase, the Ansible playbook has to be dynamically created and executed. The following logic presented below (Figure 21) can be used.

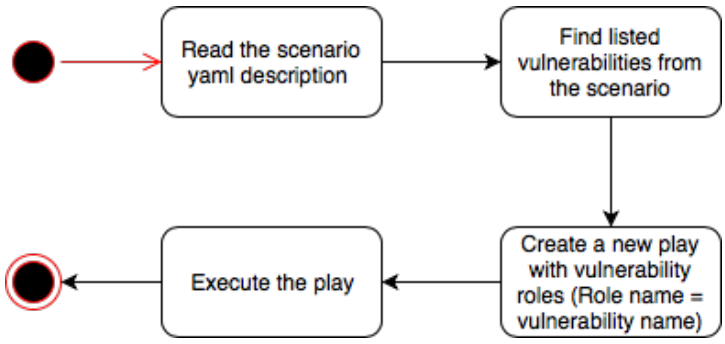


Figure 21: Scenario configuration phase in Python

4.6.4. Scenario postconditions phase

For checking postconditions, similar actions to preconditions need to be taken, except only with the "postcondition" tag. In this case, conflict avoidance can be skipped. Figure 22 illustrates the actions needed for checking scenario postconditions.

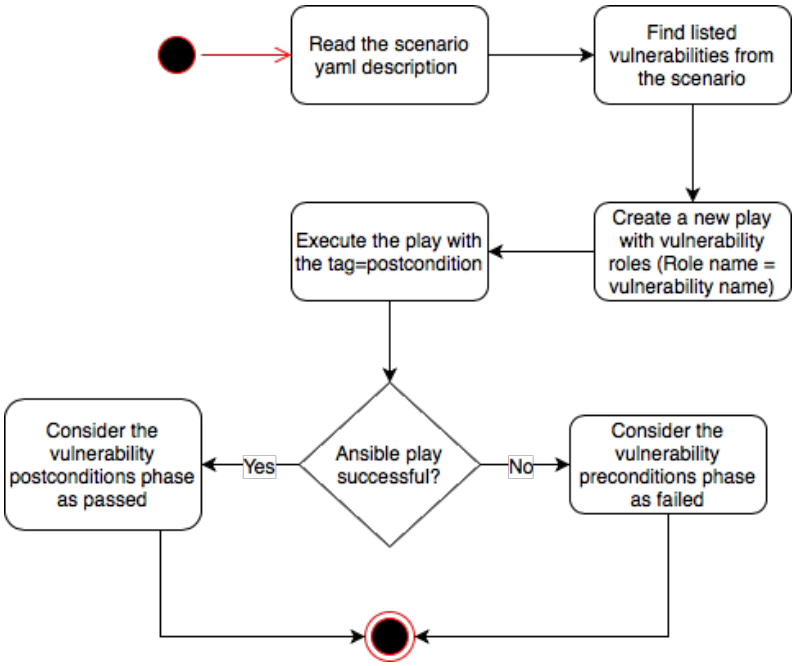


Figure 22: Scenario postcondition phase in Python

4.7. Command line interface

For the prototyping purposes, only the scenario of running and searching was implemented. A Python tool (Appendix 3) was created to allow running the scenario description (Figure 10). The prototype also allows searching for existing vulnerabilities. The tool uses partial string matching for searching vulnerabilities. The partial searching uses name, description, tags, CWE, CVE fields described in the framework DSL section.

4.8. Performance

For performance measurement of the implementation of the vulnerability application framework (Appendix 4), VirtualBox 5.2.8 with Ubuntu 17.10 was set up with sshd daemon and Python 2.7.14. The VirtualBox machine was used as a target that needs to have vulnerabilities. The MacOS 10.13.4 (17E199) was used with Python 3.6.5 and Ansible 2.5 for determining the performance the following measurements were done:

- Apply Shellshock to Ubuntu 17.10
- Apply Proftpd-1.3.3c-backdoor to Ubuntu 17.10
- Apply Shellshock and Proftpd-1.3.3c-backdoor to Ubuntu 17.10

Before each measurement, the virtual machine was restored to the initial snapshot. The goal is to figure out how fast a vulnerability can be applied. For time measurement the OSX time command line function was used.

Vulnerabilities	Result
Proftpd-1.3.3c backdoor	python3 src/run.py --scenario examples/example.yml 3.34s user 1.20s system 69% cpu **6.489** total
Shellshock	python3 src/run.py --scenario examples/example.yml 3.51s user 1.11s system 18% cpu **25.171** total
Proftpd-1.3.3c backdoor and Shellshock	python3 src/run.py --scenario examples/example.yml 5.23s user 1.89s system 23% cpu **30.262** total

Table 3: Applying vulnerabilities in practice - performance

The results depend on the network speed (50Mbit/s upload and 50Mbit/s download speed) because the example scripts are downloading content from the Internet. The speed can be improved by using precompiled binaries and removing the network dependency. Generally, the scenario running time is highly dependent on nature of the vulnerability configuration implementation, however, it will still be measurable within seconds or minutes (otherwise the implementation is wrong or inefficient). Manually applying the Shellshock vulnerability to a machine will take about 5 minutes (the author estimated the speed of applying the vulnerability manually via command line interface before creating a script, almost 10 times slower than with the script) if one knows exactly what needs to be done and the machine has all the preconditions satisfied, yet, debugging and figuring out all the prerequisites may take hours of searching and testing.

5. Estimating the benefits of the framework

According to RangeForce experience, one hour of cyber security exercise content is created within 200 working hours and planning, developing and testing have relative time ratio 20:40:40 (Appendix 2). The similar problem, yet in a larger scale, exists in Locked Shields exercise, where the content is actually created for 2 days [37]. The expanded graph of planning, developing and testing phases is presented in Table 1.

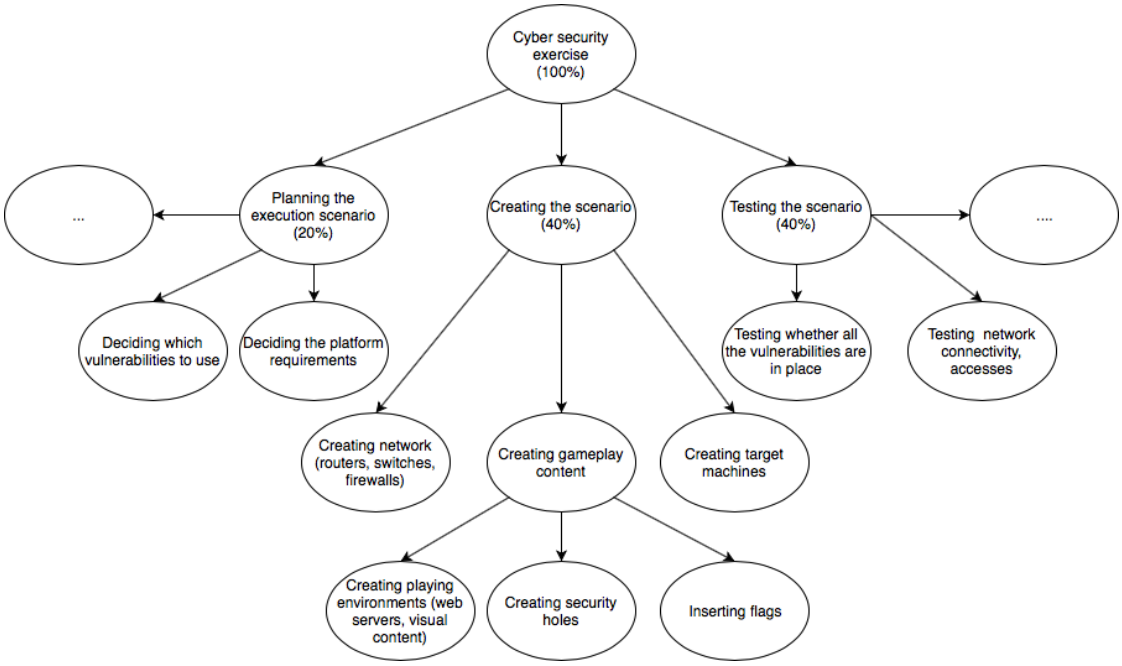


Figure 23: Cyber security exercise phases extended

The purpose of the framework is to reduce the time for preparing the vulnerable machines. By the design of the framework, it is meant to save time in creating the scenario (development) phase, however, there might be additional benefits within the planning phase as well. For example, the framework search functionality allows finding vulnerabilities for specific purposes, which in turn can help to generate ideas for cyber security exercises. Due to planning phase execution differences (depending on a purpose, methods used, scale, targets), it is hard to actually measure the benefit of the framework in the planning phase. Nevertheless, there certainly is a small improvement in the planning phase. The actual time saving happens in the creating the scenario phase – in the creating security holes sub-phase. If all desired security holes exist already in the framework, the security hole sub-phase is completely automated and can be completed within minutes. Depending on the framework vulnerability coverage, the latter sub-phase

can be completely automated, semi-automated or not automated at all. To estimate the actual time savings, one needs to know the percentage of time spent on creating gameplay content within creating the scenario phase, the percentage of time spent on creating security holes within creating the gameplay content and how many vulnerabilities that are needed for the scenario are pre-scripted in the framework. The actual time-saving in percentage can be calculated as S

$$S = T_{sh} * i,$$

where T_{sh} is the time in percentages spent on creating security holes and i is the percentage of the desired vulnerabilities automated in the framework. According to the RangeForce practice and Figure 23, the T_{sh} can be in the range of 0% and 40%. The upper bound is possible if no time is spent on network, machines, flags and user content. The lower bound is possible in case no security holes are created at all. By assuming equal distribution everywhere and all vulnerabilities are covered within the framework ($i = 1$), the T_{sh} can be calculated as $T_{sh} = S = 40 * 0.33 * 0.33 \sim 4.44$, which means that 4.44% of the time can be saved by using the proposed framework. The latter estimation is rather meant for larger cyber security exercises such as Locked Shields, where creating target machines, network, scenario user content and flags will also take a considerable amount of time.

6. Conclusion

The goal of the paper was to reduce the amount of time spent on preparing vulnerable machines. The author proposed to mitigate the problem via the automated vulnerability application framework. As a result of the thesis, the design of the proposed framework was created and implemented as Metapply tool (Appendix 1). The architecture of the framework was built keeping vulnerability collision detection and prevention in mind. The framework defines the lifecycle of the application of the vulnerability – preconditions, configuration and postconditions phases of a scenario. The scenario concept is defined using the framework domain specific language to allow users to describe which vulnerabilities should exist in target machines. The Metapply (as a prototype implementation of the framework) was used in order to measure the performance of the proposed method. According to the tests, applying two already existing vulnerabilities to a target machine takes around 30 seconds; however, manual approach can take up to 5 minutes per vulnerability if one knows exactly all the steps needed for vulnerability application and more than an hour for cases where the steps are not known. In the case of cyber security exercises, the proposed approach can save up to 40% of the preparation time, for more comprehensive exercises such as Locked Shields, where preparation time involves network, machine, flag and user content preparation, around 4% of the preparation time can be saved. The goal of the paper was achieved, yet, the methodology needs additional work in order to apply it on a larger scale – Metapply is the first prototype of the framework and does not contain all the possible vulnerability creation scripts and conflict avoidance rules.

7. Further work

The current paper was written in order to reduce the amount of time spent on preparing vulnerable machines. The author proposed creating a vulnerability application framework, implemented a prototype and predicted that with this approach, up to 40% of overall cyber security exercise preparation time can be saved. The prototype itself is not production ready and therefore needs a follow-up work to analyze the tool performance in practice and build up the community of users. The author introduced the conflict table of OS level abstraction layer functions (conflicting Ansible actions) to prevent conflicts between the vulnerabilities, yet, the analysis of all the conflicts between the Ansible actions remained unsolved due to the huge number of Ansible modules and lack of usage of the tool. The content of the framework (vulnerability creation scripts) is currently not enough to satisfy a large user base and all the vulnerability needs of cyber security exercises, therefore, additional work is needed to improve the collection of vulnerability creation scripts.

References

[1] D. M. Ritchie and T. Ken, "The UNIX time-sharing system," *Communications of the ACM,* vol. 17, no. 7, pp. 365-375 , 1974.

[2] MITRE, "CVE - Frequently Asked Questions," The MITRE Corporation, [Online]. Available: http://cve.mitre.org/about/faqs.html. [Accessed 16 April 2018].

[3] SSH Communications Security, Inc., "SSH Protocol," SSH Communications Security, Inc., 29 August 2017. [Online]. Available: https://www.ssh.com/ssh/protocol/. [Accessed 19 April 2018].

[4] Rapid7, "Metasploit," Rapid7, [Online]. Available: https://www.metasploit.com/. [Accessed 16 April 2018].

[5] Z. C. Schreuders, T. Shaw, M. Shan-A-Khuda, G. Ravichandran, J. Keighley and M. Ordean, "Security Scenario Generator (SecGen): A Framework for Generating Randomly Vulnerable Rich-scenario VMs for Learning Computer Security and Hosting CTF Events," in *26th USENIX Security Symposium*, Canada, 2017.

[6] HashiCorp, "Introduction to Vagrant," HashiCorp, [Online]. Available: https://www.vagrantup.com/intro/index.html. [Accessed 16 April 2018].

[7] Oracle Corporation, "Virtual box manual," Oracle Corporation, [Online]. Available: https://www.virtualbox.org/manual/ch01.html. [Accessed 16 April 2018].

[8] VulnHub, "VulnHub," VulnHub, [Online]. Available: https://www.vulnhub.com/. [Accessed 17 April 2018].

[9] Rapid7, "Metasploitable," Rapid7, [Online]. Available: https://information.rapid7.com/metasploitable-download.html. [Accessed 16 April 2018].

[10] Puppet, "puppet-agent: What is it, and what's in it?," Puppet, [Online]. Available: https://puppet.com/docs/puppet/4.10/about_agent.html. [Accessed 16 April 2018].

[11] "SecGen Shellshock vulnerability scenario," [Online]. Available: https://github.com/SecGen/SecGen/blob/812ba7dab8cd4a09a2c61b0c36469d7fa44f0bfa/scenarios/examples/vulnerability_examples/shellshock_vulnerability.xml. [Accessed 14 April 2018].

[12] L. A. Caroline , "Shellshock Attack on Linux Systems – Bash," *International Research Journal of Engineering and Technology (IRJET),* November 2015.

[13] Uber, "Collection - Win clipboard data," Github, November 2017. [Online]. Available: https://github.com/uber-common/metta/blob/b0788c0070fc5c89571da3bbeec6df0d1030452b/MITRE/Collection/collection_win_clipboard_data.yml. [Accessed 16 April 2018].

[14] P. Hulin, A. Davis, R. Sridhar, A. Fasano, C. Gallagher, A. Sedlacek, T. Leek and B. Dolan-Gavitt, "AutoCTF: Creating Diverse Pwnables via Automated Bug Injection," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, Vancouver, 2017.

[15] Rapid7, "Apache mod_cgi Bash Environment Variable Code Injection (Shellshock)," Rapid7, [Online]. Available: https://www.rapid7.com/db/modules/exploit/multi/http/apache_mod_cgi_bash_env_exec. [Accessed 16 April 2018].

[16] MITRE, "Overview - What Is CWE?," The MITRE Corporation, [Online]. Available: http://cwe.mitre.org/about/index.html. [Accessed 16 April 2018].

[17] S. Engle, S. Whalen, D. Howard and M. Bishop, *Tree Approach to Vulnerability Classification.*

[18] P.-W. Tom , "Semantic Versioning 2.0.0," [Online]. Available: https://semver.org/spec/v2.0.0.html. [Accessed 19 April 2018].

[19] "SecGen Shellshock vulnerability metadata," [Online]. Available: https://github.com/SecGen/SecGen/blob/812ba7dab8cd4a09a2c61b0c36469d7fa44f0bfa/modules/vulnerabilities/unix/bash/shellshock/secgen_metadata.xml. [Accessed 21 April 2018].

[20] U.S Deparment of Commerce, "CVE-2014-6271 - Shellshock," [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2014-6271. [Accessed 16 April 2018].

[21] Debian, "Package: bash (4.4-5)," Debian, [Online]. Available: https://packages.debian.org/stretch/bash. [Accessed 16 April 2018].

[22] B. Meyer, "Applying 'design by contract'," *Computer,* vol. 25, no. 10, pp. 40 - 51, October 1992.

[23] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen and S. Zacchiroli, "Why do software packages conflict?," in *MSR '12 Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, Zurich, 2012.

[24] Debian, "Debian Policy Manual," Debian, [Online]. Available: https://www.debian.org/doc/debian-policy/#syntax-of-relationship-fields. [Accessed 16 April 2018].

[25] M. Eriksson and V. Hallberg, "Comparison between JSON and YAML for data serialization," School of Computer Science and Engineering Royal Institute of Technology, 2011.

[26] E. Bailey, "JSON and XML: Trade-offs and the Future," Tufts School of Engineering.

[27] W3C, "The Rule of Least Power," 23 Februrary 2006. [Online]. Available: https://www.w3.org/2001/tag/doc/leastPower.html. [Accessed 16 April 2018].

[28] R. Sherstobitoff, "Anatomy of a Data Breach," *Information Security Journal: A Global Perspective,* vol. 17, no. 5-6, pp. 247-252, 2008.

[29] R. E. Pike, "The “Ethics” of Teaching Ethical Hacking," *Journal of International Technology and Information Management,* vol. 22, no. 4, 2013.

[30] "Chef Client Overview," Chef Software, Inc., [Online]. Available: https://docs.chef.io/chef_client_overview.html. [Accessed 16 April 2018].

[31] SaltStack Inc., "SaltStack," SaltStack Inc., [Online]. Available: https://saltstack.com/. [Accessed 5 May 2018].

[32] Red Hat, "How Ansible works," Red Hat, Inc., [Online]. Available: https://www.ansible.com/overview/how-ansible-works. [Accessed 16 April 2018].

[33] Red Hat, "Ansible best pratices," Red Hat, Inc., 8 April 2018. [Online]. Available: http://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html . [Accessed 16 April 2018].

[34] Red Hat, "Ansible Python API," Red Hat, Inc, [Online]. Available: http://ansible-docs.readthedocs.io/zh/stable-2.0/rst/developing_api.html. [Accessed 16 April 2018].

[35] Red Hat, Inc, "Ansible callback hadler," GitHub, 20 Februrary 2018. [Online]. Available: https://github.com/ansible/ansible/blob/stable-2.5/lib/ansible/plugins/callback/__init__.py. [Accessed 16 April 2018].

[36] Red Hat, "Ansible modules," Red Hat, Inc, 16 April 2018. [Online]. Available: http://docs.ansible.com/ansible/devel/modules/list_of_all_modules.html. [Accessed 16 April 2018].

[37] CCDCOE, "Cyber Defence Exercise Locked Shields 2013 - After Action Report," 2013. [Online]. Available: http://www.ccdcoe.org/publications/LockedShields13_AAR.pdf. [Accessed 17 April 2018].

Appendix 1 – Metapply in Github

One outcome of the thesis was the prototype implementation of the proposed framework. The source code can be found in https://github.com/arturluik/metapply/tree/thesis.

Appendix 2 – Interview with RangeForce

RangeForce is an Estonian origin company that provides game-based online cyber security training for developers, devops and security experts. They already have an in-house built framework to deploy and maintain personalized cyber security trainings. Current paper involves a subset of their functionality and, therefore the interview with Margus Ernits (CTO of RangeForce) was conducted in order to understand the problem and scope. According to their experience, 1 hour of content takes 200 hours of work and the content creation is at the moment a semi-manual task. According to their experience, the hours are divided as

- ~20% planning (learner analysis, learning objectives, scenario, network topology design, attack phases, scoring principles)
- ~40% templates, scripts (scoring, attacks) and additional scripting
- ~40% testing, debugging, scaling (race conditions)

Appendix 3 – Prototype command line interface

```
usage: run.py [-h] [--scenario SCENARIO] [--search SEARCH]

Metapply vulnerability application framework.

optional arguments:
  -h, --help           show this help message and exit
  --scenario SCENARIO  Scenario to start
  --search SEARCH      Search vulnerabilities by keyword
```

Appendix 4 – Metasploit vulnerability collection

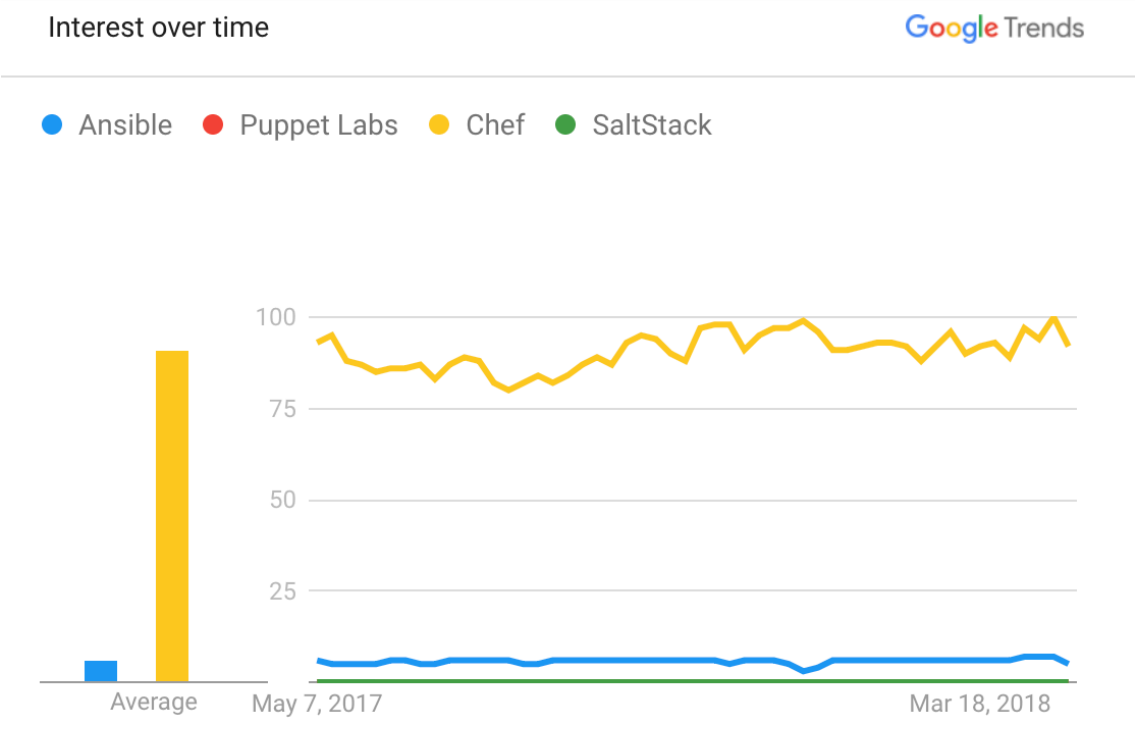
```
[!] Module database cache not built yet, using slow search

Matching Modules
================

   Name                                                Disclosure Date  Rank       Description
   ----                                                ---------------  ----       -----------
   auxiliary/scanner/http/apache_mod_cgi_bash_env      2014-09-24       normal     Apache mod_cgi Bash Environment V
ariable Injection (Shellshock) Scanner
   auxiliary/server/dhclient_bash_env                  2014-09-24       normal     DHCP Client Bash Environment Vari
able Code Injection (Shellshock)
   exploit/linux/http/advantech_switch_bash_env_exec   2015-12-01       excellent  Advantech Switch Bash Environment
 Variable Code Injection (Shellshock)
   exploit/linux/http/ipfire_bashbug_exec              2014-09-29       excellent  IPFire Bash Environment Variable
Injection (Shellshock)
   exploit/multi/ftp/pureftpd_bash_env_exec            2014-09-24       excellent  Pure-FTPd External Authentication
 Bash Environment Variable Code Injection (Shellshock)
   exploit/multi/http/apache_mod_cgi_bash_env_exec     2014-09-24       excellent  Apache mod_cgi Bash Environment V
ariable Code Injection (Shellshock)
   exploit/multi/http/cups_bash_env_exec               2014-09-24       excellent  CUPS Filter Bash Environment Vari
able Code Injection (Shellshock)
   exploit/multi/misc/legend_bot_exec                  2015-04-27       excellent  Legend Perl IRC Bot Remote Code E
xecution
   exploit/multi/misc/xdh_x_exec                       2015-12-04       excellent  Xdh / LinuxNet Perlbot / fBot IRC
 Bot Remote Code Execution
   exploit/osx/local/vmware_bash_function_root         2014-09-24       normal     OS X VMWare Fusion Privilege Esca
lation via Bash Environment Code Injection (Shellshock)
   exploit/unix/dhcp/bash_environment                  2014-09-24       excellent  Dhclient Bash Environment Variabl
e Injection (Shellshock)
   exploit/unix/smtp/qmail_bash_env_exec               2014-09-24       normal     Qmail SMTP Bash Environment Varia
ble Injection (Shellshock)


msf >
```

Appendix 5 – Google Trends about automated configuration tools (5.05.2018)



Worldwide. Past 12 months. Web Search.