Master's thesis

NTNU
Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Mihkal Dunfjeld

Cyber security testbed provisioning using a domain specific language

Master's thesis in Information Security

June 2019

NTNU
Norwegian University of Science and Technology

Mihkal Dunfjeld

Cyber security testbed provisioning using a domain specific language

Master's thesis in Information Security
June 2019

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Norwegian University of
Science and Technology

Cyber security testbed provisioning using a domain specific language

Mihkal Dunfjeld

01-06-2019

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Information Security and Communication Technology
Norwegian University of Science and Technology,

Supervisor: Assoc. Prof. Basel Katt
Co-Supervisor: Assoc. Prof. Erik Hjelmås
Co-Supervisor: Danny Lopez

Preface

This master thesis is the culmination of two years on the Information Security program at the Norwegian University of Science and Technology in Gjøvik. The project ran during the spring semester of 2019 and was one of multiple projects from the Norwegian Cyber Range as part of their goal to create a digital arena for education in cyber security.

Gjøvik, 01.06.2019

Mihkal Dunfjeld

Acknowledgment

I would like to thank my supervisor Assoc. Prof. Basel Katt and co-supervisor Assoc. Prof Erik Hjelmås. Thanks to Danny Murillo Lopez for helping with so many things throughout the project. Thanks to Jannis Schaefer for providing insightful feedback. Thanks to Michaela for help with proofreading.

(M.D)

Abstract

Practical assignments in a cyber security testbed is an effective method to train students in the field of cyber security. Setting up these training environments is a complex procedure that requires time, resources and knowledge that limits those who can carry it out successfully to a small group with the consequence that practical cyber security training is available to fewer than what is desirable.

In this thesis, two contributions are made in an attempt to reduce the impact of this problem. A domain specific language based on YAML is designed to have the abstractions necessary to model the components of a scenario in a cyber security testbed. A DSL is a language specific to an application domain that allow users to define their problem in a concise manner. A benefit with using a DSL is that it hides many of the unnecessary implementation details normally found in traditional programming languages.

The other contribution is a compiler that transforms the DSL into low level artifacts that are based on OpenStack Heat and Ansible in order to automatically provision the testbed based on the scenario that is defined in the DSL.

Contents

Preface i
Acknowledgment ii
Abstract iii
Contents iv
List of Figures vii
List of Tables viii
Listings ix
1 Introduction 1
1.1 Topic covered 1
1.2 Problem description 1
1.3 Justification, motivation and benefits 2
1.4 Scope 2
1.5 Research questions 3
1.6 Planned contributions 3
2 Background 4
2.1 Cloud provisioning 4
2.1.1 Infrastructure as code 4
2.1.2 Configuration management 5
2.2 Cyber security exercises 6
2.2.1 Capture the flag 6
2.2.2 Red team/Blue team exercises 7
2.2.3 Norwegian Cyber Range 9
2.3 Syntax formats 9
2.3.1 JSON 9
2.3.2 XML 9
2.3.3 YAML 9
3 Related work 11
3.1 Cyber security testbeds 11
4 Theoretical Framework 13
4.1 Model-driven development 13
4.1.1 Domain Specific Languages 13
4.2 Case study research 14
5 Methodology 17
5.1 Language modelling 17

5.2 Development . . . 17
5.3 Verification . . . 17
6 DSL for security scenarios . . . 19
6.1 Abstract syntax . . . 19
6.1.1 Scenario . . . 20
6.1.2 Node . . . 21
6.1.3 Router . . . 23
6.1.4 Vulnerability . . . 23
6.1.5 Service . . . 23
6.1.6 Challenge . . . 24
6.1.7 Team . . . 24
6.1.8 Agent . . . 24
6.1.9 Phase . . . 25
6.1.10 Objective . . . 25
6.1.11 Rule . . . 25
6.2 Concrete syntax . . . 25
7 Technical design and implementation . . . 27
7.1 Compilation process . . . 27
7.1.1 Compilation output . . . 29
7.2 Compiler design . . . 29
7.3 Provisioning pipeline . . . 30
7.4 Low level design decisions . . . 32
7.4.1 Infrastructure components . . . 32
7.4.2 Software components . . . 35
8 Case study . . . 38
8.1 Overview . . . 38
8.1.1 Data collection . . . 39
8.1.2 Analysis procedures . . . 39
8.2 Case study 1 - NCSC 2019 Jeopardy . . . 40
8.2.1 Replicating NCSC part 1 using the original method . . . 40
8.2.2 Replicating NCSC part 1 using CTFgen . . . 42
8.2.3 Analysis . . . 44
8.3 Case study 2 - Attack & Defense . . . 46
8.3.1 Creating an attack & defense environment with CTFgen . . . 47
8.3.2 Analysis . . . 49
8.4 Validity . . . 50
9 Discussion . . . 51
10 Conclusions . . . 53
10.1 Future work . . . 53
Bibliography . . . 54

Appendix A Interview questionnaire **56**

List of Figures

1 The thesis broken down into its major steps 17
2 Conceptual entity-relation diagram illustrating the relationship between the concepts 20
3 Compilation process broken down into major steps 27
4 Class diagram of the compiler 30
5 Sequence diagram displaying the process of provisioning a scenario 31
6 Network architecture for jeopardy and wargame CTFs 33
7 Network architecture for Attack & Defense competitions 34
8 Network architecture for a single red team/blue team environment 35
9 Network architecture for the jeopardy part of NCSC 2019 41
10 NCSC 2019 Attack & Defense network architecture 46

List of Tables

1 Characteristics of domain specific and general purpose languages [1] 14
2 Table shows which concepts are applicable to the different types of scenarios 19
3 Status of components in the attack & defense environment created with CTFgen . . . 47

Listings

6.1 General structure of an object 25
6.2 Overall structure of a scenario 26
7.1 Node running apache with default settings 36
7.2 Node running apache where port is changed to 8000 36
7.3 Example definition of two challenges 37
8.1 Jeopardy part of NCSC 2019 defined in the DSL 42
8.2 DSL definition of AD scenario that resembles the NCSC 48

1 Introduction

This chapter will give an introduction to the current situation related to cyber security testbeds in order to get an understanding of the premise of the project and why it is conducted.

1.1 Topic covered

A good way to learn practical cyber security is to have a disposable training environment where students can practice without worrying about breaking anything. This is where cyber security testbeds come into play. These are environments that can be set up to focus on both offensive and defensive aspects of cyber security. A cyber security testbed is the digital equivalent of a shooting range where specific scenarios can be organized to make the training more targeted.

Practical labs are currently one of the preferred methods for educating students at NTNU and elsewhere. The goal of this project is to improve the current system and make it more easy to use.

1.2 Problem description

Cyber security exercises and capture the flag competitions requires infrastructure to run on. The infrastructure used by students at NTNU today to practice cyber security is currently hosted in a private OpenStack cloud named SkyHigh. OpenStack allows the user to create define how the infrastructure will look like in a Heat template and instantiate it to a virtual environment. Heat's task is limited to creating the environment. Configuring the nodes is a task that needs to be handled by a configuration management system. Heat coupled with a configuration management system does work to some extent, however these applications are generic and not intended to create any specific type of environment and thus do not accurately represent the components that a cyber security scenario consist of.

The technical skills needed to realize a scenario manually or using Heat with a configuration manager is likely to put severe limitations on the pool of people capable of implementing them. There is a large knowledge gap in the requirements between being able to design a scenario and actually implementing it.

For this reason, there is a desire to automate this process and add a layer of abstraction such that the knowledge gap is reduced and environments can be rapidly deployed to an arbitrary number of students without having to worry about implementation details.

1.3 Justification, motivation and benefits

Currently, the deployment of infrastructure in SkyHigh is a semi-automatic process where some manual configuration is needed. It is a difficult and time consuming effort for those who organize practical cyber security events to set up and maintain everything that is needed to have a functional testbed. Thus, the primary motivation behind this thesis is to reduce the knowledge, time and resources spent with configuring these environments.

The outcome of this thesis will primarily benefit instructors and students but also the NTNU institution. Instructors benefit from this because they are able to create a reproducible scenario environment which saves them a lot of time. It also closes the gap between design and implementation such that instructors can avoid low level details and instead focus on improving other areas of the assignments.

The students benefit because assignments they're given may be more advanced due to the reduced complexity of creating the assignments. This may result in an increased learning outcome for the student.

This in turn also benefits NTNU, as it increases students learning efficiency and improve the skillsets of the students that graduate from NTNU. Improving the curriculum yields more satisfied and more capable students, which reinforces the reputation and position of NTNU as a top technical university in industry and society.

This thesis is a part of the Norwegian Cyber Range project at NTNU. The findings of this thesis could prove valuable in the development of the Norwegian Cyber Range.

1.4 Scope

The size of this project is potentially very large. In order make it manageable for one person within the time frame of one semester, it is necessary to narrow down the scope.

- The target platform for this thesis is limited to OpenStack. Other cloud platforms, physical environments and local virtualized environments are out of scope.
- The product is intended to be a proof-of-concept. There will be missing components and is not expected to hold production quality at the end of this thesis.
- This thesis is limited to creating a domain specific language and its accompanying deployment software. The software is to do provisioning only where the goal is to get the environment into its intended state. Web interfaces, databases, monitoring and other management functions that are generally required for a production-ready environment are not part of the scope of this thesis.
- The DSL's node abstraction will support any contemporary operating system, but the implementation will be limited to only include linux-based operating systems.

1.5 Research questions

We have two research hypotheses for this thesis. The first hypothesis is that a cyber security scenario defined in a DSL and that is compiled into an appropriate underlying technology can be used to reduce the time and effort it takes to create a cyber security scenario.

Due to the difficulty of implementing a cyber security testbed, the person who designed the scenario may not have the technical skills needed to implement it. This person is dependent on having access to personnel who possess this knowledge. There is a large gap in the knowledge required to design a scenario and the knowledge required to implemented it. This is the basis for the second research hypothesis which states that this system can be used to reduce the knowledge gap between a scenario designer and the infrastructure developer.

In light of this, we ask the following questions:

Q1. Can we reduce the time, resources and technical knowledge needed to create cyber security testbeds by using model-driven development methodology?

This question can be further broken down into two subquestions:

Q1.1 Can we develop a domain specific language which represent the abstractions needed to define a cyber security testbed?

Q1.2 Can we develop a proof-of-concept compiler that automatically provisions the testbed based on the developed DSL?

Q2. Can this system be used to reduce the knowledge gap between the scenario designer and the infrastructure developer?

1.6 Planned contributions

The goal is to make two primary contributions. The first is a domain specific language that allows the scenario designer to define a cyber security testbed in text-based language. The second contribution is a compiler that transforms the DSL into low level artifacts that are based around infrastructure provisioning and configuration management tools that can be used to automatically create the environment as it is defined in the DSL.

2 Background

This chapter explains some of the essential background knowledge that this thesis relies on. Section 2.1 introduces the underlying cloud technology that is used. Section 2.2 introduces the various types of cyber security scenarios that exist. Section 2.3 looks at the multiple file formats that were considered.

2.1 Cloud provisioning

This section details the underlying technology that this thesis is based on. To fully automate the creation of a cyber security testbed, a software stack of consisting of two components is needed. This includes an infrastructure provisioning tool and a configuration management system. These tools and the reasons they were chosen are discussed in the sections below.

2.1.1 Infrastructure as code

For automated deployment of servers and networking components, we have chosen to use Heat [2]. There is to the best of our knowledge only one serious contender to Heat on the OpenStack platform, and that is Terraform. There are several reasons Heat is preferred over Terraform as seen in the list below.

- Heat is Openstack's native tool for infrastructure provisioning and should therefore have good support for all of Openstacks features.
- Heat is the de facto standard tool for infrastructure provisioning among the Openstack users at NTNU Gjøvik and there are people with considerable expertise using it. Access to this expertise is very beneficial if unexpected situations occur and assistance is required.
- Heat has been around longer and is more mature than Terraform.
- The author has experience with Heat.

Terraform does have an edge in the fact that it is platform agnostic, something that Heat is not. However, this would allow the testbed to be created on any other cloud platform supported by Terraform. Due to the high criticality of the functions performed by the chosen tool, extra emphasis was put on maturity. Thus, despite not being platform agnostic, Heat is seen as the preferred tool.

Heat allows instructors to define a virtual infrastructure using YAML formatted templates. However, while Heat is a powerful tool it does have drawbacks. Firstly, it is limited to infrastructure which is too narrow for our purpose. It is possible to inject code into the virtual machines upon boot for bootstrapping purposes, although this feature has limits on how many bytes can be transferred which severely limits its usefulness. Secondly, the Heat templates are structured in way that

is not immediately intuitive to understand and the templates can become rather large for big infrastructures. It's possible to hide many of the unnecessary details using nested Heat templates and environment files to make things more manageable but this is only a viable option for a static infrastructure. If the infrastructure needs to be redesigned, the Heat templates must be changed accordingly.

Due to these limitations, Heat alone is insufficient for our purpose, however when combined with a configuration management system, it becomes a viable foundation to build upon.

2.1.2 Configuration management

When virtual machines are deployed in OpenStack they are created from a static image with little to no software installed. Therefore, automated post-deployment configuration and installation of software and vulnerabilities are necessary to make the scenario as realistic as possible. A number of configuration management tools exist that are potentially capable of achieving our goal but Puppet [3] and Ansible [4] were the primary contenders.

Puppet's main advantages is that it is the most mature tool with a wide variety of modules. It is also the de facto standard tool at NTNU, as it's taught to students and is used by IT staff. This results in an incredibly competent Puppet community at NTNU which could be provide assistance if needed. Its disadvantages are that it has its own domain specific language, which means that dealing with puppet files from Ansible is likely to be problematic. Also, it requires that each VM is configured with a Puppet client, which leaves a large footprint on the VM.

Ultimately Ansible was chosen as the configuration management backend primarily for the following reasons:

- It uses SSH to connect to clients which means that no software is required on the client side. Because of this, virtually any device that supports SSH can be configured with Ansible.
- Configuration is done using the YAML format which is also used to configure Heat. In addition to being easy for humans to understand, it's also integrated very well with Ansible. A YAML file that is loaded into memory in Ansible turns into a dictionary object, which makes data structure manipulation incredibly easy. An added benefit of this is the potential to compile input data directly into Ansible templates.
- It can operate in both push and pull mode. This makes it very flexible as the server can either push changes to the client or the client can pull changes from the server.

At the most basic level, in Ansible terminology a *task* refers to a single configuration change. One or more logically related tasks are organized into *playbooks* to execute a set of operations on a node. A playbook can be executed directly or it can be a subset of a *role* to achieve a higher level goal. For example, installing and configuring a piece of software may require a sequence of tasks so these are added to a playbook. This sequence may differ depending on the environment, so multiple playbooks are created to accommodate the various environments. By organizing all these playbooks into a role the goal can be achieved regardless of the environment. Another important aspect of Ansible is the inventory. The inventory is used to group nodes logically depending on

their role and configuration. Each node in the inventory is also associated with an IP address or hostname which is used when applying the configuration to a particular group.

The software created in this thesis makes extensive use of inventories, roles and playbooks to apply almost all software configuration necessary to transform the scenario into the state defined by the user using the domain specific language.

2.2 Cyber security exercises

Many different terms are used in the context of cyber security scenarios that must be clarified. Such scenarios are commonly referred to as capture the flag (CTF) exercises and are used synonymously. The goal in a CTF is to obtain a flag that awards points through the means of solving security related tasks or exploiting vulnerabilities to gain or elevate access in a system.

2.2.1 Capture the flag

Several types of CTF competitions exist. These will be explained in further detail in the following sections.

Jeopardy

In jeopardy style CTFs, players are participating either individually or as a team to solve challenges, typically in the categories of web, binary exploitation, forensics, cryptography, mobile, steganography or reverse engineering. Points are awarded by submitting flags as challenges are completed. The amount of points a challenge awards depends on its difficulty and at the end, it is the team that has accumulated the most points that is the winner.

Challenges in this type of exercise vary from files that the participants download and work with on their own computer to more advanced challenges, where multiple services may be involved. Because of this, hosting such scenarios requires few resources on the organizers side. Due to its simplicity, jeopardy style CTF is the most prevalent type of CTF competition. This also means that the concept of colored teams is inapplicable to this type of scenario.

The challenge section of HackTheBox [5] and the Norwegian National Security Authority's (NSM) challenges [6] are good examples of publicly available jeopardy style CTF exercises.

Attack & Defense

In Attack & Defense style scenarios, each participating team takes the role of both blue and red team. Teams are assigned a lab environment that include one or more vulnerable services which they must defend against other teams while simultaneously attacking other teams' environments. To ensure focus on the vulnerable application, the environment is often rather simple. This often means that custom applications are used to avoid having patches publicly available, which in turn forces teams to find the vulnerability and patch it themselves.

A central game server manages the flags for each team and replaces them at certain intervals. To score points, teams must exploit the opposing teams application to gain access to their flag and submit it to the game server. This process must be done every time the flag is renewed. Teams lose points when opposing teams submit their flag.

The concept of service level agreements are used to ensure that team keep their service reachable. The game server will check the availability of each teams servers and points will be deducted if the service is offline.

When hosting an attack & defense scenario, the infrastructure must be configured to avoid a situation where a team can use firewall rules to block all traffic except traffic that originates from the gameservers service checker. The way this issue is commonly resolved varies, although alternatives include having every host connect to each other through a VPN or a proxy to obfuscate the source IP address. Another option is that teams are not allowed access to the server hosting the application but rather use a continuous delivery pipeline to send patches to their application while using a different machine to attack.

Due to all these requirements and the need for game server software and a custom vulnerable application, make hosting an attack & defense scenario a resource demanding effort.

Wargames

In wargame style CTFs, the participant solves challenges that get increasingly difficult. The player starts at the bottom of a ladder and works their way up by solving challenges. A challenge must be completed in order to advance to the next level. The winner is the person/team who got the highest up on the ladder or the person/team who climbed the ladder the fastest.

The main difference between wargames and jeopardy style CTFs is that in jeopardy, all challenges are completely unrelated to each other whereas in wargames they are not. Wargame challenges should stay within a chosen category and they should differ in how difficult they are to solve. Because of this, organizing wargames requires more effort than what is necessary for jeopardy.

OverTheWire [7] and SmashTheStack [8] are good examples of publicly available wargame style CTF exercises.

2.2.2 Red team/Blue team exercises

Red team/blue team exercises are scenarios that attempt to simulate real world environments as close as possible. This generally means that the network architecture may range from simple to very complex and with a wide variety of services running on the nodes within the network. Consequently, it is necessary that the tools for designing the architecture are flexible such that it can accommodate a wide variety of configurations.

Jeopardy and attack & defense style competitions generally narrow focus on operational aspects of cyber security, whereas red team/blue team scenarios may have a much wider scope. NATO's Locked Shields [9] is a good example of such an exercise. There the goal is to train all relevant groups of people that need to be involved from participants in the operations team to high level decisions makers that must act and react based on information provided from operations team.

This makes hosting a red team/blue team exercise a far more resource demanding effort than hosting any other type of scenario. This added complexity requires that support personnel is avail-

able for technical assistance and to fill the roles of various elements of the information chain. Consequently, actors in this type of scenario are assigned to a colored team to indicate their role.

Blue team

The blue team are the defenders of the systems in a scenario. This role is typically meant to simulate an IT team or the security team. The primary objective of the blue team is to secure the computers and networks to prevent the attackers from succeeding [9]. The blue team may apply a wide variety of methods to achieve their goal such as threat intelligence, forensics, reverse engineering or monitoring.

Red team

The red team is typically an external entity tasked with finding vulnerabilities in a system. They may also try to exploit these vulnerabilities to emulate a real world attacker and to discover flaws in the blue teams defense. The red team typically has to abide by rules of engagement that regulates which malicious actions they are allowed to take [9].

The red team may also be tasked with assessing and exploiting non-digital security barriers such as physical security and the organizational security of the company. Social engineering is a method used commonly for this purpose.

Purple Team

The purple team is more of a function rather than an actual team. The function is intended to improve the communication between the red and blue teams. Effectively, both the red and blue teams are on the same team, so purple teaming allows red teams to better communicate their findings to the blue team [10].

White team

The primary goal of a security scenario is usually to educate red and blue team members. The white team facilitates this goal by designing the scenario, set the objective and rules of engagement. They control all high level aspects of a scenario and often design the scenario to educate participants in one or more specific areas of cyber security [9].

Green team

The green team is responsible for developing and maintaining the infrastructure and services in the exercise. This includes most technical aspects of a scenario such as setting up the servers, creating scoring systems, networking, giving remote access to the environment and monitoring the health of the systems throughout the scenario [9].

Yellow team

In a security scenario, the purpose of the yellow team is to act as a liaison between the red and blue teams and the white team to provide situational awareness to the white team. This is done to emulate a situation where upper management must be able to make informed decisions in a timely manner. In NATO's Locked Shields exercise in 2013, the method to achieve this was to require the red and blue team to submit regular reports of the status quo [9].

2.2.3 Norwegian Cyber Range

The work for this thesis is part of a larger project. There are two other projects running in parallel that are essential to the overall goal of this project which is to create a platform that makes it easier to create cyber security testbeds.

The whole project goes under the name of *Pentesting Exercise Management Application* (PEMA) which contains three subprojects.

PEMA is the user-facing web application that users and instructors will use to interact with the system. *Pentesting Lab Environment Database* (PLED) is a database application that will contain vulnerabilities, malware, challenges and metadata information so that they can be installed.

The compiler and the DSL which is the work of this thesis is the backbone of the PEMA system. It is intended to tie these other projects together and create the scenario that the instructor defines through the web interface.

2.3 Syntax formats

During the initial planning of the project, it was necessary to find an existing file format to create the DSL in. By using an existing file format, we don't have to write software libraries to load and dump files. An existing library is also likely to have syntax validation which is a huge benefit.

Several file formats were considered. These are discussed in the following sections.

2.3.1 JSON

JSON (JavaScript Object Notation) is one of the most commonly used file formats, especially in web technology. Creating the DSL around JSON was considered early on. It's a data-interchange format that is designed to be easy for humans to read and easy for machines to parse and generate [11]. JSON uses two different data structures; key/value pairs and lists. These are designed to easily translate into data structures that are native to a wide variety of programming languages.

A major drawback of JSON is that it does not support comments. We consider the ability to write comments in the file to be a huge benefit to the user, and chose not to use JSON because of this.

2.3.2 XML

XML (eXtensible Markup Language) is a markup language that is designed to store and transport data [12]. XML was initially considered because it supports schemas that can be used to put constraints on the contents and structure of the file.

It was however dropped quite early on due to being excessively difficult to both read and write correctly. User-friendliness is an aspect we want to improve as seen in the second research question in section 1.5.

2.3.3 YAML

YAML (YAML Ain't Markup Language) is a data serialization language closely related to JSON although YAML has different priorities. YAML is designed to be easy to use and read for humans, to be portable between programming languages and to allow programmers to manipulate YAML using the native data structures of agile programming languages [13].

YAML stood out as a good fit for several reasons. It is easy to read and write and thus fits our goal in the second research question. It is well supported by most programming languages, but more importantly, is that it supported in Python which is the language we will be using. It is a practical benefit that both the infrastructure provisioning and configuration management tools we will be using are also YAML based. YAML also have solutions for schemas.

3 Related work

Problems similar to those addressed in our research questions have been partially addressed in the literature. This chapter will look into some of the literature related to provisioning of cyber security testbeds.

3.1 Cyber security testbeds

I-tee

Ernits et al. [14] states that cyber defense exercises are effective learning tools, but are rarely used due to how complex and time consuming task hosting such an event is. Therefore, they created a platform called I-tee which automates the setup and configuration of a virtualized testbed where students can practice. The platform features both legitimate and malicious background traffic generators to increase the realism. The platform also has a system for scoring.

While this platform is interesting, it does have drawbacks. Firstly, I-tee uses virtualbox as its virtualization platform. We believe that using a cloud-based platform will be more future-proof and will allow the exercises to scale beyond what is possible with the I-tee platform. With the increased popularity of cloud-based infrastructure, having hardware on-premises becomes increasingly rare. And considering how resource demanding hosting this type of event can be, it is likely that the infrastructure requirements will limit both the scale of the event and who has the possibility to host them.

Secondly, the I-tee platform requires technical skills that are normally only held by system administrators [15] and does not focus on lowering the knowledge requirement for deploying a cyber security scenario, which is a problem addressed in the second research question of this thesis.

SecGen

Schreuders et al. [16] also recognize that creating cyber security scenarios requires a lot of effort. Their angle is that VMs and challenges are static once created and that modifying these depends on reinvesting a lot of initial effort. The approach they take to solve this is to define a scenario using XML. In the scenario definition, certain parameters are randomized in order to make them dynamic. By randomizing parameters such as vulnerabilities, a VM can be generated multiple times with different vulnerabilities from the same source template [16]. Secgen is not suitable for our use case as it generates virtualbox VMs and NTNU wants to utilize its OpenStack platform. This thesis also aims to expand the idea of using a domain specific language to define full cyber security testbeds. Whereas, secgen allows you to define certain technical aspects of a scenario, a scenario may consist of other non-technical aspects as well, all of which will be defined in our DSL.

KYPO

Čeleda et al. [17] presents a cyber defense exercise platform based on the OpenNebula cloud platform. The KYPO platform is designed to run general purpose red team/blue team exercises. The work is interesting and shares many of the same aspects that we want to achieve in this thesis such as using a cloud platform and the ability to instantiate arbitrary network topologies. The work of this thesis however is dependent on OpenStack as the cloud platform. Our projects also have very different requirements. This thesis aims to make it easier to set up these environments, while KYPO has more functional requirements.

iCTF

Vigna et al. [18] developed a framework for hosting attack & defense style CTF competitions. The framework features a database that tracks the game state and a scorebot that automates checking of service availability and flag management. It also has a web interface where users can see the scoreboard and submit the flags. The system generates a virtual machine image that participating teams must transfer to their local machines.

The work is interesting as it focuses on CTFs rather than on cyber defense and especially the service checker and flag management modules stand out as potentially useful for this thesis.

Others

Chandra et al. [19] proposes a cyber warfare testbed that facilitates scenario-driven exercises. The testbed is virtualized using the XEN hypervisor.

Gephart et al. [20] presents a design for a virtual computer lab for hosting cyber security exercises. The system runs on dedicated hardware and uses XEN as the virtualization hypervisor.

CTFgen compared to related work

All the related projects listed above have some aspect with them that make them unusable for this thesis. The projects [14, 16, 19, 20, 18] are all based on a non-cloud platform and are therefore not usable. KYPO [17] does use a cloud platform based on OpenNebula but since we do not have access to an OpenNebula platform, it is not possible to use in this thesis.

All the above projects focus automating the provisioning pipeline in various ways although none of them aim to reduce the knowledge that is required to set cyber security testbeds. Improving the user-friendliness is something that we want to improve upon.

4 Theoretical Framework

This chapter explains the theory behind the methodology used in the thesis. The way this is used in practice is reviewed in chapter 5.

4.1 Model-driven development

Model-driven development is a software development method where a model is built to represent a target domain. The model is represented in a domain specific language that is then transformed by a compiler into artifacts of the underlying technology. In this thesis, we use a prescriptive model to automatically create the target system based on a domain specific language.

4.1.1 Domain Specific Languages

A *Domain Specific Language* (DSL) is a language that is designed to solve problems in a specialized and well defined application domain [1]. DSLs are contrasted by *General Purpose Languages* (GPLs) which are designed to solve problems in a wide and unspecific domain. DSLs use abstractions that are closely linked to its domain to allow the user to express their intentions very concisely. These abstractions manifest themselves as concepts that have a syntax specific to the language. A DSL allows users to focus on the problem alone and by removing the unnecessary overhead that comes with dealing with libraries and frameworks when using general purpose languages, one can see several positive side effects:

- Increased productivity because DSLs are semantically rich, meaning more functionality can be expressed with less code.
- It's easier to semantically validate the code.
- Overall quality of the product increases as unnecessary freedom is taken away from the user. This also increases code maintainability.
- Data longevity increases because a model can be expressed independently of its implementation.
- The absence of cluttering implementation details makes it suitable for use as a thinking and communication tool.

Execution method

Every computer language needs a way to be executed and there are two main approaches to do this. The first method is *compilation*, which is the process of translating computer code into a different language that the target platform is capable of executing. An instance of the C programming language for example, is compiled into native machine code. DSLs are generally higher level than GPLs so they are often translated into the source code of a GPL or into another DSL.

The second method is *interpretation*. Interpreted languages need an execution engine running on top of the target platform which continually loads and executes the code during execution of the program.

Domain specific languages vs General purpose languages

Generally, it can be said that a DSL sacrifices the ability to express a wide variety of problems to increase how concisely and efficiently a problem in the intended domain can be addressed. The situation is however not entirely black and white. A DSL may have characteristics from GPLs and vice versa. Table 4.1.1 shows the characteristics that are generally associated with DSLs and GPLs [1].

Characteristic	GPL	DSL
Application Domain	Broad, complex and unspecific	Special purpose and specific
Size	Large	Small
Turing completeness	Always	Almost never, although it is possible
User defined abstractions	Sophisticated	Limited
Execution	Via intermediate GPL	Native
Lifespan	Long (years or decades)	Months to years depending on context
Designed by	Guru or committee	Domain experts and engineers
User community	Large and widespread	Small
Evolution	Slow as its often standardized	Fast-paced
Deprecation	Almost impossible	Feasible

Table 1: Characteristics of domain specific and general purpose languages [1]

Components of a DSL

A DSL mainly consist of three components, an *abstract syntax*, *concrete syntax* and a *compiler/interpreter* [1]. The abstract syntax shows the internal structure of the language's application domain while ignoring the notation. Concrete syntax is the notation which the user interacts with directly. By interacting with the concrete syntax, the underlying abstract syntax is also modified. While the concrete syntax is typically textual, it may also be graphical or tabular. The generator is the piece of software that either compiles or interprets the concrete syntax and executes it on the target platform.

4.2 Case study research

The book *Case study research in software engineering* [21] defines case study research as follows:

> "*An empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified [21].*"

Because case studies are conducted on real world phenomena, it is necessary to balance the

degree of realism under which phenomena are observed and the level of control the researcher has over variables. The nature of a case study is such that isolating control variables decreases the realism and increasing the realism decreases the control.

Research is generally motivated mainly by four reasons [21]. Case studies are generally used for exploratory or descriptive reasons.

Exploratory: The goal is to observe one or more phenomena in order to gain knowledge and insight and to form hypothesis for later research.

Explanatory: The goal is to seek an explanation of a particular phenomenon.

Descriptive: The goal is to gain an understanding of the status quo of a particular phenomenon.

Improvement: The goal is to improve upon a particular aspect of a phenomenon.

This thesis will use a combination of improvement and descriptive research. We create the DSL and compiler as a contribution to the current toolset but it is also necessary to test the system when it's finished in order to measure its success.

The process of conducting a case study consists of five major phases that the research must consider [21]:

1. The first phase involves defining the rationale, objective, data selection, methodology and strategy as well as assuring that the case study holds an acceptable quality. A case study for software engineering project should also take into account that software can be complex and dynamic phenomena. For this reason, the design should be flexible and the research should take into account that certain elements may change during the case study.
2. In the second phase, data collection is planned in detail. This includes carefully selecting what data sources to use and how to collect the data. Runeson et al. [21] identifies five primary data collection methods for software engineering case studies. These are *interviews, focus groups, observations, metrics* and *archived data.* A software engineering project can be very complex and output can come in a vast variety of forms. It is therefore crucial that the researcher makes a thorough analysis of which data sources are relevant for the particular case. Case studies also tend to produce large amounts of raw data so the researcher should also plan how to organize the data to facilitate data analysis.
3. The third phase consists of executing the data collection planned in the previous phase.
4. In the fourth phase, the researcher analyzes the data collected during previous phases. The goal is to understand what happened in the studied case. The way this phase is conducted will vary greatly depending on whether qualitative or quantitative data is collected.
   The goal of qualitative analysis is to identify patterns, relationships or sequences that can be used to make generalizations. In qualitative analysis, this often requires an iterative process. As new insights are uncovered during analysis, there may be a need to go back and collect additional data. In fact, these phases should not be considered a sequence of steps, but rather general elements a case study should consist of. In reality, the researcher is likely go back and

forth between the phases in an iterative manner.
The researcher starts with organizing the collected data into logical groups. When this is done, the researcher can start to look for patterns or relationships that can be used to form a hypothesis. As more data is collected, this might strengthen or weaken the hypothesis. This process continues until a set of knowledge can be formalized as the final result. In order to allow readers of the research to understand how and why a particular conclusion is drawn, it is important that the researcher is systematic in their approach and that each step is documented. This will allow the reader to follow the chain of evidence and verify its trustworthiness.

Quantitative analysis is typically conducted using statistical methods. For this type of analysis, generally the research design is assumed to be fixed, meaning that the design is finalized before data collection begins. Changes cannot be made to the design after collection has started. Although it may be possible to use quantitative analysis in a flexible research design, changes must be accounted for. Depending on the topic, this may be difficult to achieve, so a qualitative approach is generally preferred for case studies.
5. The final phase consists of formalizing a report detailing the findings of the research.

Runeson et al. [1] identifies two approaches to designing case studies. In a *holistic case study* design, the case is studied as a whole. It is suitable in cases where the theoretical framework is also holistic and in cases where the observed phenomenon lasts over a relatively short amount of time. Holistic designs take a broad perspective on the case and do not go into deep details which can lead to insufficient amounts of data. The holistic approach is what we will be using in this thesis.

Embedded case study design is appropriate in cases where the phenomena are complex and the investigated case has a long duration. Embedded design is also more suited in cases where the units that are analyzed form logical subcategories [1].

5 Methodology

The thesis can be broken down into three phases as illustrated in figure 1. Modelling, development and verification. These phases are discussed below.

Figure 1: The thesis broken down into its major steps

5.1 Language modelling

In the modelling phase, the problem domain was closely examined in order to identify the concepts that were necessary to represent the types of scenarios that we wanted to support. As the relevant concepts were identified, it was necessary to know how the abstract concepts would map to the concrete abstractions used in the technology that would be used.

The abstract concepts were categorized into three groups depending on which concrete abstraction they map to. These categories are *infrastructure*, *software* running on the infrastructure and *external* concepts where the latter category holds all the concepts that do not map into the first two categories.

5.2 Development

In the development phase, we used *model driven development* to create the transformation engine that converts the abstract concepts in DSL format into concrete artifacts based on Heat and Ansible.

5.3 Verification

To verify and assess the performance of the system created during development, case studies are conducted. While the model targets four types of scenarios, the goal is to conduct a case study on two of them. Each case study aims to verify and assess the language's and compiler's ability to recreate a scenario of a particular type.

For each type of scenario, a relevant CTF or exercise will be used as a reference point that the new system will try to recreate.

Data analysis will follow a qualitative approach where observations are made during the cases.

Domain experts are brought in to test both the old and the new system. While the experts test the systems, their actions and thought process will be observed. After the testing is done, each expert is interviewed to provide additional feedback.

6 DSL for security scenarios

In this chapter, we start by going into details of the abstract syntax before looking at the concrete syntax.

6.1 Abstract syntax

This section goes into details of each element of the abstract syntax defined in the language. Abstract syntax is the set of components of a language that are semantically relevant to represent an application domain [1].

We use the term *concept* to refer to any element of the DSL that can be used to represent a component within the targeted application domain.

The domain specific language that has been created contains 11 concepts as seen in table 2. On its own, each concept is meaningless so they must exist in the context of a scenario. A scenario is also considered to be a concept, although it differs in nature because the other concepts exist within it.

There are four types of scenarios that this DSL aims to accommodate. Not all concepts are applicable to every type of scenario. Table 2 shows which concepts that are applicable to each type of scenario.

Concept	**Jeopardy**	**A&D**	**R&B**	**Wargame**
Node		X	X	
Router			X	
Challenge	X			X
Vulnerability		X	X	
Service		X	X	
Team	X	X	X	X
Objective	X	X	X	X
Rule	X	X	X	X
Agent		X	X	
User_account		X	X	
Phase	X	X	X	X

Table 2: Table shows which concepts are applicable to the different types of scenarios

The relationships between the concepts are seen in figure 2. The figure does not differentiate between the scenario types. Whereas the `challenge` concept is the center of the jeopardy and wargame style CTFs, the `nodes` with accompanying `services` and `vulnerabilities` are the main concepts for attack & defense and red team & blue team scenarios.

Note that the scenario concept is not included in figure 2. This concept is different from all the other concepts in that it is not possible to define directly. Instead, the other concepts are defined within a scenario.

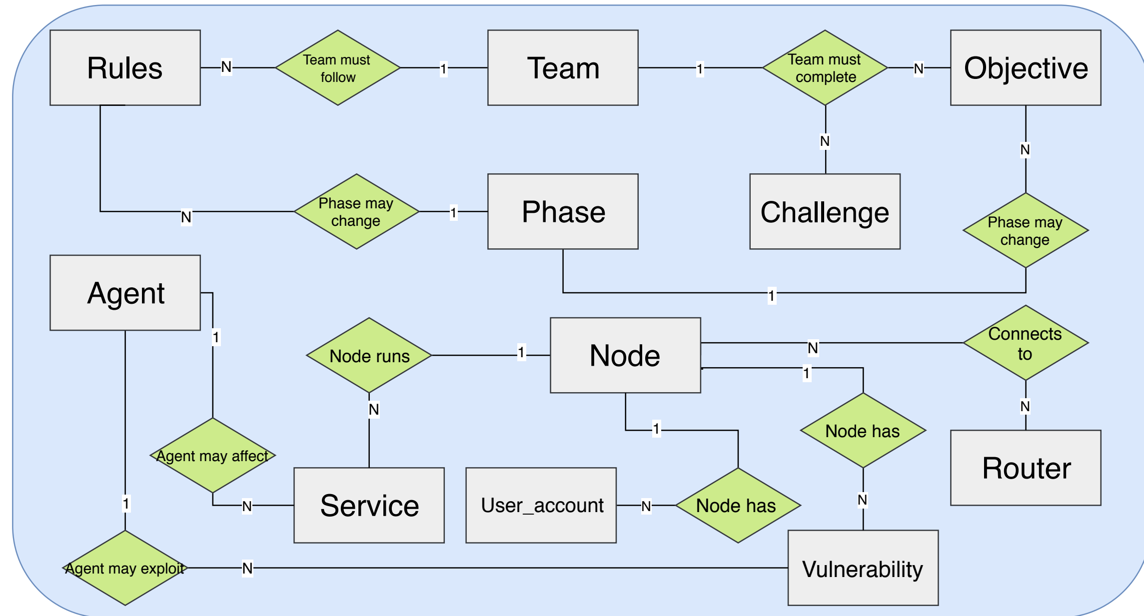


Figure 2: Conceptual entity-relation diagram illustrating the relationship between the concepts

The concepts have properties where information about them can be stored. The details of each concept and its properties are explained in the following sections.

6.1.1 Scenario

The `scenario` concept defines which type of scenario is being created and the general outline of it. It also creates an environment for the other concepts to exist in.

Properties

Name: Name of the event. Type: *string*.

Type: The type of scenario being created. Possible string values are: `jeopardy`, `attack-defense`, `wargame` and `redteam-blueteam`

Start_date: The date at which the scenario or CTF begins. Format: `dd.mm.yyyy`

End_date: The date at which the scenario or CTF ends. Format: `dd.mm.yyyy`

Start_time: The time when the scenario or CTF begins. Format in 24h time: `hh:mm`

End_time: The time when the scenario or CTF ends. Format in 24h time: `hh:mm`

Docker_hosts: Takes an *int* as a value. The value represents the number of VMs used to run docker containers from. `docker_hosts` is an optional property and if not defined a default setting of

2 is used.

Objectives: A list of overall objectives for the scenario as a whole. Values are given in a list according to YAML's specification where each entry is a reference to an `objective` object.

Agents: A list of agents that are active during the scenario/CTF across all phases. Values are given in a list according to YAML's specification. Each entry is a reference to an `agent` object.

Rules: A list of rules of that apply for the entire scenario/CTF across all phases. Values are given in list according to YAML's specification. Each entry is a reference to a `rule` object.

Teams: A list of teams participating in the scenario/CTF. Values are given in a list according to YAML's specification. Each entry is a reference to a `team` object.

It's important to note that properties that are lists such as `objectives`, `agents`, `rules` and `teams` do not contain the definition of these concepts. Instead, each entry of the list is a reference to a concept defined as its own object.

6.1.2 Node

A node represents a virtual machine participating in a scenario or CTF.

Properties

Type: Nodes are defined by setting the type property to `node`. Type: *string*

Flavor: OpenStack allocates the amount of RAM, CPU and disk space to a VM based on the flavor it has. Possible values map directly to the set of available flavors in OpenStack. Flavor is an optional property of type *string* and if not specified a default setting is used.

OS: The `OS` property refers to the particular operating system that is specified. Possible values map directly to the OpenStack images that are available. In the backend it also distinguishes between Linux and Windows based images out of necessity because the two platforms have different requirements and dependencies. *OS* is an optional property of type *string* and if not specified a default setting is used.

Public_IP: Type: *bool*. Refers directly to OpenStacks concept of *floating IPs*. A VM created in OpenStack is by default not accessible from the outside. In order to do so, a floating IP is needed. By setting `public_ip` property to true, the VM will be assigned a floating IP and thus be accessible from the outside. `public_ip` is an optional property and if not specified it will default to `false`.

Networks: The `networks` property is a list of networks that the node is connected to. Each network is represented as a collection of two or more properties.f The `networks` property is a required property and must contain at least one network.

- **Router**: Mandatory property of type *router*. The name of the router which the network belongs to. For attack & defense scenarios value must be `management`
- **Subnet**: Mandatory property of type *string*. The name of the subnet to connect to. The specified subnet must exist on the router specified on the *router* property. For attack & defense scenarios value must be `attack_defense_subnet`
- **Port_security**: Possible values are TCP and UDP. Both have lists of integers which repre-

sents ports to open for each respective protocol. By default all ports are closed and must be explicitly opened. Because SSH is necessary for configuration, TCP port 22 is opened even if not explicitly done so. ICMP is also allowed for debugging purposes. Port_securty is an optional property.

Vulnerabilities: The `vulnerabilities` property is a list of vulnerabilities that are installed on the node. The name/id of each vulnerability corresponds to a vulnerability in a database which must exist. Unless explicitly specified, the vulnerability will use its default settings. A vulnerability may have its own specific properties which can be changed. If this is the case, then a new object of type `vulnerability` must be created as a child of the `resources` tree. The object must have the same name or id as in the nodes vulnerabilities list. The idea behind this is to separate the configuration of a vulnerability from the node configuration to avoid having few extremely large top level objects.
Type: *string*.

Services: The `services` property is a list applications running on the node. These applications are intended to provide meaningful content to the node without being intentionally vulnerable. The difference between a vulnerability and a service is that a vulnerability is purposefully vulnerable whereas a service is not. Each listed entry is used to uniquely identify a service in a database and must be present in the database. A service may have specific properties that the user may want to override. If this is the case, then a new object of type `service` must be created as a child of the `resources` tree.
Type: *string*.

User_accounts: The `user_accounts` property is a list of users on the node where each user is a collection of properties. By default a node only has the default user and the root user which is insufficient to simulate real world scenarios.

- **Username**: type: *string*. Username of the new user.
- **Name:** type: *string*. The users full name as seen in the Unix gecos field.
- **Password:** type: *string*. Password must be entered in the form of a hash.
- **Uid:** type: *string*. The user's UID.
- **Gid:** type: *string*: The user's primary group ID.
- **Group:** type: *string*. Option to override the default primary group.
- **Groups:** type: list of strings, where each entry is a group that the user is added to.
- **Home:** type: *string*. Home folder for the user.
- **ssh_key:** type: list of ssh keys public keys that can be used to access the user.
- **Shell:** type: *string*. The user's shell. Defaults to `/bin/bash`

Note that the `user_account` maps directly to an Ansible role. This role may be swapped out with another role in the compilers configuration file. Other roles may have different property names. The `user_accounts` capability is determined by the selected Ansible role.

6.1.3 Router

The `router` provides network functionality to the nodes.

Properties

Type: Routers are defined by setting the type property to `router`.

Networks: The `networks` property contains all subnets present on the router. It's structured as a tree where each subnet is a branch under the `networks` root. Each subnet has several sub-properties which are listed below.

- **CIDR:** The IP range of the subnet. `cidr` is an optional property and if not specified, a /24 subnet will be automatically allocated. Type: *string*.
- **GatewayIP:** The gateway IP of the router. `gatewayIP` is an optional property and if not specified, the first address in the range is assigned. `gatewayIP` cannot be specified without also specifying the `cidr` property and `gatewayIP` must be a valid IP address in the chosen IP range. Type: *string*.
- **Routes:** The `routes` property is a list of static routes to other subnets. This property is needed because OpenStack by default lacks support for dynamic routing protocols so routes to other networks must be statically defined. Each static route is defined as a key-value pair where the key is the destination subnet and the value is the next-hop IP address. `routes` is an optional property and the list remains empty if nothing is defined. Type: list of strings.

6.1.4 Vulnerability

A `vulnerability` represents an intentionally vulnerable component of a node and are defined by setting the type property to `vulnerability`.

Properties

Type: Vulnerabilities are defined by setting the type property to `vulnerability`.

Vulnerabilities will only have properties that are specific to each vulnerability. The properties may have any type that is valid YAML. This is defined entirely by the vulnerability itself. These properties can be used to override default values that the vulnerability might have. The documentation for the specific vulnerability should be used to get the specific property names.

6.1.5 Service

A `service` is application running on a node. Services differ from vulnerabilities in that vulnerabilities are intentionally vulnerable whereas services are not. Services are defined by setting the type property to `service`.

Properties

Type: Services are defined by setting the type property to `service`.

Services will have properties that are specific to each service. The properties may have any type that is valid YAML. This is defined entirely by the service itself. These properties can be used to override

default values that the service might have. The documentation for the specific service should be used to get the specific property names.

6.1.6 Challenge

A challenge is an independent exercise available in jeopardy style CTFs. Challenges can come in various shapes and forms from simple text or files to advanced container applications. The properties of a challenge depend on what type of challenge it is. The list is likely to be extended as support for other types are added.

Properties

Type: Challenges are defined by setting the type property to `challenge`.

Points: The amount of points awarded for completing the challenge. Type: *int*.

Port: The port to access a challenge that is container based. Type: *int*.

Prerequisites: A list of prior challenges that must be completed before this challenge can be accessed. Type: list of strings.

6.1.7 Team

The `team` concept corresponds to one or more individuals participating in a scenario or a CTF. This concept is intended to keep track of scoring and to designate to whom an environment belongs to.

Properties

Type: Teams are defined by setting the type property to `team`.

Members: The `members` property is a list that contains information and contact details about each individual member of a team. Type: list where each entry is a string.

Team_type: Whether the team is a red or blue team. Optional property that is only relevant for red team & blue team scenarios.

6.1.8 Agent

An `agent` represents an event the participant must react to or an action the participants must perform. The concept is inspired by NATO's Locked Shield exercise [9] where blue teams were required to hand in situation reports and by I-tee [14] where blue teams need to react to automated attacks performed by bots. While this is the inspiration, the `agent` is intended to be much broader than that. It's intended to simulate almost any occurrence of almost any nature during a scenario that forces the participant to assess the situation and react appropriately. The description is intentionally vague to accommodate the vast number of forms an agent may have.

Properties

Type: Agents are defined by setting the type property to `agent`.

Subtype: A property where the `agent` is categorized in more detail. Identifying such categories has not been a priority for this thesis and is considered future work.

6.1.9 Phase

A `phase` represents a subset of a scenario with a particular focus. As scenarios transition from one phase to another, many aspects of the scenario are subject to change. This ensures that all participants has the same premise at the beginning of a phase and that they are not affected by errors made in previous phases.

Properties

Type: Phases are defined by setting the type property to `phase`.

Objectives: The `objectives` property is a list of objectives that indicate the purpose and goal of the particular phase. Type: list of objective objects.

Rules: Type: *list*. A list of rules that are phase specific.

Agents: Type: *list*. A list of agents that are phase specific

6.1.10 Objective

An `objective` represent a goal that must be reached for a particular task to be considered completed. Multiple objectives may exist for a scenario at different levels such as overall objectives for a scenario and objectives specific to a phase.

Text: type: *string*. The objective in a textual format.

6.1.11 Rule

A `rule` define the boundaries of acceptable behavior for participants in a scenario or CTF.

Properties

Type: Rules are defined by setting the type property to `rule`.

Text: Type: *list*. A rule in a textual format.

6.2 Concrete syntax

Concrete syntax is the notation that is used to create a scenario and it is what a user interacts with directly [1].

The DSL follows the YAML specification. Indentation is used to create a hierarchical structure, and data can be represented in lists, key/value pairs or as a combination of both [13].

The syntax to define an object is identical for all concepts. An object is structured as seen in listing 6.1.

```
identifier:
  type: concept-type
  properties:
    some_property: some_value
```

Listing 6.1: General structure of an object

An object is identified by its *identifier* and is used when making a reference to an object. This is examplified in listing 6.2, where the networks' subproperty *router* makes a reference to another

router object called management. The identifier may have any name permitted in YAML. The identifier will also become the name of the object in cases where an objects name is relevant such as hostnames for nodes.

Within an object, the keyword `type` is mandatory and must be present to signify which type of object it is. Permitted values in the `type` field are found in table 2. The exception to this is the scenario concept which may have either `attack-defense`, `jeopardy`, `redteam-blueteam` or `wargame` as a value to the `type` field.

The `properties` keyword separates all properties of a concept into its own substructure. It is optional to include this keyword in which case default values are used although some concepts may have mandatory properties in which case the properties keyword must also be included. The properties that are available to each concept can be found in section 6.1.

When zooming out from individual objects and looking at the language as a whole, it is structured as a forest with two trees. The `scenario` concept is the root of its own tree and the objects of other concepts are defined as branches under the *resources* root. This is illustrated in listing 6.2.

```
scenario:
  type: attack-defense
  properties:
    name: Test-CTF

resources:
  example_node:
    type: node
    properties:
      public_ip: yes
      networks:
      - router: management
        subnet: attack_defense_subnet
```

Listing 6.2: Overall structure of a scenario

7 Technical design and implementation

This chapter starts by explaining details of the compilation process in section 7.1, before moving on to detailing the architectural design of the compiler in section 7.2. Section 7.3 explains the provisioning pipeline in detail and the systems involved during that process. In section 7.4, many of the design decisions that were made to construct the scenario are explained.

7.1 Compilation process

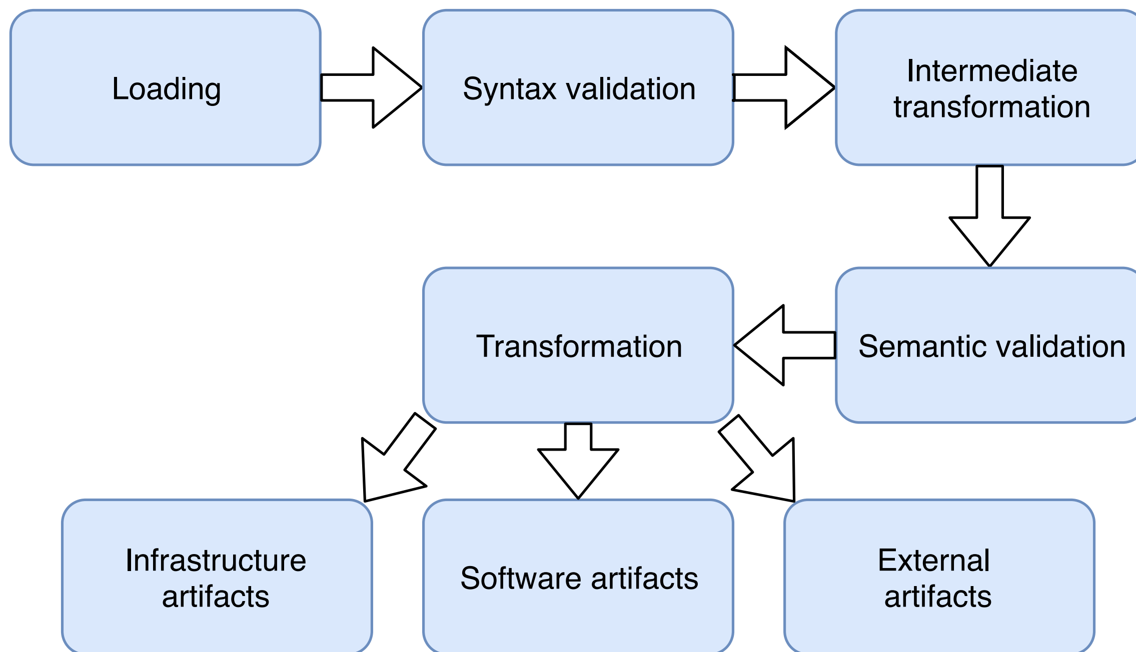


Figure 3: Compilation process broken down into major steps

Loading

The compilation process is a multistep process as seen in figure 3. Assuming that the user has created a scenario using the DSL and saved it to a file, the process starts with running the compiler and specifying the file as input. Then an instance of the DSL is loaded by the Python YAML library `oyaml`. Oyaml is a drop-in replacement for the more commonly used PyYAML library except that it has the benefit of preserving dictionary ordering when loading or dumping YAML.

Syntax validation

Upon loading the file, the YAML library will also do syntax validation to ensure that the input is correct according to the YAML specification. If the syntax is incorrect, validation fails and the compilation process exits.

Intermediate transformation

When syntax validation is passed, the elements of the DSL is transformed into a Python dictionary. This data structure is very flexible and provides an easy way to access data and apply logic on it. YAML's ability to transform into native data structures along with its ease of use is largely why YAML was chosen.

Semantic validation

It is possible that the input is syntactically correct but semantically incorrect. Multiple methods are used to verify the semantic correctness of the input:

- Verify that the input follows the structure presented in section 6.2.
- Verify the presence of mandatory properties.
- Check for presence of optional properties and make appropriate adjustments if they are not.
- Verify that the scenario only contains object types that are valid for that type of scenario.
- Verify that the selected challenge/service/vulnerability is supported and exists in the database.
- Verify that IP addresses are correct and in the correct subnet.

Transformation

When semantic validation is complete, the final step of the compilation process is to transform the internal data structure into artifacts that can be used by the lower level technology. Figure 3 shows that the data is transformed into three different types of artifacts. The infrastructure artifacts constitute the elements in a Heat template. This is used to build the virtual machines and networking components in OpenStack.

All artifacts that are related to installation and configuration of software and the operating system itself are transformed into Ansible templates. Each group of virtual machines will have its own template that defines the state they will have in the scenario.

Finally, there are the external artifacts. These are artifacts that do not fit into the infrastructure or software category. Many of the concepts listed in chapter 6.1 are abstract and do not fit into the first two categories. The `objective` and `rule` concepts are purely informational and are examples of artifacts that need to be presented to the end user in order to be useful. This also applies to file-based challenges that the user downloads to their personal computer. In these cases, all that is needed is a textual description of it and a URL to download the challenge from.

The proposed way to create the external artifacts is to create a data structure that can be used as input into a web application that displays the information to the end users. Developing the web application is a separate project which is running in parallel with this thesis. Therefore, defining how this data structure should look like and integrating these two projects has not been a priority and is considered future work.

7.1.1 Compilation output

The compilation process generates several files for various purposes. The below list details the purpose of each file.

- `ansible_deploy_key`: This is an SSH private key. It is transferred to the manager node to enable it to use SSH to connect to other nodes. It is necessary for ansible to work.
- `ansible_deploy_key.pub`: This is an SSH public key that is transferred to all generated nodes. It is necessary for ansible to work.
- `Heat_stack_<date>_<id>.yaml`: This is the generated Heat template that is used to create the infrastructure.
- `requirements.yaml`: The manager node needs to install all Ansible roles in order to apply the configuration they yield. This file is a list of Ansible roles that is necessary to achieve the desired state of the scenario.
- Ansible templates. The name of the file(s) depends on the identifier used in the DSL. The contents of the file define the software configuration. One Ansible template is generated for each node group.

Upon compilation, the files are added to a directory called *output* that must be transferred to the manager node. If the compiler is set to initiate the provisioning process, the files will be transferred automatically.

7.2 Compiler design

Architecturally the compiler is comprised of five classes as seen in figure 4. The scenario class object is the center and is responsible for instantiating `node, router` and `challenge` objects. A component can be said to be any part of a node that is installed and/or configured using an Ansible role. In practice, the component class implements the user_account, service and vulnerability concepts.

Notice that challenge objects are not created through the node class. This is because challenges are limited to jeopardy and wargame style scenarios where predefined Heat templates are used. This means that there are no node objects in these types of scenarios.

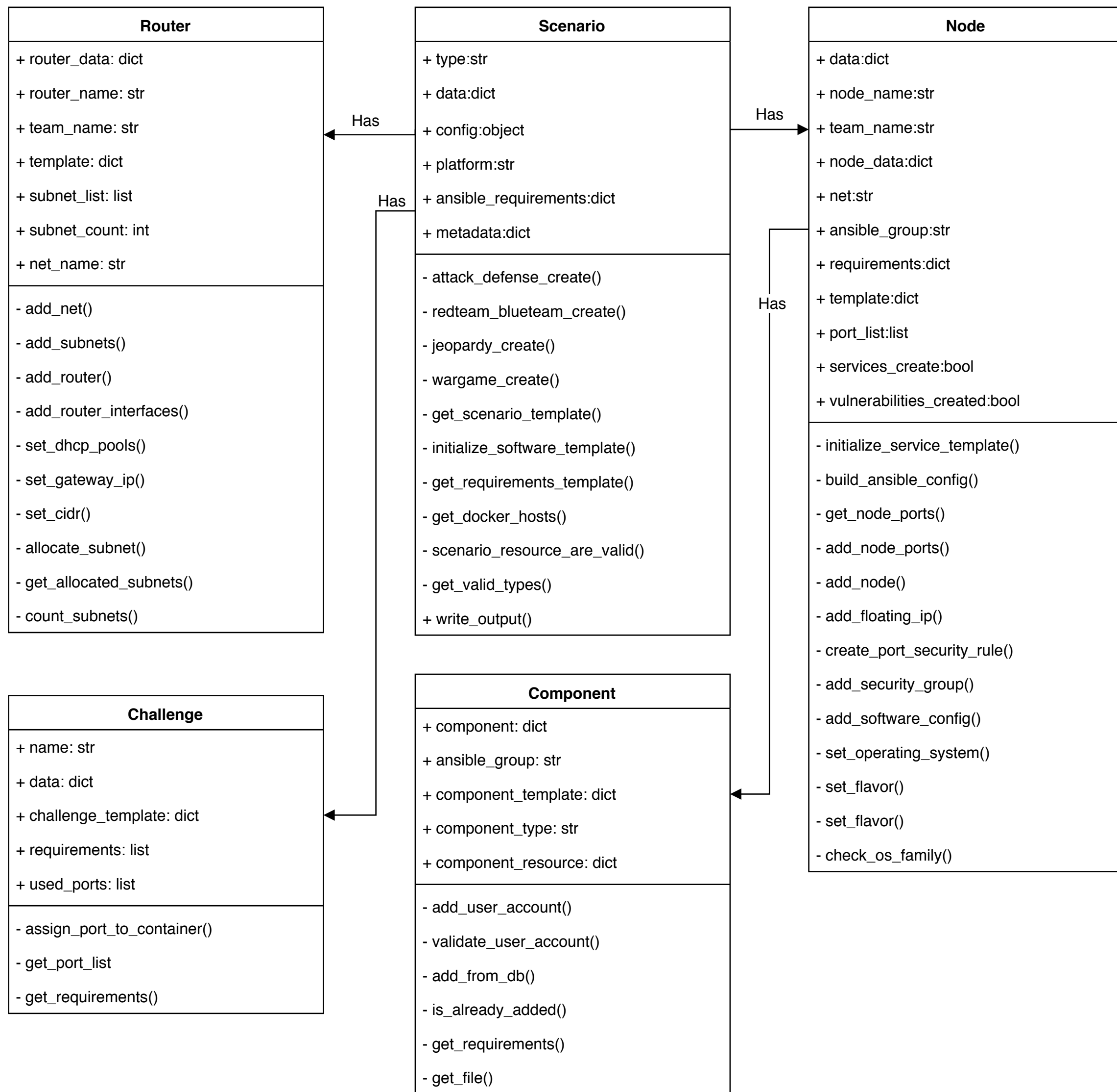


Figure 4: Class diagram of the compiler

7.3 Provisioning pipeline

In addition to being a compiler, the application also has the capability to initiate the provisioning process. This makes the compilation and provisioning a one-click process. The sequence diagram in figure 5 shows how the components interacts with each other during the provisioning process.

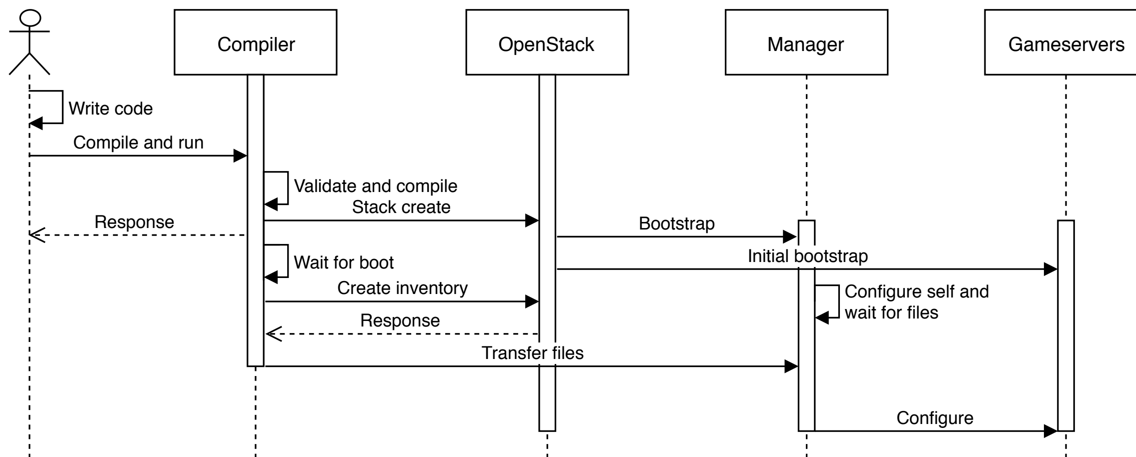


Figure 5: Sequence diagram displaying the process of provisioning a scenario

The user starts by writing the code that defines the scenario. The code is then compiled and validated by the compiler. If the code compiles successfully a request is sent to OpenStacks orchestration API and a Heat stack is created based on the generated Heat templates. OpenStack has the option to pass configuration scripts to the VMs using Cloud-init. This functionality is used to initiate basic bootstrapping of the VMs such as transferring SSH keys and installing Ansible. In an ideal situation, Cloud-init could have been used to completely configure the VMs, however this is not possible because Cloud-init limits the size of the file transfer to a maximum of 10240 bytes which is likely to be exceeded.

After the Heat stack is created and the nodes are booted up, the Ansible templates must be transferred to the manager. However for this to happen the Ansible inventory must be created first. The nodes acquire IP addresses from DHCP upon boot so the compiler does not have the IP addresses of the nodes on compile time. Therefore, the compiler makes a query to OpenStack requesting the IP addresses of all the nodes that are created. Notice that after the stack is created, a waiting period is added. This is to ensure that all nodes have obtained an IP address and have booted up before making the request. The compiler then populates the inventory with the IP addresses and transfers it to the manager along with all the generated Ansible templates. To transfer the files, the SSH protocol is used.

After the manager node's bootstrap, it enters a waiting state, awaiting the Ansible files to be received from the compiler. When the files are received, the manager runs Ansible which starts the process of configuring all the other nodes in the scenario into their defined state.

Since Ansible is a push-based configuration management utility, configuration must come from somewhere. Implicitly, this means that it must be a machine located inside the created environment because it is a possibility that not all nodes within the environment are reachable from the outside.

The *manager* nodes job is to be the link between the local machine used by the user and the machines in the created environment.

7.4 Low level design decisions

The focus of this thesis has primarily been to implement the concepts that are related to the infrastructure and the software configuration. These are concrete concepts that translate to either Heat templates or Ansible templates.

This section will explain some of the design decisions that were made in order to have a functional environment.

7.4.1 Infrastructure components

`Nodes` and `routers` are the concepts that constitute the infrastructure components that translate into Heat templates.

Wargame and jeopardy

The only concept which touches the infrastructure in wargame and jeopardy CTFs are `challenges`. Each challenge is run as a docker container on infrastructure created from a predefined Heat template. This is a design decision that was made due to several reasons:

- Network architecture is not important for these types of CTFs. A predefined architecture that will fit most situations can be used without disadvantages.
- Reduced complexity for the scenario designer. The user only needs to be concerned with which challenges to host. Network architecture and host configuration is completely transparent and hidden from the users point of view.
- Easy to implement for the developer.

Figure 6 shows how the network architecture is structured. To allow competitions to scale up, all nodes are identical and the user can specify the number of instances to create. Due to time constraints, there was not enough time to implement the OpenStack load balancing feature. Implementing this is considered future work.

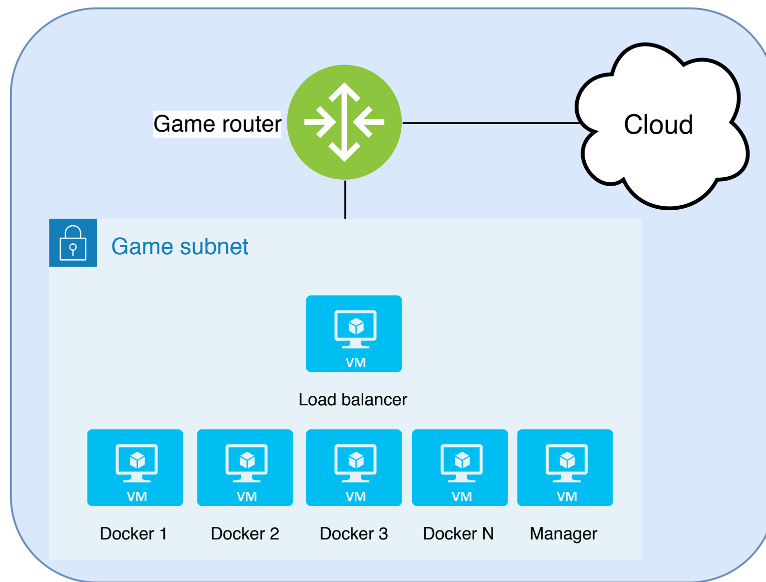


Figure 6: Network architecture for jeopardy and wargame CTFs

Attack & Defense

In A&D scenarios, the scenario designer can freely define the nodes and their configuration but not the network architecture. The network is statically predefined and the node must use this network. Figure 7 shows the network architecture for A&D scenarios. The game subnet consists of all the nodes belonging to participants and a VPN node. In section 2.2.1 we mention an architectural problem in attack & defense competitions that must be taken into consideration. The way this is solved in this thesis is to have a VPN server in the game subnet that the gameserver and each team will connect to. This way, it will look like all traffic originates from the VPN server, obfuscating the original sender. An added benefit is that the participants can be allowed root access to the node hosting the game application.

The intention is to use Wireguard as the VPN service to due its relative simplicity compared to other VPN solutions. Wireguard encrypts the connection and requires that each node is associated with a keypair. Each node must have the VPN server's public key and the VPN server must have the public key of each node. This presents us with a key distribution problem that we have not had the time to automate due to time constraints.

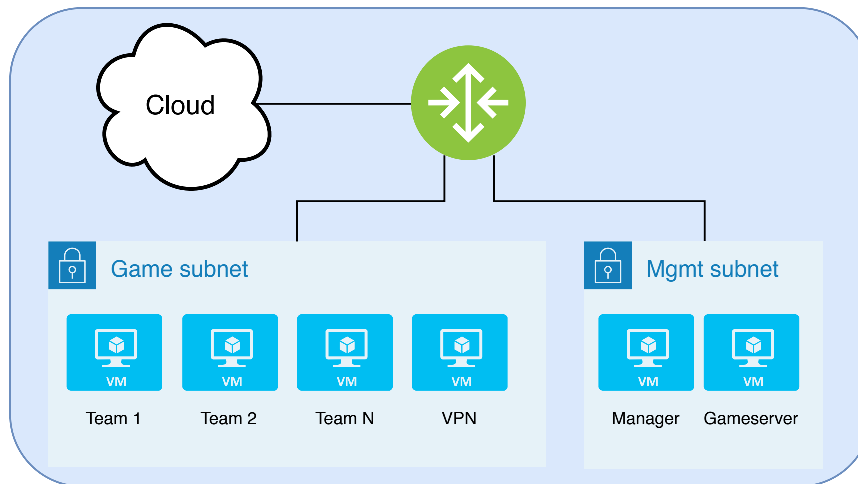


Figure 7: Network architecture for Attack & Defense competitions

Red team/Blue team scenarios

In R&B scenarios the network architecture in addition to all nodes and their configuration, can be defined. Figure 8 shows an example environment for a single set of red and blue teams in a R&B scenario where the scenario consists of three subnets with two VMs each. A manager node must always be present as it provides the functionality necessary to configure the other nodes. Its placement differs depending on the type scenario as seen in figure 6, 7 and 8.

Both A&D and R&B scenarios are defined for one team but can be replicated to the number of teams participating in the scenario. This ensures that all teams get identical setups and it minimizes the amount of code that must be written to create the scenario.

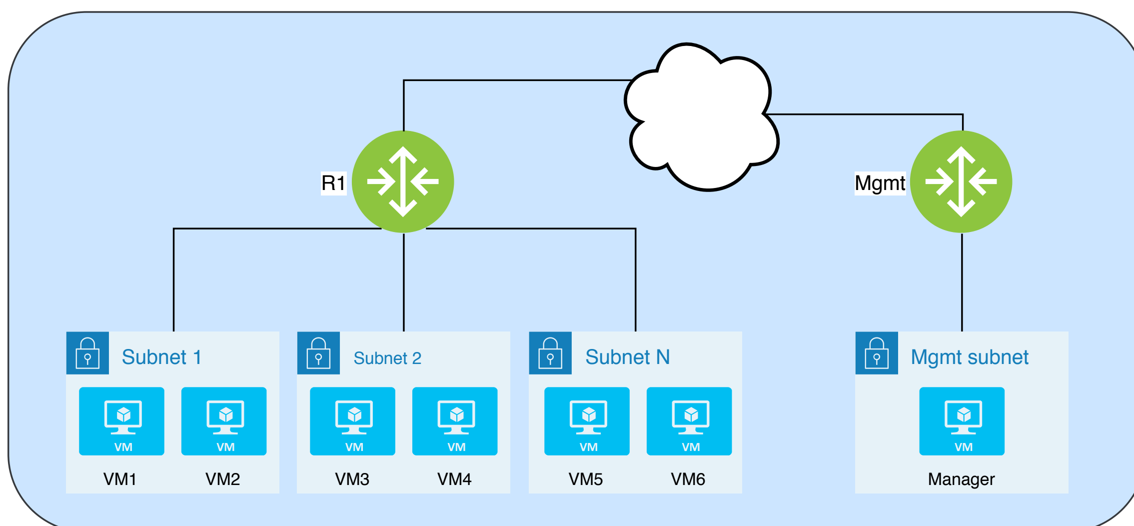


Figure 8: Network architecture for a single red team/blue team environment

R&B scenarios require that the network architecture is structured in such a way that it can be generalized to a wide variety of architectures. Due to the way OpenStack works, creating arbitrary network topologies is not entirely trivial for two reasons.

1. OpenStack's router abstraction can not do dynamic routing. This means that the manager in the mgmt subnet of figure 8 can not communicate with any of the VM's in R1's subnets and vice versa without first configuring static routes to each others subnets.
   In a complex environment, this complicates configuration significantly. To configure a static route, the destination subnet and a next-hop address is required. As these addresses are allocated during stack creation, these addresses are not known in advance. This could potentially be solved using Heat's built-in intrinsic functions although due to time constraints, this has not been explored in more detail.
2. Connecting a router to OpenStacks external networks is easily configured with built-in properties, whereas connecting two routers together is a complex operation that requires several components of the router to be configured. This makes configuring networks where routers are connected to each other difficult.

The result of this is that there are limitations to the network architectures that are reasonably possible to create. There is currently no easy way to directly connect two routers together and attempting to do so appears to be unnecessarily cumbersome.

7.4.2 Software components

Concepts that run as software on top of virtual machines utilize Ansible to apply the configuration. Currently the concepts that are configurable on a node are `user_accounts`, `services` and

`vulnerabilities`. `Challenges` are also implemented but are run as a standalone entity rather than as part of a node.

Vulnerabilities, services and user_accounts

Implementation-wise, the only difference between a service and a vulnerability is that a vulnerability is intentionally vulnerable whereas a service is not. Each vulnerability/service maps directly to an Ansible role. A database is used to keep track of the supported software. A unique identifier for each supported vulnerability, service or challenge is added to a database which maps this identifier to a URL and the underlying role used to install the particular software. The advantage of this approach is that the compiler can create the components without being dependent on the underlying role.

Roles typically have variables whose default values can be overridden. These variables can be defined directly in the DSL by specifying a new object with the same name as the identifier. Listing 7.1 illustrates an example of where a node is configured with apache using the default settings of that role. In listing 7.2 is the same example is used except that the apache service is configured to override the default value. Instead of using the default port, it uses port 8000.

In both listings, unrelated properties are removed for brevity.

```
resources:
  example_node:
    type: node
    properties:
      services:
        - apache
      ...
```

Listing 7.1: Node running apache with default settings

```
resources:
  example_node:
  type: node
  properties:
    services:
      - apache
    ...

  apache:
    type: service
    properties:
      apache_listen_port: 8000
```

Listing 7.2: Node running apache where port is changed to 8000

The implementation of `user_accounts` is similar to the implementation of services and vulnerabilities. An Ansible role is selected to implement this but the role remains replaceable since the input from the DSL is passed directly to the underlying role.

The main advantage of this approach is that we can use the role's variables directly inline in the DSL without doing any translation. This makes it very easy to add support for new roles because development effort is minimized. The disadvantage is that backwards compatibility might break if at some point the role is swapped out for a similar role that either lacks the same features or uses different variable names.

Challenge

The implementation of challenges are limited to those running as docker containers. Container-based challenges are implemented using a simple Ansible role that was developed for this purpose [22]. Listing 7.3 shows an example of how a scenario consisting of two challenges are defined using the DSL.

```
resources:
  example1:
    type: challenge

  example2:
    type: challenge
    properties:
      port: 1337
```

Listing 7.3: Example definition of two challenges

The challenge identifier is looked up in a database to verify its existence. The database also contains a mapping between the identifier and a data structure containing three properties; the name of the image, the port being exposed by the image and a docker registry path to the image. With this approach, images are built locally and added to the docker registry. This shortens the deployment time since the images are already built.

8 Case study

8.1 Overview

Rationale

Hosting cyber security scenarios and competitions require high levels of technical knowledge, time and effort. The domain specific language and the proof-of-concept compiler created during this thesis is a contribution to the community in an attempt to make hosting such events easier.

Objective

The goal of the case study is to answer the research questions defined in section 1.5 so they are the basis for the work in this thesis.

Q1. Can we reduce the time, resources and technical knowledge needed to create cyber security testbeds by modelling them in a domain specific language?
Q1.1. Can we develop a domain specific language that represent the knowledge needed to automatically create a cyber security testbed?
Q1.2. Can we create a proof-of-concept compiler that automatically provisions the testbed based on the developed DSL?
Q2. Can this system be used to reduce the knowledge gap between the scenario designer and the infrastructure developer?

The case study is also well suited to identify strengths and weaknesses with the system. Future contributors can use this information to make an informed decision on which parts of the system to improve. Each case is selected because it targets a type of scenario that is supported by our application.

Design

The cases are designed as holistic case studies. This means that the focus is on the broader perspective of the application rather than on small details. The duration of each case is also very short, so no case study protocol will be maintained during the execution of the cases.

For each case the plan is to first setup the environment by following original documentation for the respective case. The documentations can be provided upon request. Common for all of them is that they follow a semi-automated setup process primarily based on Heat for infrastructure orchestration and Puppet for configuration management. Domain experts that were involved in the original projects will oversee the replication of the cases to ensure that they run as smoothly as possible.

8.1.1 Data collection

The evaluation of performance will rely on data from primarily two sources, metrics and interviews.

Quality metrics

Four metrics have been identified as relevant key performance indicators for the system. These metrics are relevant to all case studies conducted in this thesis. These metrics also reflect the data that is collected.

Efficiency: An indicator of how much time and manual labor is required to create a scenario. Time and the level of automation are data points that will be collected. The source of this data is expected to be both observations of the system and feedback from the interview subject.

Usability: An indicator of how easy to use the system is for the user who creates the scenario. To review this metric, an interview will be conducted with the personnel responsible for managing the infrastructure of the case studies. They will be allowed to try the system and their feedback will be used to determine the usability. The source of this data is expected to come primarily from the interview subject.

Completeness: A measure of the extent to which the replica system is capable of mirroring the original system in terms of features. The gold standard of completeness is considered to be a production ready environment that contains all the bells and whistles for that particular type environment. In order to measure completeness, we will collect data during execution of the case about how the replication went. The source of this data is expected to be primarily observations of the system although the interview subject will provide feedback.

Flexibility: An indicator of how easy it is modify existing or add new content to the scenario after it has been created. Post-deployment modification to the scenario has not been a focus of this thesis as the main goal has been to create an application that can create the initial state of the scenario. Data pertaining to flexibility is mainly collected to identify the current state so that it can be improved upon in later research.

Expert opinion

Two domain experts will be used to assess the old and new system to provide feedback. *Expert 1* was on the team hosting NCSC and has hands-on experience with the tools and methods used to set up NCSC using the original method. *Expert 2* has experience with OpenStack, Heat and Puppet and domain knowledge in organizing cyber security testbeds although lacks familiarity with some of the specific tools that were used to deploy NCSC.

The interviews will be conducted in a semi-structured manner with a mix of closed and open-ended questions. The questions that will be asked during the interview are seen in appendix A.

8.1.2 Analysis procedures

Analysis will follow the steps outlined in Runeson et al. [21]. When data is collected it will be labeled with a code. Each quality metric will have its own code where the data provides informa-

tion that either positively or negatively impact the particular quality metric. The findings will be used as a basis to form a hypothesis about the performance of CTFgen before assessing whether generalizations can be made.

In order to increase validity of the findings, a triangulation method will be used where conclusions are drawn based on information from multiple data sources [21].

8.2 Case study 1 - NCSC 2019 Jeopardy

The *Norwegian Cyber Security Challenge 2019* is a capture the flag competition hosted by NTNU [23]. NCSC is a competition consisting of two rounds where participants must qualify in the first round to be eligible for participation in the second round. NCSC is also a selection process to find members of the national team that will compete in the European Cyber Security Challenge.

The first qualifying round is a jeopardy style CTF featuring challenges in domains such as hardware, cryptography, forensics, reverse engineering, IoT and more. The second round is an attack & defense competition.

This goal of this case is to first recreate the infrastructure and configuration that was used during the jeopardy part of NCSC using the same method as when it originally ran. Next, we will use the new system that is developed to create an environment that resembles the original environment.

NCSC stands out as a good candidate to assess the performance of the system for two reasons. Firstly, because it is a jeopardy style CTF which is one of the scenarios that the work of this thesis attempts to make easier. Secondly, access to both the documentation and the people originally involved with it is good because it was hosted at NTNU.

8.2.1 Replicating NCSC part 1 using the original method

Figure 9 shows the infrastructure of NCSC 2019. All nodes in the management subnet have various roles. The sensu, graphite and grafana nodes exist for monitoring purposes. The jump host exist because it's the only node with a public IP address. All other nodes have private IP addresses so in order to access the other nodes, one must go via the jump host.

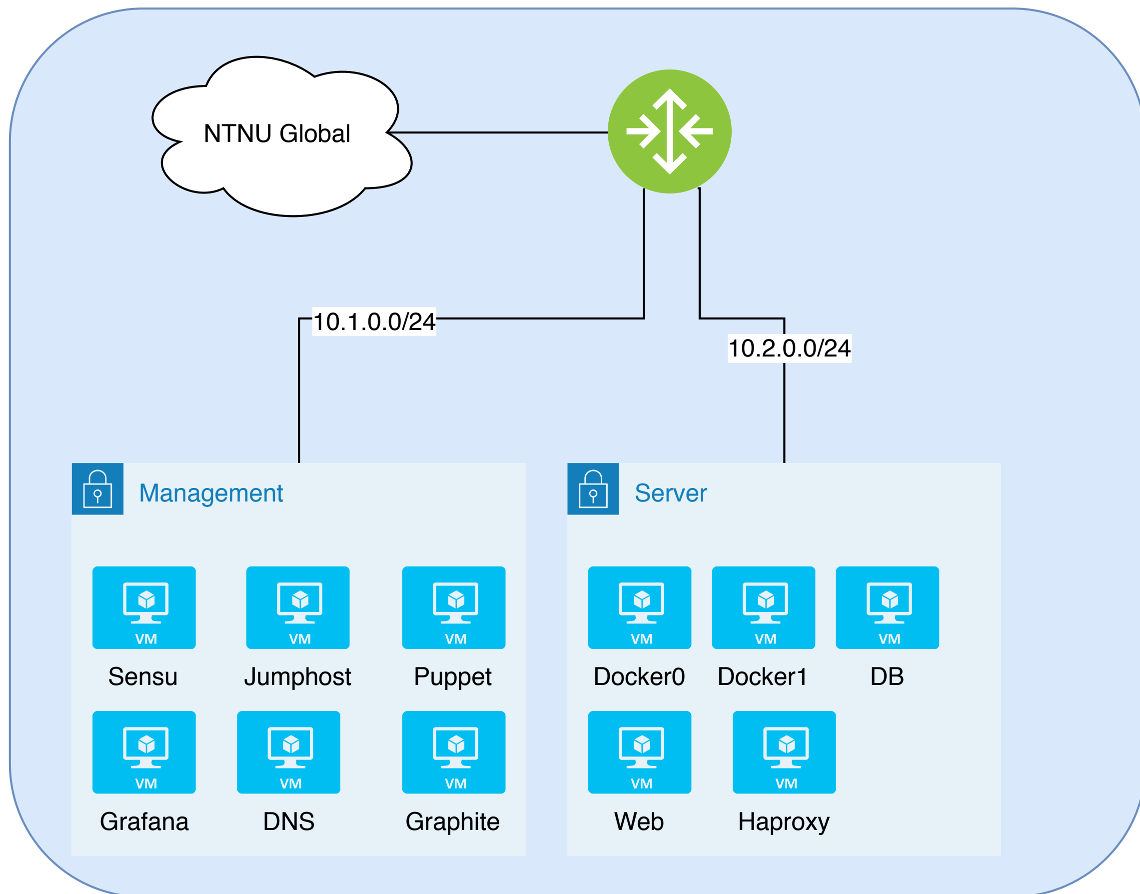


Figure 9: Network architecture for the jeopardy part of NCSC 2019

Deploying the environment is a semi-automated process consisting of multiple steps where each step requires input from the user. Large parts of the process have been scripted and can broken be down into steps as seen in the list seen below. Each step is either fully or partially handled by a shell script.

1. Deploy the Heat stack by running the command
   `openstack stack create -t ctf.yaml -e ctf_env.yaml ctf`
2. The Puppet node needs to access a private git repository in order to apply its configuration catalog. The user must manually go into the repository and add the generated public key to its list of authorized keys. Once the key is added, Puppet deploys the catalog for the environment.
3. Puppet requires that the Puppet master signs all the client keys. A script has been written that handles this although for security reasons, the user is required to verify each client.
4. After the keys has been signed, Puppet must be initiated manually on each node. A script has

been written to do this so accomplishing it is easy.

5. Puppet unfortunately does not setup everything. NCSC used the YACTFF [24] web framework as its frontend. Manual configuration on the web node is needed to get YACTFF working. This step is unnecessarily complicated due to the fact that internal nodes do not have a floating IP, so any time the user needs to access a web interface on the management infrastructure, access must happen by using the jump host as a proxy.
6. A script has been written to handle the challenge creation. This script will build the docker image, compress it and subsequently transfer it to the Puppet node.
7. In order to launch the Docker containers for use by participants, they must be enabled in a configuration file on the Puppet node. Therefore we had to log into and edit the configuration file to include the desired container.

Steps 6 and 7 must be repeated for every container that is to be included in the scenario. Setup of the environment takes around 20-25 minutes for an individual that is familiar with the process and the tools used to set it up.

8.2.2 Replicating NCSC part 1 using CTFgen

In order to deploy a container-based challenge using CTFgen, the container must exist in a docker registry. As the registry that is intended to store these challenges is the work of a parallel project (PLED), integration towards the PLED database is not ready yet.

Because of this, for the purposes of this case study, we will use example containers hosted on Docker Hub as replacements for the challenges that were originally in NCSC 2019. This should not affect validity as the goal is to study the process of deploying a scenario similar to NCSC rather than the challenges themselves.

We start by defining a list of challenges that we want to run as seen in listing 8.1. Due to the lack of a centralized database, we need to add support for the challenges to the local data structure in data.py [25] that acts as a placeholder until integration with PLED is added. Next we compile the DSL and execute it to start the provisioning process. It takes about five minutes for all challenges to come online.

```
scenario:
  type: jeopardy
  properties:
    name: NCSC 2019
    docker_hosts: 2

resources:
  crypto-01:
    type: challenge
  crypto-02:
    type: challenge
```

```
crypto-03:
  type: challenge
crypto-04:
  type: challenge
crypto-05:
  type: challenge
crypto-06:
  type: challenge
crypto-07:
  type: challenge
crypto-08:
  type: challenge

reverse-engineering-01:
  type: challenge
reverse-engineering-02:
  type: challenge
reverse-engineering-03:
  type: challenge
reverse-engineering-04:
  type: challenge

forensics-01:
  type: challenge
forensics-02:
  type: challenge
forensics-03:
  type: challenge
forensics-04:
  type: challenge

web-01:
  type: challenge
web-02:
  type: challenge

pwnable-01:
  type: challenge
pwnable-02:
  type: challenge
pwnable-03:
  type: challenge
pwnable-04:
  type: challenge
```

Listing 8.1: Jeopardy part of NCSC 2019 defined in the DSL

8.2.3 Analysis

In this section we will use the quality metrics defined in section 8.1.1 to analyze the results from the case study.

Efficiency

Setting up the environment as per the documentation takes around 20 minutes assuming that the user knows what to do. Several shell scripts have been written to automate most of the process, but there are still occasions where the user is required to interact with the system.

Environment setup using CTFgen can be broken down into two steps. The first step is submitting the challenges to the database and the second step is running CTFgen. The whole process takes about 5 minutes.

There are several factors that are likely contributors to making CTFgen faster. Firstly, some time is wasted in between each step during the setup process of NCSC because user input is required. Secondly, the NCSC environment contains components that are not present in the environment set up with CTFgen. CTFgen's management infrastructure is currently limited to the Ansible manager node. Sensu, Grafana, Graphite, DNS, Web and DB are services that, for the purposes of this project, are considered "nice to have, but not strictly necessary" and thus implementing them has not been prioritized. CTFgen's environment also currently does not have a web interface as this is the work of another project. The fact that there are more servers to configure, is likely to cause Puppet to appear slower. Puppet, which is used for the configuration in NCSC, must configure 11 servers while Ansible only needs to configure 3 servers.

As far as the level of automation goes, CTFgen's process is more streamlined although we must take into account that the original NCSC environment has components not present in CTFgens environment. As CTFgen's environment does not have a web interface, we can exclude step 5 from the list.

Usability

Setting up a jeopardy scenario using CTFgen provides significant improvements to usability when compared to methods used previously. The semi-automated process requires that the scenario designer has working knowledge with Heat and Puppet. They also need to understand the limitations of this approach and know how to use various tools to circumvent these limitations.

Creating a jeopardy scenario through CTFgen simply requires that the scenario designer use the DSL to specify which challenges the scenario should consist of. A prerequisite for this, is that the challenge is present in the database where challenges are fetched from. In cases where it is not, an entry pointing to the challenge must be added to the database.

The case was conducted twice, one time for each expert. To set up NCSC using the old method, Expert 1 needed about 20 minutes to get everything up and running, as he was familiar with the existing tools and methods. Expert 2 had no prior knowledge of the setup process and had to rely on the documentation to a larger extent. All in all, it took expert 2 about two hours to set up NCSC.

When presented with the DSL and CTFgen, both experts quickly understood the structure of the language and were able to define the challenges to be included within a few minutes. Both experts had seen the presentation on the language prior to the case, but had not used in themselves. They were also able to deploy the scenario with little effort.

When comparing CTFgen to the old provisioning method, both experts support the authors opinion that usability is the metric where the performance increase is the highest. The main reason being the greatly reduced time it took to get an understanding of how to get the environment set up to the desired state. Furthermore, feedback from the experts as well as observations support the second research hypothesis.

The reason for the high increase in usability is likely due to the reduction in abstractions the user needs to take into account. Many of the underlying abstractions are easily hidden due the simplicity and static nature of the infrastructure used in jeopardy scenarios. This makes implementation easy because templates can be reused with very little modifications from scenario to scenario.

Completeness

Completeness is still somewhat lacking as jeopardy scenarios are currently limited to container based challenges. The most pressing issue is the lack of an output data structure containing information about concepts not related to infrastructure or software. While this is supported in the DSL itself, it has not been implemented yet. This data structure is necessary in order to present relevant information to the user. It should be noted that there are redeeming factors to this. Both the web interface where this data would be presented and the database where data is fetched from, are parallel and independent projects conducted by other students. Close collaboration is necessary to create a platform that integrates the different components successfully. This is out of scope for this thesis and should be focused on in later research.

Another factor that inhibits completeness is the lack of feedback after a scenario has been created. Monitoring should be in place to track the health status of the environment. Other features that should eventually be implemented include DNS and load balancing.

Despite the limitations, jeopardy scenarios created with CTFgen are functional albeit in a very bare bone state. The goal derived from research question 1.2 was to create a proof of concept implementation that can provision a scenario based on an input model. For jeopardy scenarios, the case study has shown that this goal is fulfilled.

Flexibility

Post-deployment modifications to the scenario environment was not part of the scope for this thesis, so it is currently not possible to modify an existing scenario through CTFgen. This has an impact on usability as any update to the infrastructure or software must be done manually outside of CTFgen.

8.3 Case study 2 - Attack & Defense

The second and final round of the *Norwegian Cyber Security Challenge 2019* is an attack & defense competition. This round is used to select participants that will represent Norway in the *European Cyber Security Challenge*.

An interesting note is that the competition did not go as planned. Unfortunate circumstances led to a situation where the people behind the technical implementation of the competition could not be present during the competition. When this is combined with lack of documentation, the result was an operations team that could not set up the environment because they were unfamiliar with the software stack.

Due to the lack of documentation, this case will not attempt to recreate the exact environment planned to be deployed. Given that the available domain experts were unable to finish a deployment in time for the competition we assume the task to be too time consuming. Instead, we use CTFgen to model an attack & defense scenario which is similar in functionality and infrastructure to the one deployed during the previous year. Since it was planned to use the same framework during the competition this year we deem them be equivalent for our purposes.

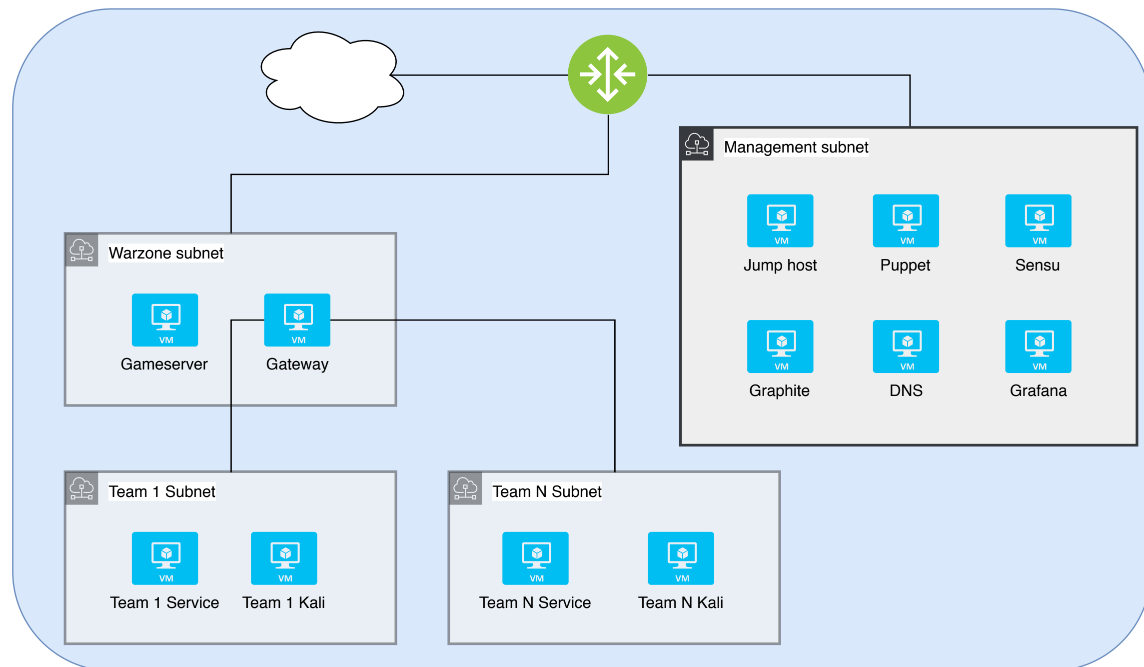


Figure 10: NCSC 2019 Attack & Defense network architecture

Figure 10 shows the network architecture of NCSC 2019 A&D. Each team has two nodes located in a dedicated subnet; a node for hosting the vulnerable service and a node running Kali linux for

attacking other teams' service node. The team subnet connects to a gateway node whose job is to obfuscate the origin of incoming connections.

Section 2.2.1 mentions various solutions to the issue of teams being able to filter network traffic to deny opposing teams access to their application if the players have root access to the node hosting the vulnerable application. In NCSC 2019, this is handled by setting up a reverse NAT host that packets are routed through to obfuscate the origin. This task is handled by the gateway node seen in figure 10.

The gameservers' task is to coordinate the progress of the competition by placing new and revoking old flags at certain intervals. It also checks that each team's service is available and working as intended. The gameserver is also where participants submit the flags they capture.

8.3.1 Creating an attack & defense environment with CTFgen

Listing 8.2 shows the definition that was used to create an attack & defense scenario that resembles the original NCSC. It contains two nodes for each team; an attacker node and a service node. The VPN and gameserver node are created behind the scenes.

A requirement for configuring nodes with services or vulnerabilities in CTFgen is that the service/vulnerability is associated with an Ansible role. A single application was created for use in NCSC, but no Ansible role was created for it so we were unable to use it with CTFgen. Instead, apache was used as a dummy service to showcase that service configuration through CTFgen works. Note that the only difference between services and vulnerabilities from an implementation perspective is the database they are fetched from.

In CTFgen, all nodes which have a management purpose are statically defined and their configuration is hidden from the user. The VPN node provides a VPN service used as a jump host and to obfuscate the source address. We did not finish it's implementation due to time constraints, as discussed previously in section 7.4.1.

The gameserver is supposed to run an NTNU's forked version of FaustCTF [26], although an Ansible role is require in order to install it using CTFgen. It was therefore not possible to configure the gameserver to do anything meaningful.

Table 3 summarizes the status of the nodes in the environment that was created.

Attack & defense	
Node	**State**
Attacker node	Adequate
Service node	Adequate, but lack of A&D challenges is a problem
VPN	Inadequate, minor issues must be resolved first
Gameserver	Inadequate, missing appropriate Ansible roles

Table 3: Status of components in the attack & defense environment created with CTFgen

```
scenario:
  type: attack-defense
  properties:
    name: NCSC 2019 AD

resources:
  service_node:
    type: node
    properties:
      networks:
      - router: management
        subnet: attack_defense_subnet
        port_security:
        tcp:
        - 80
        - 443
      public_ip: true
      services:
      - apache2

  attack_node:
    type: node
    properties:
      networks:
      - router: management
        subnet: attack_defense_subnet
      public_ip: true

  Team1:
    type: team

  TeamN:
    type: team
```

Listing 8.2: DSL definition of AD scenario that resembles the NCSC

8.3.2 Analysis

In this section we will use the quality metrics defined in section 8.1.1 to analyze the results from the case study.

Efficiency

The first research hypothesis is that we can automate the provisioning process to reduce time and resources needed to create a scenario.

Currently, manual intervention on the VPN and gameserver nodes are needed to have working scenario. The gameserver's configuration is out of our control as it uses an external framework and an Ansible role must be written to automate the installation.

Despite some missing features, the provisioning process have been proved to work well, especially when Ansible roles for the desired software already exist. The improvement to efficiency indicates that research question 1.2 is fulfilled.

Usability

It is encouraging to see that usability of CTFgen shows improvements over more low level provisioning methods. The node and service/vulnerability concepts appear to work well. The improvements allow scenario designers to focus on the scenario's content rather than on how to set up the the management components that an A&D environment consists of. This supports the hypothesis derived from research question 2 which is concerned with reducing the knowledge required to create cyber security training environments.

Completeness

The attack & defense scenario needs manual work before it can be used because the VPN and gameserver nodes provide essential features that are currently missing. Providing VPN functionality was within the scope of the thesis but time constraints prevented us from finishing the implementation. Fully configuring the gameserver is outside the scope of this thesis because it requires a CTF framework for flag management. This framework is developed by a third party and there is currently no Ansible role to automate its installation.

Currently, a downside with the platform that impacts all the quality metrics is the lack of software that is supported by the platform. It impacts efficiency because work that would otherwise be automated must be done manually, which in turn has a negative impact on usability. The lack of software will make the environments less complete. The compiler is designed with this problem in mind. The process of adding support for services/vulnerabilities is easy provided that an Ansible role for the desired application exists. The compiler only needs to know where to get the role from. The problem is especially prevalent for CTF specific frameworks and applications as it's a small niche and Ansible roles for them do not exist.

Flexibility

The situation in A&D scenarios are similar to what they are in jeopardy as described in the first case. Post-deployment modification to the scenario must be done manually. Since challenges in A&D are generally dependent on the CTF framework that is used, the extent to which it is possible to make

post-deployment modifications is largely dependent on the framework. The framework currently used by NTNU based on FAUSTCTF [26] does not provide adequate abstractions and support for this.

The *phase* concept introduced in the DSL is dependent on an environments capability of transitioning from one state to another. This feature will drastically improve usability and should therefore be focused on in later research.

8.4 Validity

Both cases have been conducted under circumstances that may impact validity. The author is responsible for both the case analysis and the development of the application. It is therefore a possibility that the author has various biases which impacts the results.

Another limiting factor is the small sample size of two domain experts in the first case and one in the second case. This may have an impact on the result's trustworthiness as nuances that would otherwise present themselves, can be due to the small sample size.

The application has not undergone any serious testing procedures. It is therefore possible that there are bugs or other issues that prevent the application from running as intended on other systems or platforms. The application has only been tested on NTNU's OpenStack platform. No investigation has been done to verify that this platform do not contain custom configuration that prevents the results from being generalized to any other OpenStack platform.

9 Discussion

The results obtained from the first case study where jeopardy style CTFs were observed are promising. Both the author's own observations and feedback from the domain experts suggest that the DSL and its automated provisioning pipeline provides significant improvements to usability and efficiency. Completeness is still lacking as there are several important features that are missing. The challenge concept must be expanded to include other types of challenges such as file-based challenges.

In the results we see that provisioning of a functional jeopardy scenario containing only container-based challenges is fully automated. The environment is created based on the DSL code in listing 8.1. This shows that the system has merit and it supports the first research hypothesis.

The second research hypothesis we state that this system will contribute to reduce the knowledge gap between the scenario designer and the infrastructure developer. The results show that there has been a reduction, mainly due to a reduced number of components a scenario creator must take into account but also increased abstraction and automation that hide low level details.

In the second case study, the goal was to use the DSL and the compiler to create an attack & defense scenario that is functional and as realistic as possible. The results shows that we were successful at deploying the infrastructure and at configuring the nodes that belong to the teams. In the A&D architecture, there are two management nodes that are essential to a fully functioning A&D environment. The VPN server that is required was only partially configured and the gameserver node was not configured at all. Key distribution for the VPN server is a requirement which have many viable solutions.

When we take into consideration that fixing the gameserver is out of scope, the results are very encouraging. We were able to use the DSL code in listing 8.2 to provision and partially configure an A&D scenario. Feedback and the authors own observations suggest that setting up an A&D scenario is easier when using the DSL than when directly using infrastructure provisioning and configuration management tools. This claim is supported by the components of the DSL being more concise and everything being located in the same place. These arguments support the research hypotheses for this thesis.

Red team/blue team exercises were not formally observed in a case study, although the author has formed some opinions about its state during development. The intention is that the user should be able to define the network architecture in addition to all the other concepts available in an A&D scenario. Due to OpenStacks lack of dynamic routing protocols and the difficulty of configuring router-to-router networks, creating a system on top of this seems complicated and counter intuitive.

It is currently not clear to us how to solve this problem. It may be necessary to redesign the DSL's networking abstraction around something other than the OpenStack router although this needs to be investigated in more detail. Because of this, the preliminary results show that for red team/blue team exercises, we are unable to improve upon the status quo in regards to research question 1 and 2 including the subquestions.

The DSL is also intended to support a wargame type CTFs. This scenario type has been largely ignored during this thesis due to it being less prevalent and quite similar to jeopardy CTFs. After a discussion with the supervisors, it was decided to support it in the DSL for possible future work but to focus on the other scenarios instead during this thesis.

It is not possible to test all the DSL's concepts in the current iteration. This means that many of the concepts have not been tested as thoroughly as the ones that have been implemented. A consequence of this is that they may have to be modified when development starts and they come under scrutiny. This applies specifically to the *rule, objective, team, phase* and *agent* concepts.

10 Conclusions

This thesis set out to reduce the knowledge, time and resource requirements to create a cyber security testbed. We did this by creating a DSL that is capable of representing various types of cyber security testbeds and create a proof-of-concept compiler that transforms the DSL into low level artifacts that can be used to automate provisioning of the testbed defined in the DSL.

The conclusion is that this goal is largely achieved for two out of the three types of testbeds we wanted to support. The jeopardy and A&D environments created with the DSL are still missing some important features although they are within the scope set for the thesis. For red team/blue team exercises we encountered some limitations with OpenStack's network abstractions ability to scale to complex network topologies. This prevented us from implementing this feature within the given time frame. Future work is needed to overcome these challenges.

10.1 Future work

The long term goals for this project extend far beyond what is possible to achieve during one semester. Therefore, the scope had to be narrowed down significantly. Several essential components for a fully functional production ready environment have been excluded from the scope. Below is a list of suggested areas that future work should focus on to progress towards a production ready environment:

- Integrate towards the web interface and database that is conducted in other projects.
- The networking abstraction for red team/blue team exercises must be investigated further and perhaps redesigned to overcome the issues discussed in section 7.4.1.
- External artifacts detailed in section 7.1 need to be modeled and implemented.
- Fix minor issues that we did not have time to fix during this thesis such as the VPN server for A&D and load balancing for jeopardy environments.

Bibliography

[1] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E., & Wachsmuth, G. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

[2] OpenStack Foundation. 2018. Heat openstack orchestration. URL: https://wiki.openstack.org/wiki/Heat.

[3] Puppet. 2019. Configuration management. URL: https://puppet.com/.

[4] Red Hat. 2018. Ansible documentation. URL: https://docs.ansible.com/ansible/latest/index.html.

[5] Hack The Box. Hack the box challenges. URL: https://www.hackthebox.eu/home/challenges/.

[6] Nasjonal Sikkerhetsmyndighet. i18 challenge. URL: challenge.i18.no.

[7] OverTheWire. Overthewire. URL: http://overthewire.org/wargames/.

[8] Smash the stack Wargaming Network. Smashthestack. URL: http://smashthestack.org/wargames.html.

[9] NATO CCDCOE. Locked shields 2013 after action report. Technical report, NATO CCDCOE, 2013. URL: https://www.ccdcoe.org/uploads/2018/10/LockedShields13_AAR.pdf.

[10] Miessler, D. 2019. The definition of a purple team. URL: https://danielmiessler.com/study/purple-team/.

[11] www.json.org. Introducing json. URL: https://www.json.org/.

[12] World Wide Web Consortium (W3C). Extensible markup language (xml). URL: https://www.w3.org/XML/.

[13] Oren Ben-Kiki, Clark Evans, I. d. N. 10 2009. Yaml ain't markup language (yaml™) version 1.2. URL: https://yaml.org/spec/1.2/spec.html#Basic.

[14] Ernits, M., Tammekänd, J., & Maennel, O. 2015. I-tee: A fully automated cyber defense competition for students. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, 113–114, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/2785956.2790033, doi:10.1145/2785956.2790033.

[15] Ernits, M., Tammekänd, J., & Maennel, O. I-tee github. URL: https://github.com/magavdraakon/i-tee.

[16] Schreuders, Z. C., Shaw, T., Shan-A-Khuda, M., Ravichandran, G., Keighley, J., & Ordean, M. 2017. Security scenario generator (secgen): A framework for generating randomly vulnerable rich-scenario vms for learning computer security and hosting CTF events. URL: https://www.usenix.org/conference/ase17/workshop-program/presentation/schreuders.

[17] Čeleda, P., Čegan, J., Vykopal, J., & Tovarňák, D. 2015. Kypo – a platform for cyber defence exercises. In *STO-MP-MSG-133: M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence*, nestránkováno, Munich (Germany). NATO Science and Technology Organization. URL: https://is.muni.cz/repo/1319597/2015-NATO-MSG-133-kypo-platform-cyber-defence-exercises-paper.pdf, doi:http://dx.doi.org/10.14339/STO-MP-MSG-133-08-doc.

[18] Vigna, G., Borgolte, K., Corbetta, J., Doupé, A., Fratantonio, Y., Invernizzi, L., Kirat, D., & Shoshitaishvili, Y. 2014. Ten years of ictf: The good, the bad, and the ugly. In *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*, San Diego, CA. USENIX Association. URL: http://www.usenix.org/conference/3gse14/summit-program/presentation/vigna.

[19] Chandra, Y. & Mishra, P. K. 2019. Design of cyber warfare testbed. In *Software Engineering*, Hoda, M. N., Chauhan, N., Quadri, S. M. K., & Srivastava, P. R., eds, 249–256, Singapore. Springer Singapore.

[20] Gephart, N. & Kuperman, B. A. October 2010. Design of a virtual computer lab environment for hands-on information security exercises. *J. Comput. Sci. Coll.*, 26(1), 32–39. URL: http://dl.acm.org/citation.cfm?id=1858449.1858457.

[21] Runeson, P., Host, M., Rainer, A., & Regnell, B. 2012. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st edition.

[22] Dunfjeld, M. 2019. Ansible role: Docker container. URL: https://github.com/mdunfjeld/ansible-role-docker-container.

[23] NTNU. 2019. Norwegian cyber security challenge 2019. URL: https://www.ntnu.no/ncsc.

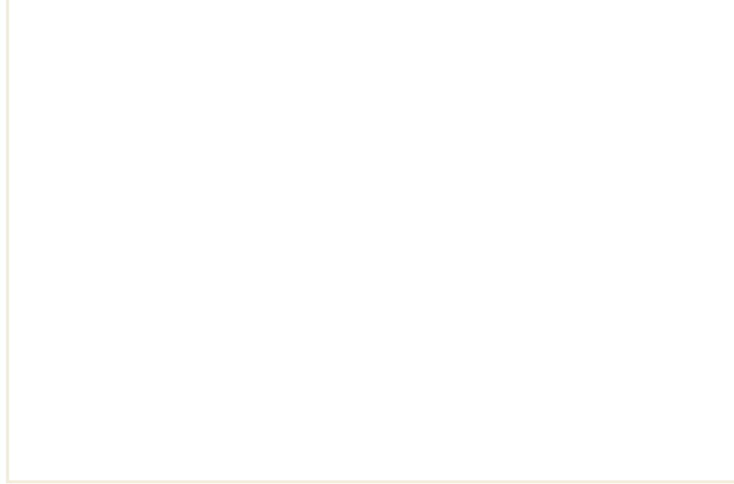
[24] Jenseg, O. 2019. Yet another ctf framework (yactff). URL: https://github.com/odinje/yactff.

[25] Dunfjeld, M. 2019. Ctfgen. URL: https://github.com/mdunfjeld/ctfgen.

[26] FAU Security Team: CTF team of Friedrich-Alexander University Erlangen-Nürnberg (FAU). 2015. Ctf gameserver. URL: https://github.com/fausecteam/ctf-gameserver.

A Interview questionnaire

1. Questions about CTFgen's efficiency
   1.1. How fast do you think CTFgen is compared
      a. To the way NCSC was originally deployed?
      b. To do doing it manually?
   1.2. How well do you think CTFgen automates the process of creating a jeopardy/AD scenario compared
      a. To Heat/Puppet/Ansible or something similar?
      b. To doing it manually?
   1.3. Is there anything that could have been done differently to improve efficiency?
2. Questions about CTFgen's usability
   2.1. How easy is it create a jeopardy/AD scenario using CTFgen?
   2.2. Do you think CTFgen would be useful for someone who is already proficient with existing tools or creating scenarios by hand?
   2.3. How much technical knowledge do you need to create a jeopardy/AD scenario using CTFgen compared
      a. To doing it manually?
      b. To using Heat/puppet/ansible or something similar?
   2.4. Is there anything that could have been done differently to improve usability?
3. Questions about completeness
   3.1. In your opinion, is there anything missing in the environment created with CTFgen?
4. Questions about flexibility
   4.1. How easy do you think it is to change aspects of the scenario created with CTFgen after it has been deployed?
   4.2. Is there anything that could have been done differently to improve flexibility?



NTNU
Norwegian University of
Science and Technology