City University London
School of Mathematics, Computer Science & Engineering

Thesis presented for the degree of Doctor of Philosophy

A machine learning approach for smart computer security audit

by

Konstantin Pozdniakov

1st supervisor: Dr Eduardo Alonso
2nd supervisor: Dr Vladimir Stankovic
Adviser: Professor Kevin Jones

December 2017

Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

I would like to offer my deepest gratitude to Dr. Eduardo Alonso for his support and guidance. I would like to thank Professor Kevin Jones for motivation and guidance throughout the course of this PhD. I also grateful to Dr. Vladimir Stankovic for his support as a second supervisor.

I’m thankful to my sister Victoria and to my parents Alexander and Olga for great support and care. This PhD work would not be possible without their love.

DECLARATION

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only copies made for study purposes, subject to normal conditions of acknowledgement.

ABSTRACT

This thesis presents a novel application of machine learning technology to automate network security audit and penetration testing processes in particular. A model-free reinforcement learning approach is presented. It is characterized by the absence of the environmental model. The model is derived autonomously by the audit system while acting in the tested computer network. The penetration testing process is specified as a Markov decision process (MDP) without definition of reward and transition functions for every state/action pair. The presented approach includes application of traditional and modified Q-learning algorithms. A traditional Q-learning algorithm learns the action-value function stored in the table, which gives the expected utility of executing a particular action in a particular state of the penetration testing process. The modified Q-learning algorithm differs by incorporation of the state space approximator and representation of the action-value function as a linear combination of features. Two deep architectures of the approximator are presented: autoencoder joint with artificial neural network (ANN) and autoencoder joint with recurrent neural network (RNN). The autoencoder is used to derive the feature set defining audited hosts. ANN is intended to approximate the state space of the audit process based on derived features. RNN is a more advanced version of the approximator and differs by the existence of the additional loop connections from hidden to input layers of the neural network. Such architecture incorporates previously executed actions into new inputs. It gives the opportunity to audit system learn sequences of actions leading to the goal of the audit, which is defined as receiving administrator rights on the host. The model-free reinforcement learning approach based on traditional Q-learning algorithms was also applied to reveal new vulnerabilities, buffer overflow in particular.. The penetration testing system showed the ability to discover a string, exploiting potential vulnerability, by learning its formation process on the go.

In order to prove the concept and to test the efficiency of an approach, audit tool was developed. Presented results are intended to demonstrate the adaptivity of the approach, performance of the algorithms and deep machine learning architectures. Different sets of hyperparameters are compared graphically to test the ability of convergence to the optimal action policy. An action policy is a sequence of actions, leading to the audit goal (getting admin rights on the remote host). The testing environment is also presented. It consists of 80+ virtual machines based on a vSphere virtualization platform. This combination of hosts represents a typical corporate network with Users segment, Demilitarized zone (DMZ) and external segment (Internet). The network has typical corporate services available: web server, mail server, file server, SSH, SQL server. During the testing process, the audit system acts as an attacker from the Internet.

1 Introduction

This chapter represents the context of the research, problem description and research challenges. It describes the motivation behind the project, research assumptions, objectives and the hypothesis of the project. Additionally, it specifies the thesis outline.

1.1 Background and motivation

All computer systems face the risk of being compromised. Systems become more and more complex, which leaves room for more errors left by developers. Those errors can provide the route for potential unauthorized intruders. There is a strong upward trend in the number of vulnerabilities per year (Secunia, 2015). This makes companies invest more resources into vulnerability assessment activities such as penetration testing.

Penetration testing is a process of attacking a computer system in order to reveal potential vulnerabilities (Mukhopadhyay, 2014). It helps to determine whether the system is secured against computer security threats. There are many ways to theoretically evaluate potential vulnerabilities in the system, however they need to be tested in reality by performing penetration testing activities.

The importance of penetration testing is beyond question. It helps to reveal security issues by performing real-life attacks. Regular audit decreases the financial or reputational risks related to data loss, network attacks, malware attacks, Denial of Service attacks and so on. It can address the information protection problems connected to firewall configuration vulnerabilities, access control methods and intrusion detection faults. By revealing the weak points of the target system, penetration testing helps to sort the risks according to their level of importance. It helps to redirect financial and human resources to deal with the most critical security issues in a minimal time frame.

There are different areas of penetration testing, such as social engineering, application penetration testing, and network penetration testing (Shah and Mehtre, 2015b). Social engineering is a process of psychological influence in order to compromise a security system (Atkins and Huang, 2013). It is based on human interactions and cannot be automated in nature. Application penetration testing, in turn, is a process of testing the software during its development cycle (Arkin, Stender and McGraw, 2005). It is an integration of security mechanisms in the application development cycle and is deeply integrated into the software development process. Network penetration testing is, therefore, a subject of interest for this research. Network penetration testing is a simulation of network attacks in order to trigger potential vulnerabilities (Denis, Zena and Hayajneh, 2016). By network penetration testing, we define the execution of exploits sequences with a combination of investigation actions (OS check, Port scan, etc.).

Automation of penetration testing is one of the biggest challenges in computer security. It reduces human error and decreases the resources spent by the penetration testing team. The automation of this process is considered nowadays as an automation of separate parts such as port scans and exploit compilation/execution, with a human as decision-maker.

Penetration testing is a crucial method for auditing network security by executing real world attacks to compromise the target defense systems (Bechtsoudis and Sklavos, 2012; Holik *et al.*, 2014; Rushing, Guidry and Alkadi, 2015). Such a computer security audit is traditionally made manually by cyber security experts with the help of security tools providing semi-automation during different penetration testing phases: target definition, environment analysis and attack execution (Stefinko et al., 2016).

Analysis of the penetration testing market showed the absence of smart penetration testing tools able to learn a decision-making strategy automatically. It means they have predefined sequences of attack/probe actions that can be chosen by user. They are unable to develop new attack strategies themselves, based on their previous experience. There is a variety of penetration testing/vulnerability assessment software on the market: Nessus/OpenVAS, Core Impact, Nexpose, GFI LandGuard, Qualys Guard, MBSA, Retina, Secunia PSI, Nipper, Saint, Metasploit and so on.

One of the most popular vulnerability scanners is Nessus (Tenable Network Security Inc., 2015): a security tool that makes a vulnerability analysis. Nessus scans for various types of vulnerabilities, ones that hackers could exploit to gain control or access a computer system or network. Furthermore, Nessus scans for possible misconfiguration (open mail relay, missing security patches and so on), default passwords, common passwords which it can execute through Hydra (an external tool) to launch a discretionary attack. Nessus has its own database of Network Vulnerability Tests (NVT) and they can be added manually as plugins. After the penetration testing task is set up by a computer security specialist, Nessus starts to execute NVTs, one by one, using the list of targets. Such a method does not provide any automation in decision-making. As a result, it will not be able to operate independently of the computer security specialist.

Core Impact (2015) is one of the most advanced vulnerability scanners on the market. It is audit software that replicates cyber attacks to assess the security of web applications, network systems, endpoint systems, mobile devices, users and wireless networks. Core Impact is the smartest penetration testing system on the market. It is the most automated software at present. It has an ability to provide automated plans for penetration testing.

The next popular penetration testing tool is Rapid 7 Nexpose (Nexpose 2015). It is a vulnerability scanner which aims to support the entire vulnerability management life-cycle: detection, verification, risk classification, reporting and mitigation. It has Metasploit integrated for vulnerability exploitation. The main advantage of Nexpose is the possibility of executing multiple scan engines at the same time. In this case, the vulnerability scanner can significantly decrease the scan duration. However, Nexpose cannot learn any attack strategy by itself.

Modern vulnerability scanners/penetration testing software requires better automation. They are still significantly dependent on computer security experts. For example, a Metasploit framework provides only the set of exploits and payloads. All configuration setup like information about the attack source, targets and payloads is done manually by penetration testers. It does not provide recommendations about the attack strategy as well. Modern penetration testing software cannot learn attacks by itself or make an attack strategy adaptive during penetration testing. The Core Impact vulnerability scanner, which is the most advanced to date, has the ability to build automated attack plans that can be used by an audit security officer for further penetration testing. However, it has a variety of problems. The Core Impact scanner creates a possible attack plan based on a model of the environment, which should be built first. It leads to extensive probe action executions against the target network and host configurations. The resulting attack plan significantly depends on the created model quality. While building the model and generating the optimal attack plan, the Core Impact scanner assumes that the configuration of the network stays static. It means that, in order to deal with the network/host configuration changes, it needs to recreate the environment model and recalculate the whole attack plan, which significantly decreases the adaptation properties of the tool. Some internal model parameters of this software are based on statistics derived from internal company simulations and expert knowledge. In cases where the tool does not have enough data for the new OS version or network configuration, it might have problems generating the optimal attack plan.

Therefore, there is a lack of automation in penetration testing software on the market. The tools have no ability to accumulate experience and learn the attack process during the computer security audit for future usage to become more autonomous. The software also has adaptation problems. It means, there is no possibility to update the attack strategy during penetration testing process depending on environment changes.

The research community is attempting to automate penetration testing in a more efficient way. The traditional approach to the process is based on modelling the target system and deriving possible attacks from that model (penetration testing plan) for later execution by the auditor. The automation of that process relates to the automation of model development and planning stages (Futoransky, Notarfrancesco, Richarte, & Sarraute 2003; Qiu, Jia, Wang, Xia, & Lv 2014).

There are different approaches to modelling potential attacks: attack trees, attack graphs, MDP and POMDP (Barik, Sengupta & Mazumdar 2016; Durkota & Lisy 2014; Qiu et al. 2014; Report 2010; Sarraute, Buffet, & Hoffmann 2013; Zhao, Shang, Wan & Zeng 2015).

An attack tree is a formal description of the attack model based on different attack sequences (Schneier, 2000). The root defines a particular goal of the attack and branches – different sub-goal sequences. Nodes of the tree can have OR relationships if they are alternatives or AND relationships if they represent the steps of achieving the goal. The tree construction is based on detailed information about the target security system. After the tree is developed, each node can be assigned a value (e.g. Possible/Impossible). This assignment is the subject of serious research by cyber security experts. As a result, it will be possible to derive attack sequences by processing the tree from the leaves to the root (Kordy, Kordy, Mauw & Schweitzer 2013; Zhu, Chen, Zhang & Xin 2008).

An attack graph is another formal representation of the attack model (Swiler & Phillips 1998). Compared to attack trees, an attack graph is more related to the representation of attack actions broken down into atomic components, described by preconditions and post conditions over system properties. They represent the information about the combination of atomic threads. Every node defines the state of the attack. Attack graph development can be based on attack statistics (e.g. vulnerabilities scoring system) and/or probe actions, performed in advance. Different paths of the graph represent different attack sequences compromising a target security system (attack vector).

Another representation of the attack model can be based on the Markov Decision Process (MDP) (Durkota & Lisy 2014). MDP is a discrete time stochastic process. The states of the process can define the attack state and transitions represent the set of actions available on the current attack step. The attack graph can be described by MDP as well. The attack model can be defined with a higher level of uncertainty using partially observable Markov Decision Process (POMDP). It is a generalization of MDP. It is assumed that system dynamics (the behavior of the system) is defined by MDP but the agent cannot fully derive the state. The agent observes it as a probability distribution over the set of possible states. Such a modelling approach is used in order to model uncertainty of the real world. An example of this approach in penetration testing is presented in Sarraute et al. (2013).

Traditionally, after the model is designed, a model based planning algorithm is used in order to generate optimal attack paths for penetration testing. It generates the sequence of attacks (attack vectors) based on the graph/tree traversal algorithms (Jha, Wing & Sheyner 2002; Ou, Boyer, & McQueen 2006; Qiu et al. 2014). If there are enough statistics regarding the transition probabilities between attack states then another approach can be used. It consists of the representation of the attack graph as MDP and selects the actions with lower cost in every state. As a result, the planning algorithm will generate a sequence of actions (attack vector) based on the MDP representation (Barik et al. 2016; Durkota & Lisy 2014; Sheyner, Haines, Jha, Lippmann & Wing 2002).

The problem of automated planning is a definition of the static known model of the environment, which should be built first. Imprecise models lead to simulation errors and generation of penetration testing action sequences having no sense. The environmental model of the most advanced penetration testing software, Core Impact, is based on an enormous amount of statistics, gathered from its own virtual infrastructure. Such a huge amount of simulations does not resolve the resource intensiveness problem. The static nature of the model makes it not environmentally adaptive. Every time the network topology changes, the model should be rebuilt, otherwise the planning algorithm will generate the wrong sequence of penetration testing actions.

Therefore, there is an unresolved problem of full penetration testing automation. None of the approaches give automation in terms of decision making. The existing penetration testing systems need to be supervised. Most of them just propose part-automation with a cyber security specialist as an attack strategy decision-maker. Existing approaches have a lack of adaptivity as well.

This thesis proposes a novel adaptive approach to automate the penetration testing process, including the decision making stage. The automation of the penetration testing will exclude a human from the actual penetration testing stage. It will help to mitigate a number of drawbacks related to the human factor, such as: inability to process huge volume of information, delays in decision making, work hours constrain and high cost of cyber security specialists.

In the same time, a given approach will raise a cyber security expert to the higher level of control. The specialist still will be able to tune up learning algorithms, develop learning strategies and create machine learning solutions, which will increase the performance of the automated penetration testing process. It will give an opportunity to tune up this process according to the given tasks and goals of the project. Therefore, a cyber security expert will be excluded from the routine level of penetration testing and will be able to redirect his skills towards a higher-level control tasks.

Our approach is based on application of a model-free reinforcement machine learning algorithm. The penetration testing system does not have a full predefined model of the environment and learns it by acting in the network using a Q-learning algorithm. This fact helps to mitigate all the disadvantages of environmental model development. In order to generalize the solution, the algorithm was modified using an approximator based on artificial neural network (ANN) and recurrent neural network (RNN). In order to prove the concept, an audit tool and testing environment was developed.

1.2 Research aims and objectives

Current machine learning approaches allow the development of a tool that significantly advances the automation of vulnerability discovery in an arbitrarily configured network. The overall aim of the research is to:

> Apply a machine learning technique to automate the penetration testing process in order to make it learn the attack strategy itself on the go and make decisions on penetration testing actions:

- ➢ O1: Identify the weaknesses of modern approaches to penetration testing and problems raised in the scope.
- ➢ O2: Investigate machine learning algorithms that can be used in the cyber security scope.
- ➢ O3: Use the theoretical ground defined in O1 and O2 to develop an adaptive automated model-free learning approach to the penetration testing process.
- ➢ O4: Generalize the approach revealed in O3.

1.3 Contribution

The thesis makes a contribution to the automation of penetration testing systems and the vulnerabilities assessment scope:

- The automation of penetration testing was examined in the context of a self-learning penetration testing system. It was shown that it can learn the attack strategy itself, while in action.
    - The penetration testing process was modelled as MDP.
    - The model free approach to the automation of penetration testing was applied, based on a traditional Q-learning algorithm. The proof of concept tool was developed. It showed the effectiveness of the chosen approach.
    - Testing infrastructure with 80+ virtual machines was built, representing a typical corporate network.
- The generalization of the approach to the automation of penetration testing based on the approximation.
    - The model free penetration testing process was generalized using an approximator.
    - A new way to represent hosts based on a specific feature set was defined.
    - Deep architecture of the approximator Autoencoder + ANN was developed.
    - Deep architecture of the approximator Autoencoder + RNN was developed.
    - Performance comparison of Deep architectures was made.
- The automation of the new vulnerabilities search, buffer overflow vulnerabilities in particular.
    - The model free approach to the automation of new vulnerability discovery was applied, based on traditional Q-learning algorithm. The proof of concept tool was developed.

1.1 Thesis outline

The thesis is organized as follows:

- Chapter 2 discusses the literature review of previous works that have been done to automate the penetration testing process. Problems and challenges are highlighted. The general approach to penetration testing is discussed as well. Information on different machine learning techniques is presented.

- Chapter 3 presents the Markov decision process (MDP) concept. It also describes the definition of the penetration testing process in terms of MDP, using transition function, reward function, action and state spaces. It represents the definition of buffer overflow vulnerabilities. Information on the Q-learning algorithm is given. The concept of Q-learning approximation is discussed, as well as deep machine learning architecture, including autoencoder, ANN, RNN. Additionally, the penetration testing tool architecture and testing environment is presented. A set of experiments is briefly introduced.

- Chapter 4 illustrates the actual results of the experiments. It describes the experiment setups, hypotheses, experiment outcomes with different sets of hyperparameters and acting policies. The performance of learning algorithms is compared.

- Chapter 5 concludes with the result of the undertaken research. It compares the achievements with the research scope results and gives future research directions.

2 Literature review

2.1 Introduction

This chapter presents a more focused review of the literature of the presented context. The Attack graphs and Attack trees sections describe the research related to the development and analysis of such models as well as to automation attempts of their generation. The Planning section relates to the planning approach to derive the attack sequences for computer security audit. The Tools section describes the literature on automation attempts of penetration testing by aggregating existing attack tools. The Alternative approaches section presents the rest of the literature related to the automation of penetration testing. The last part of the literature review concerns machine learning applications within the field of cyber security. It also gives an overview of machine learning techniques used in that scope.

2.2 Attack trees

In Schneier (1999) and Schneier (2000), the attacks are modelled using ‘Attack trees’. This is a visual representation of the attack. It can be formally described using formal language. The goal of the attack is defined by the root of the tree. It can include subtrees as well. The nodes are structured in hierarchical order. Nodes enforce their sub-nodes to be executed first. The nodes can have an OR or AND property and are associated with the cost of execution.

The attack tree model was upgraded in Tidwell, Larson, Fitch, & Hale (2001). It was presented a new attack tree model for Internet attacks. It was developed the language to aggregate attack behavior descriptions as well. The novelty of the approach was in adding precondition and post condition assertions. Such parametric attack trees provide a framework for expressing exploits and multi-stage attacks.

In Moore, Ellison, & Linger (2001), the authors extend the previous attack tree approach by adding reusable attack patterns for characterizing an individual type of attack and attack profile to organize such patterns.

The research outlined in Mauw & Oostdijk (2006) proposes the formalization of earlier work by Schneier (1999). It formulates the conditions that allow particular manipulations with attack trees. They try to extend the approach by defining attacks as multi-sets. The paper indicates the scalability problems.

Recent literature also has examples of applying attack trees for penetration testing. In Sarraute, Richarte & Obes (2013), the attacks are modelled as an attack tree. It is later used in order to find the optimal attack path. The tree is probabilistic with nodes having asset or action types. An asset node is connected by OR relations with all actions producing this asset (attacking agent – exploit). An action node is connected by AND relations with all its requirements. By iteratively decreasing groups of tree nodes, a single attack path is generated, which minimizes the expected cost associated with attack actions. The only actions used by authors were: exploit execution, TCP/UDP Connectivity test, OS probe.

Kordy et al. (2013) present a tool for modelling attack trees and for their quantitative analysis. It can be used to modify, create and hide trees. The tool gives the framework to build security models and graphically analyse them.

In Zhao et al. (2015), a rule based tree method is presented. The idea of the automation is based on a predefined chain of rule based trees created by a cyber security specialist. The attacks are performed and assessed according to that information. The problem in this approach is the need for the creation of, and regular updates to, those trees of rules.

The attack trees approach is not as popular in recent literature. This is related to the problems associated with this attack model. An attack tree is not adaptive and cannot react to the changing environment. It can be successfully used to model the attacks on static configurations or in cases where the dynamics of the target system are not important. Therefore, in order to adapt to the changing environment, attack tree must be recreated each time the environment changes, which is computationally intensive. In case Attack trees also have a state space expansion problem. They are not able to deal with attacks on large networks. With the size of network, the tree grows significantly quickly. Attack trees cannot represent the possible loops of the attack states as well.

The more advanced alternative of network attack representation is an Attack graph model.

2.3 Attack graph

An attack graph is a data structure used to represent all possible attacks on a network. This attack model is usually related to the planning based approach to penetration testing. Attacks are divided on atomic actions described by preconditions and post conditions over properties of the system under attack. An attack graph defines an analytical knowledge about potential threats, which can occur according to the possible combination of actions. It represents possible attack plans. In order to generate such plans, planning techniques are used.

The attack graph approach was first proposed by Swiler & Phillips (1998) for network-vulnerability analysis systems. The approach needs a predefined database of common attacks, consisting of atomic steps, to be used as an input to the system. It also requires the network configuration, topology information and topology profile to be able to function. By combining all the input information together, the analysis system creates a superset attack graph. Nodes represent the current stage of attacks. The arcs represent the attacks themselves. The probabilities of success or costs can be assigned to the arcs. In that case, any graph algorithm can be used to identify the attack path with the highest probability of success.

Other early researchers (Templeton & Levitt 2000) approach the attack graphs as an action model. They propose the description of a model of attacks based on the requirements of the attack elements, their satisfaction of the requirements of each other and the method of composition of those components into complete attacks. Such a model does not need prior knowledge of the particular scenario. Therefore, such a method can be used to describe unknown attacks, which is represented as an abstracted overview of the atomic threads.

In contrast with Templeton & Levitt (2000), Ritchey, Allen, Church, & Ammann (2000) approach the attack graph not as an action, but as the state space model. They propose to generate different attack scenarios by model checkers. The idea behind the approach is to define vulnerabilities as a state machine suitable for a model checker. After definition, the model should be checked against the hypothesis that particular assets cannot be reached by an attacker. As a result, a model checker will prove it or will propose the steps of the attack as a counterexample.

A similar model-check approach can be found in Sheyner et al. (2002) and Sheyner & Wing (2004). These articles have the same approach to the vulnerability assessment process. However, the analysis stage is added after the attack graph construction. First, they create a model of the tested network as a state machine, where state transitions related to the atomic attacks are executed by an attacker. The example of the security property is ‘the attacker should never obtain root access to host A’. It is checked by the model checker and the attack graph is generated based on counter-examples, violating the property. The idea is to identify possible ways to transition into the ‘successful penetration state’, which defines successful exploitation of the potential vulnerability. During the last stage, the graph is parsed and the original meaning of state variables is recovered as they were represented in the network intrusion domain.

Later research proposed the idea of monotonicity of the attacks in the attack graph approach to penetration testing (Jajodia, Noel & O’Berry 2005; Jajodia & Noel 2010). The described approach is related to modelling of the target network based on its security conditions. The exploits that a potential attacker uses are modelled as transition rules between security conditions. A presented tool, developed by the authors, builds an attack graph and computes a possible attack path as a combination of exploits compromising particular network resources.

2.4 Planning

Futoransky et al. (2003) proposes a framework in order to automate risk assessment and penetration testing, in particular. It can also be used for attack simulations. It is based on the previous publication (Obes, Sarraute, & Richarte 2003). The approach is related to a multi-dimensional attack graph generation based on quantified goals, probabilistic assets and complex cost function. In their model, an attack is defined as a group of agents executing sequences of actions, obtaining assets to reach the goals. Agents are collaborating to accomplish the goals.

The authors offer three approaches to the attack building process. Before building actual attacks, they propose to gather all the information about the target environment. Based on that knowledge, an attack graph of all possible actions is developed and a planning algorithm is used to build the attacks. This approach works only for very small instances of the problem (less than 20 hosts network). It's related to the exponential grow of the attack graph size with number of hosts and possible actions.

Another approach is to model the environment as a Markov Decision Process (MDP) with probabilistic transitions. The transition model is derived from the huge amount of statistics gathered earlier by continuous simulations of the attacks against different network configurations. It was provided by the Core Impact pentesting software lab. After the model is constructed, the planning algorithm can be used to build the attacks.

The third approach is similar to the first one but with different topology of the attack graph. The authors propose to make it multilayer by alternating layers of goals and actions. There are connections between each goal and to the actions that might satisfy it and all actions to the set of sub-goals that it requires. During the attack graph construction phase, all quantifiers are not expanded and are treated as a single element. It helps to avoid state space explosion.

The outcome of the research is a proposed framework with automative planning using a preconstructed environment model. It is implemented in Core Impact Classic pentesting software to support the penetration testing process.

The problem with the authors' approach is in the fact that automative planning can be used only in static known models of the environment. The planning process is a model simulation. In order to reveal the potential attacks, the environment should first be modelled. In cases where it is not modelled precisely, the predicted attacks might have no sense.

Another problem is in the construction of the environment model itself. The authors used statistics derived from the Core Impact simulation data. This information is continuously connected in virtual environments by executing different attack actions against different network/host configurations. Therefore, without access to a huge amount of information statistically describing the pentesting actions' outcome, the precise model might not be accurate enough for the penetration task.

After the model of the environment is built, it cannot be changed during the planning process. It means that if the network topology or configuration of the host is changed, the planning will miss those changes due to the lack of adaptation.

In Hoffmann (2015), an overview is presented of different modelling approaches in order to simulate penetration testing. This work addresses the description of earlier models of the environment used in Core Impact research (Futoransky et al. 2003; Sarraute et al. 2013), as well as their extensions for future experiments. The work covers only the modelling stage of the penetration testing environment without application of automative planning. The paper reflects the search of a new environment representation model for Core Impact lab, mentioned in earlier research by the authors.

The models are systematized according to the uncertainty of the environment and attack component interaction. The authors mention the attack graph approach, which totally abstracts from the uncertainty and opposite POMDP extreme case incorporating the attacker's initial knowledge. The MDP model is also introduced that is a middle ground between those two extremes.

The attack graph approach is based on the attack graph model described using Planning Domain Definition Language (PDDL) with Planning algorithm application to generate possible attack paths. It is assumed that: there is no uncertainty about the network graph topology (known network graph); no uncertainty about host configuration (known host configuration); the network is static; actions are monotonic (once an asset is obtained by an attacker it cannot be lost); and that Action=hops (non-static preconditions and effect of each action corresponds exactly to the 'hop' in the network graph).

The POMDP model is more realistic. This approach models the attacker's incomplete knowledge about the network as an uncertainty of state, which is a probability distribution over possible network graphs and host configurations. An initial probabilities distribution encodes the prior knowledge to the attack phase. The states describe the network configuration and the status of the attack. In that model, it is assumed that the network graph is known and that there is no uncertainty about network topology, the network is static, actions are monotonic and equal to hops. The only assumption to be relaxed as a result of remodelling is that the host configuration should not be known anymore. This is the current model that the Core Impact lab is integrating at the moment.

Modelling penetration testing process as an MDP is a middle ground between Attack graph planning and POMDP. The idea behind that approach is a formulation of the uncertainty of an attacker in terms of action outcomes, hiding the details of host configurations. The MDP model is an abstraction of POMDP. In the MDP case, the only information that should be defined is the success probability of exploit executions. The exploit can only succeed or fail. It does not consist of any sub-actions which cause hidden states, that leads to moderate accuracy decrease. Exploit execution is an atomic action. MDP representation significantly decreases computational complexity.

In Reddy & Yalla (2016), the mathematical analysis of the penetration testing was performed. An application of the novel algorithm is discussed to create a mixed test plan for the cyber security audit. The authors highlight four stages of penetration testing:

1. Information collection about the target system.
2. Post analysis of gathered data.
3. Actual exploitation.
4. Result analysis and report.

During the first stage, the cyber security professional uses probe actions to collect the data related to the target. At the second stage, the information is analysed, the attack strategy is manually created and the attack actions (exploit execution) are performed. Therefore, the penetration testing process is defined as a composition of probe and attack actions. Such a structure for penetration testing is also recommended in the NIST standard addressing network security (Wack et al., 2003).

2.5 Tools aggregators

In Stefinko et al. (2016), a comparison of manual and automated penetration testing is presented. The philosophy of automation in this paper is related to the automation of each stage of the penetration testing as much as possible, using different audit tools like Kali Linux/Metasploit framework. The attack strategy decision-making is performed by the cyber security expert. The approach decreases the load caused by routine actions. However, it still requires a highly skilled professional to use such systems and does not exclude the human factor. By excluding the human factor it's meant the automation of decision making process during actual attack.

An easy-to-deploy penetration testing platform is developed in Duan et al. (2008). It uses a LiveDVD SolarSword developed by the Opensolaris OS team. The authors propose a semi-automatic pentesting method with multi-target start conditions. The approach allows to set multiple scripts against multiple targets at the start, which will be used accordingly. It is based on the originally developed approach using post-processing data by a control centre. This module provides administrative support to the pentest system. The control centre stores and performs all automatic tests manually created by the system supporter, including scripts, templates, semantic rules and so on. The penetration testing process consists of three stages: information gathering, vulnerability exploitation, and results analysis. During the first phase, the system uses the LiveDVDSolar Sword to scan the network environment. After scan results are transferred to the control center, the attack strategy decision should be made. The decision-making process is based on a CMU tool, which generates the attack graphs (Sheyner et al. 2002; Wing 2008). The CMU internal decision-making algorithm was used. The problem with such an approach is that the system is hard to support and maintain. The scripts that the system uses are created manually by cyber security experts for the user of the system. They need to be updated regularly. It means that the time and expertise that was decreased for the user increases for the specialist that supports this penetration testing system. Therefore, the human factor is not decreased in general.

In Shah & Mehtre (2015a), the Net-Nirikshak tool is presented. This tool provides a sequential automation of the penetration testing process for a limited range of tasks. It scans the target system to derive versions of the services installed, and checks the possible vulnerabilities against the National Vulnerability Database (NVD). As a result, the tool generates a report about found vulnerabilities. It also can perform SQL injection attempts.

2.6 Alternative approaches in the cyber security domain

In Pan & Li (2009), a new penetration testing approach for an E-commerce authentication system is presented, based on different generation test cases. First, it is proposed that the new input program sets be generated from the existing ones in order to trigger vulnerabilities. For this purpose, a linear system based on the relaxation iteration algorithm (Gupta & Mathur 1998) was developed. During the second phase, the pentest cases are refactored and used again. In the last stage, the dynamic taint propagation (Haldar et al. 2005) method is used to verify the possibility of potential vulnerabilities being feasibly exploited.

Qiu et al. (2014) proposed automation using penetration testing scheme generation. By penetration testing scheme, the authors mean a description of execution of penetration testing. Such a description is constructed based on manually developed rules by penetration testing consultants. After automatic penetration testing scheme generation, the vulnerability scanner uses the vulnerability database to check the target system. After the potential vulnerabilities are revealed, the exploitation mechanism starts actual exploit executions, one by one.

Therefore, there is a lack of research related to the automation of penetration testing. None of the approaches give real automation of penetration testing in terms of attack strategy. The existing penetration testing systems need to be supervised or need to have a predefined set of rules. Most of them just propose part-automation with a cyber security specialist as an attack strategy decision-maker.

The current research proposes a new approach to the automation of penetration testing. It suggests letting the penetration testing tool (without any prior knowledge) explore/attack the target system and, using machine learning, learn by trial and error the right attack strategy on the go, according to its experience. In order to understand which machine learning approach to choose, a literature review on machine learning was carried out.

2.7 Machine learning approach

There is a variety of machine learning approaches: supervised learning, unsupervised, semi-supervised, reinforcement learning (Erik 2014; Kotsiantis et al. 2006; Sutton & Barto 2012).

Supervised learning methods (Aguilar and Riquelme, 2007) are based on a function that maps input to desired output. This approach is often used for classification problems. The learning process is based on feeding pairs of data to the algorithm: input and the respective output. After processing those pairs, the algorithm is able to predict the correspondent output for the particular input. Such an approach demands a large amount of data and quantity of iterations. Such an approach is limited to the labeled data only.

Unsupervised learning (Baldi, 2012) is the opposite to the supervised learning approach. Its data is not labeled. It automatically associates the input information with possible existing classes to reduce the dimensionality of data. The algorithm analyses data itself and tries to find similarities and differences in given input to find existing patterns.

The third type of learning algorithm is reinforcement learning (Xu, Zuo and Huang, 2014). The agent that learns according to this algorithm learns by trial and error. It travels through states by performing actions without any prior information and receives the reward signal from the environment. If the reward signal is high, it will be motivated to do such actions more often, if the reward signal is low, or negative, it will avoid those actions. Therefore, after a period of exploring, the agent will be able to learn some sort of desirable behavior or acting policy. All types of learning are widely used in computer science. The next section describes the application of those algorithms related to the cyber security domain.

2.7.1 Machine learning approaches applied in the cyber security domain

According to the literature, there is a variety of specific cyber security tasks related to the application of machine learning approaches: Intrusion detection systems (IDS), Phishing/Spam detection, Behavior biometrics, Security of human interaction proofs, Cryptography, Code analysis (Ford and Siraj, 2014).

Most of the literature about the application of machine learning in cyber security concerns IDS systems, rather than vulnerability assessment/penetration testing. An intrusion detection system is a software solution that monitors the network in order to identify malicious activity.

A hybrid approach to the IDS development based on the ability to tune its own classifiers and feature extraction algorithms is represented in Anfilofiev et al. (2015). The novelty of the approach is the specific usage of genetic algorithms. There is a lot of literature related to the application of semi-supervised learning in IDS systems, starting with older publications such as Lane (2006) and Aslam et al. (2006), up to the most recent ones such as Wagh (2014), Shinde & Parvat (2014) and Sahu et al. (2014).

The machine learning classifier is applied in Khan et al. (2014) for DNS DDOS attack detection. The authors propose calculation of the Lyapunov exponent in order to access the complexity of network packet flow. After the exponent is calculated, its magnitude is used to classify whether or not the traffic is normal. The method detects a DNS DDOS attack with 66% accuracy. The experiment was used with static data, but the authors claim that it can be applied with live traffic.

Botnet detection is an area of interest for machine learning application (Haddadi et al., 2014). Haddadi et al. analyse HTTP traffic to detect bot control commands. For this reason, the classifier C4.5 (Hssina et al., 2014) and Naive Bayes was used. The experiment showed that C4.5 performance is better than Naive Bayes. Zhang et al. (2015), Di Mauro & Longo (2014) and Moore et al. (2005) show a similar application of machine learning in network traffic detection. Different algorithms discussed are: random forest, correlation-based classification, semi-supervised clustering and one-class SVM.

Abnormality detection is a popular method in intrusion detection scope (Gou et al., 2009; Tsang n.d.; Wong & Lai, 2006). The main idea for this is to improve the IDS detection rate.

A genetic immune model which is adaptive to the rule-based IDS (Xu, Sun and Xiaojun, 2003; Tomandl, Fuchs and Federrath, 2014) is proposed in Yang et al. (2013). It uses the artificial immune system model (AIS model) based on the specific algorithm described in (Yanchao et al., 2001). The AIS is an adaptive system, inspired by immunology theory and biological immune models. It's a class of rule-based machine learning systems, which algorithms are modeled after the immune system's characteristics of learning and memory. The paper proposes to model IDS as the state transition system, leading to the target compromised states. States and transitions are expressed in a double DNA chains pattern, The Machine Learning and Cybernetics (2003) paper is interesting in terms of penetration modelling. The author considers a possible attack as a collection of states leading to the compromised state. The research is based on the old DARPA works (Dasgupta & Brian 2001; Lippmann et al. 2000).

Machine learning is also applied in order to resolve the Phishing/Spam detection problem. In Abu-Nimeh et al. (2007), the authors compare different machine learning techniques for phishing prediction such as: Logistic Regression (LR), Classification and Regression Trees (CART), Bayesian Additive Regression Trees (BART), Support Vector Machines (SVM), Random Forests (RF), and Neural Networks (NNet). They use 49 features and 2889 emails to train their classification system.

Another classification framework is presented in (Zhuang et al., 2012). The research is based on analysis of Kingsoft Internet Security Laboratory malware collection. The proposed framework is intended to join different classification solutions for better Phishing/Spam detection. The specific feature set is proposed as well.

The behavior biometrics domain also uses machine learning techniques. Its main goal is to authenticate the user by its behavior. Users classification is described in El Masri et al. (2014). Microsoft Word was used as a test application. The main idea was to let users use MS Word and let the system create a pattern of interaction. After a profile is created, the system authenticates the user according to his/her actions.

Another example of machine learning application in the current domain is keystroke analysis (Revett et al., 2007). Authors examined Login/Password type pattern of 50 users and collected a series of attributes used for classification of the profiles and further authentication. They apply Probabilistic Neural Network (PNN) in order to reduce behavior training time and increase classification accuracy of users. The accuracy of the method was 90%.

Machine learning is used in Breaking Human Interaction Proofs related researches (CAPTCHAs). Chellapilla and Simard (2004) investigate the vulnerability HIPs. The goal of the research is to make the CAPTCHAs more resistant to attacks based on pattern recognition techniques. The authors performed six experiments with EZ-Gimpy/Yahoo, Yahoo v2, mailblocks, register, ticketmaster, and Google HIPs. They revealed that the most difficult stage for CAPTCHA recognition is segmentation.

In Shabtai et al. (2010), the authors present work on automated static code analysis. They used a machine learning classifier in order to process the huge database of *.apk files. Those files stored the android application source code. As a result, malicious apk files were identified.

In Younis & Malaiya (2014), it is proposed that the vulnerability exploitability metric (Alhazmi and Malaiya, 2005; Manadhata and Wing, 2011) be based on software structure properties such as dangerous API calls and reachability. Using the metric as a feature, the researchers constructed a model which uses machine learning techniques such as SVM (Lin and Chen, 2008; Wang, 2008) for automatically predicting the risk of vulnerability exploitation.

In (Yu and Cao, 2006) the machine learning approach is extended to the cryptography domain. Chaotic neural networks are used in order to generate binary sequences used for masking plaintext. This method makes the cyphertext more secure.

Therefore, the literature review showed applications of machine learning in the cyber security domain related to Intrusion detection systems (IDS), Phishing/Spam detection, Behavior biometrics, Security of human interaction proofs, Cryptography, Code analysis. However, there is an absence of application of machine learning techniques to the automation of penetration testing. This fact leaves the problem of penetration testing automation unresolved.

2.8 Chapter summary

According to the literature review, there is a lack of research related to the automation of penetration testing. The problem of full automation including decision making is not resolved. Modern tools, automating different stages of penetration testing, do not provide the automation of the decision making process. Model checker and planning approaches have serious resource consumption problems to provide a reasonable attack strategy due to the huge state space and difficult environmental models. The problem of human resources load is still unresolved. The existing penetration testing systems need to be supervised or need to have a predefined set of rules. Most of them just propose part-automation with a cyber security specialist as an attack strategy decision-maker. Existing approaches are not adaptive and not able to act in changing environments.

3 Methodology

3.1 Introduction

This chapter describes the methodology for the current research. The main goal of the project is to automate a penetration testing system by making it learn attack strategies on the go. As an additional result, the system will be able to learn multistep attacks and to search for zero-day vulnerabilities.

Based on the literature review, the reinforcement learning approach was chosen in order to reach the goals of the project. Compared to the other approaches, it does not require the labeling of input data and is ready to analyses it on the go. Because of the fact that the testing environment is not known, the main requirement for the algorithm should be that it is model free. There are not many alternatives in model-free based reinforcement learning algorithms. The choice is mainly between the well-known Q-learning algorithm and its competitor – SARSA (State-Action-Reward-State-Action) (Tokarchuk, Bigham and Cuthbert, no date; Nissen, 2007; Harm van et al., 2009). These algorithms are very similar. The only difference is in the fact that Q-learning compares the current state versus the best estimated next state, but SARSA compares the current state versus the actual next state. SARSA may work faster sometimes, but Q-learning has a bigger advantage. It is an off-policy algorithm (Geist and Scherrer, 2014). Q-learning also converges to optimality in tabular cases. That is why it was chosen as a reinforcement learning algorithm. Tabular cases are the cases when Q-values are stored in a table in the memory for every MDP state-action pair, contrary to their calculation using approximator.

Q-learning works very well for tabular cases. These are the cases that have a fairly small state/action space and the knowledge (Q-value) is accumulated in a table. The bigger the state space, the more inconvenient the usage of the table becomes. For those reasons, in real-life tasks, the Q-function is approximated (Irodova and Sloan, 2005; Melo, Meyn and Ribeiro, 2008; Xu, Zuo and Huang, 2014). The universal approximator in machine learning is a neural network (Mhaskar and Hahm, 1997; Most, 2005). This is a collection of neural units (neurons), modelled by their architecture, in the same the way that a human brain processes tasks.

The penetration testing task can have enormous state/action space. That is why it was very important to choose the right features that would describe the state/action space for the current problem. The more relevant features are used, the more precise the state/action space is described. However enormous quantity of features will result in significant load of the learning algorithm input. As a result, the learning algorithm might become so inefficient, that it would not be able to resolve given task in foreseeable time. To resolve this problem, the features set can be compressed into smaller features space prior to actual learning. For those reasons Autoencoder was used in current project. Autoencoder is a neural network that can be learned to compress its input to the smaller hidden layer with minimal error. Therefore, the feature set inputted to the Autoencoder will be compressed into the new smaller feature set. After, it can be inputted to the approximator of the actual learning algorithm, that can be defined using other neural network architecture.The detailed concept of Autoencoder will be explained further in this chapter.

Therefore, in order to minimize the load on the neural network as approximator, deep architecture was used. Instead of solely the neural network, an autoencoder and approximator neural network were connected together (Pascal and Hugo, 2010; Hinton, Krizhevsky and Wang, 2011; Lore, Akintayo and Sarkar, 2015). In the literature, such an approach is called 'deep learning'. This architecture resolves two problems: the autoencoder decreases the quantity of features representing state/action space; and the neural network approximates the state/action space based on a smaller number of features left. The type of neural network that has been chosen is a recurrent neural network (RNN) – in particular the Elman type (Li, Deng and Zhang, 1997; Samek, 1999; Zhang, Tang and Vairappan, 2007). It is a special type of network that has an additional neuron feeding to every neuron in the hidden layer of its previous value. Therefore, there is an analog of memory implemented. This architecture is distinct from others due to its ability to learn sequences of actions. This type of RNN has the vanishing gradient problem (Pacanu et al. 2013; Bengio 1994; Liu n.d. 2012). In other words, it is not able to learn long sequences of data. For these reasons, in the research community, Long Short-Term Memory (LSTM) RNNs are used (Lipton et al. 2015; Gers et al. 2000, 2002).

The type of RNN to use is dictated by the problem itself. In penetration testing, the common sequences of actions to use are not that long. It is usually the combinations of one-two exploration actions (check OS, portscan) + remote exploit execution + local exploit execution. This simplification is reasonable for exploit-type attacks. Our work was not focused on breaking the attacks on the smaller components, which considered as future work. Therefore, it means that the Elman RNN is a convenient choice. It has a simpler learning process and easier development phase. It should be mentioned that, in order to deal with the vanishing gradient problem in Elman RNN, the simple ReLU (Rectifier linear unit) activation function (Xu et al. 2016; Caglar et al. 2016) in the activation layer can be used. ReLU activation function is the function defined as a positive part of its argument.

This chapter on methodology begins with the problem description in the section 'Problem representation'. It outlines a general MDP representation and modelling information. The next section is concerned with the Q-learning description with tabular case and approximation. The artificial neural networks for approximation are described, followed by the autoencoder architecture. The final section of the chapter presents the experiment outline.

3.2 Problem representation

3.2.1 Markov Decision Process

Penetration testing can be very well described by the Markov Decision Process (MDP). It is a mathematical framework for modelling sequential decision-making problems. Given an MDP *M*, this relation is performed as follows:

Let $t \in$ N define the time or stage

Let $X_t \in X$ and $A_t \in A$ define the random state of the system and the action chosen at time t. After the action is selected, the transition will be made:

$$(X_{t+1}, R_{t+1}) \sim P_0(\cdot \mid X_t, A_t) \quad (1)$$

Where X_{t+1} is random and $P(X_{t+1} | X_t = x, A_t = a) = P(x, a, y)$ holds for:

$x, y \in X, a \in A$.

Further, the acting agent does the following:

1. Observes the next state X_{t+1} and reward R_{t+1}
2. Chooses a new action $A_{t+1} \in A$
3. Repeats the process

The main goal of the agent is to learn the best way to choose actions in order to maximize the expected total discounted reward.

The agent can select actions in any moment based on the observed history. The rule of that selection is called 'behaviour'. The behaviour of the agent and some initial random state X_0 represent a random state-action-reward sequence $((X_t, A_t, R_{t+1}); t \geq 0)$ where (X_{t+1}, R_{t+1}) are related to each other according to the formula (1) and A_t is the action given by the behaviour according to the history $X_0, A_0, R_1, \ldots X_{t-1}, A_{t-1}, R_t, X_t$.

The return of a behaviour is represented as the total discounted sum of the rewards incurred:

$$R = \sum_{t=0}^{\infty} \gamma^t R_{t+1}$$

Therefore, if $\gamma < 1$ then the rewards in the future are worth exponentially less than the reward received at the first stage. It is a discounted reward of MDP. In the case of $\gamma = 1$, the MDP is called ‘undiscounted’.

The main target of the decision-making agent is to maximize the expected return, irrespective of how the process started. This maximizing behavior is called ‘optimal’.

The next subsection demonstrates the learning process of an action strategy.

3.3 Attack strategy learning

3.3.1 Q-learning

The goal of the penetration testing system, as mentioned in the objectives section of this research, is to learn the attack strategy itself, directly from the acting process. That is exactly what Q-learning does in MDP. Its aim is to approximate the optimal action-value function Q* directly. Q-learning can be presented as a sample-based, approximate version of value iteration, generating a sequence of action-value functions (Q_k ; k >= 0). If Q_k is closer to Q*, the policy that is greedy related to Q_k will be closer to being the optimal policy.

The Q-learning algorithm works in the following way. A finite MDP is defined as $M == (X, A, P_0)$ with the discount factor γ. The algorithm stores an estimate $Q_t(x,a)$ of $Q^*(x,a)$ for each state-action pair (x, a) ∈ $X \times A$.

Observing $(X_t, A_t, R_{t+1}, Y_{t+1})$ with updated estimates according to the formulas:

$$\delta_{t+1}(Q) = R_{t+1} + \gamma \max_{a' \in A} Q(Y_{t+1}, a') - Q(X_t, A_t)$$

$$Q_{t+1}(x,a) = Q_t(x,a) + \alpha_t \delta_{t+1}(Q_t) ||_{\{x = X_t, a = A_t\}}, \quad (x,a) \in X \times A$$

The algorithm of Q-learning is presented below:

Function: Q-Learning (X,A,R,Y,Q)

Input: X is the last state. A is the last action, R is the immediate reward received, Y is the next state, Q is the array storing the current action-value function estimate

```
1: $\delta \leftarrow R + \gamma \max_{a' \in A} Q[Y, a'] - Q[X, A]$
2: $Q[X, A] \leftarrow Q[X, A] + \alpha \cdot \delta$
3: return Q
```

Where α is the learning rate and $^{\gamma}$ is the discount factor for future rewards.

Q-learning can act differently, depending on the chosen policy. The policy of the decision-making agent is called 'ε-greedy policy'. It lets the agent choose between random and greedy (estimated maximization of the reward) actions with 1-ε probability. In the case where ε is 1, the agent will act fully randomly. If it is 0, the agent will act only according to its gained experience. There are more adaptive acting policies such as softmax, which let the agent act randomly at the start to explore the state space first and, after some time, to choose actions based on experience.

In summary, reinforcement learning is an unsupervised learning technique, in that the attacking agent does not have a pre-existing model of the system, which resembles real-life conditions where an attacker does not have any knowledge of which attacks are more promising. This is unlike supervised learning because the agent is not trained with examples of what would constitute a successful attack, according to a given set of relevant attributes; rather, it learns from scratch. It is worth emphasizing this point: reinforcement learning offers a mechanism for an agent to learn the optimal sequence of actions (given metrics of its behavior) starting from complete ignorance. Of the various algorithms that solve the reinforcement learning problem, Q-learning is chosen here because of its simplicity and its convergence properties – it has been proven to converge to optimality with a probability of 1 in tabular cases Watkins & Dayan (1992).

3.3.2 Q-learning approximation

Q-learning is a reinforcement learning algorithm that learns an action-value function, given the expected utility of performing a given action in a given state. The traditional implementation is based on a Q-learning table where the utility function (Q-function) stores its values. As complexity of the state space increases, the standard Q-table approach does not scale well. This problem is usually addressed by using an approximator. Q-learning with a Q-function approximator does not learn Q-values one by one, rather, it learns the Q-function itself. In this project, an artificial neural network was used as an approximator for learning the Q-function.

3.3.3 Artificial neural networks

Artificial neural networks are models inspired by the structure and functionality of biological neural networks, as shown in Figure 3.1.

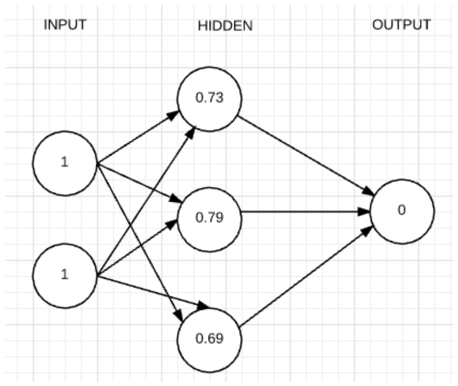


Figure 3.1 Typical neural network

The basic component of a neural network is the node. It generates the output value for a given input. In the case of a simple feed-forward neural network, the neural net uses those interconnected nodes to process information across its architecture, from input nodes to output nodes. Input nodes are the nodes on the left. They receive information from outside of the network and together form the Input Layer. They are filled by the user of the network. Input Layer nodes usually have one input and multiple outputs. The middle layer of the network is called the 'Hidden Layer'. It has multiple input connections and multiple outputs. It is the layer that processes information received from input nodes. The rightmost layer of neurons is the output layer. It outputs the result of processing the information through the neural network. There can be a different quantity of neurons per layer and a different quantity of hidden layers. Each connection between nodes has a weight, which is multiplied by the value of the neuron from the previous layer, and feeds to the current node as an input. Nodes process this information using special functions called 'activation functions' and generate the output, which, in turn, is multiplied by the weight of the next connection and feeds to the next layer neuron.

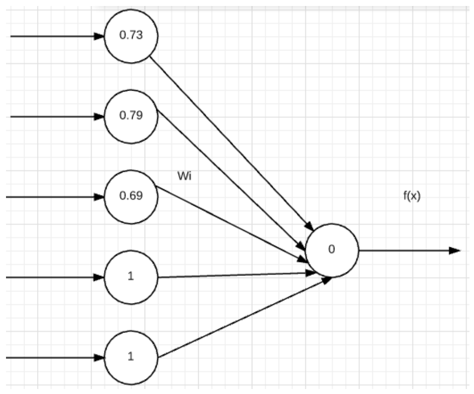


Figure 3.2 Perceptron with one output node

The simplest neural network is a single-layer perceptron (Figure 3.2). It is composed of a single layer of output nodes. The inputs are directly transferred to outputs. The sum of inputs is multiplied by weights calculated for each output node. If the resulting sum is close to some threshold in a particular node, then it will be activated and assigned a value (typically 1), otherwise it will be deactivated (set to 0). This is an example of a linear threshold unit. The comparison with the threshold is done by an activation function called the 'threshold activation function'. Different functions can be used, depending on the data/task that needs to be processed.

Neural networks used in machine learning consist of multiple layers of nodes and are called 'multi-layer perceptrons'. They are usually feed-forward. This means that they have direct connections from the input layer to hidden and from hidden to the output layer. This project uses artificial neural networks for approximation purposes. According to the universal approximation theorem, every continuous function of n real numbers can be approximated by a multi-layer perceptron with one hidden layer.

Within the literature, one of the most popular algorithms for learning a multi-layer neural network is back-propagation.

3.3.3.1 Back-propagation algorithm

The back-propagation algorithm is used in order to learn a multi-layer neural network.

Let us consider a three-layer neural network – input, hidden and output – consisting of two neurons each (Figure 3.3). Each layer, except the input layer, has one neuron connected to all the other neurons in the layer. These are called 'bias neurons'. They do not have any incoming connections, they have value 1 and are used to help make network learning smoother.

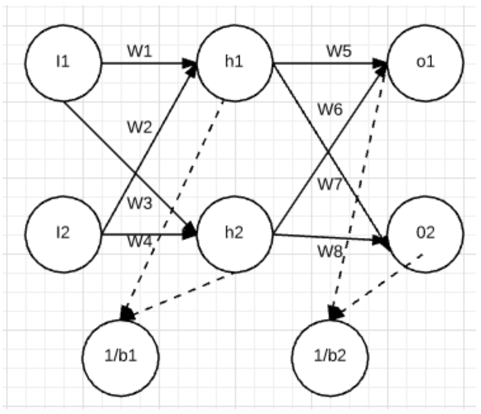


Figure 3.3 Three layer neural network

The main idea of the back-propagation learning algorithm is to change the weights of the neural network (NN) in order to make it map inputs to outputs with minimal error.

At first, an NN does a forward pass of the information fed into input units through its architecture. It calculates the value for each node of the next layer starting with the hidden one. For example, h1 will be calculated as:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

where w_1, w_2 are connection weights and i_1 and i_2 are input unit values.

Next, the activation function of the node needs to be calculated in the following way (activation function can be different):

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}}$$

This approach is repeated for all neurons in the layer. After all layer neurons have been calculated, the procedure is repeated for the next layer in the following way:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}}$$

Knowing the values for all output neurons, the total error for every output node can be calculated using the squared error function:

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

where ‘target’ is a value that the neural network is supposed to learn related to current inputs.

In the case of two output neurons, total error will be the sum of total errors of every node. This is called ‘forward pass’.

The next stage of the algorithm is to correct the weights by propagating the error back to the input nodes.

Weights should be updated in the direction of output layer to input layer. For every output weight, a partial derivative needs to be calculated in order to know the increment of correction. For example, for weight 5, the increment will be:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{h1}}{\partial net_{01}} * \frac{\partial net_{o1}}{\partial w_5}$$

In a similar manner, the correction for weight 1, for example, will be:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

The remaining weights should be calculated in a similar way. The update of the weight is done with the following formula:

$$w_5^{+} = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5}$$

where η is learning rate. This parameter sets how big the correction will be each time the error is propagated.

Therefore, after some iterations, back-propagation will correct the weights of the neural network and it will be mapping a given input to the given output without any error.

In this project, a neural network is used to approximate the Q-function in the Q-learning algorithm. As a target value, the network will receive a real utility calculated in the current state for the current action, using an environment reward value. As an input, it receives the parameters describing the state/action pair.

Another way to approximate Q-learning is to use an artificial neural network with a different architecture: the Elman Recurrent Neural Network (RNN).

3.3.3.2 Elman Recurrent Neural Network

A useful characteristic of RNN architecture is the fact that hidden layer neurons have recurrent connections (Figure 3.4). They feed the values obtained in a previous processing step back to the neurons of the hidden layer, merging them with present values. This property is considered a ‘memory’ of the NN and gives the opportunity to learn a sequence of actions. This project makes an audit system learn a sequence of actions, leading to successful security penetration. That is the main reason for using the Elman RNN as an approximator, as an alternative to a traditional ANN.

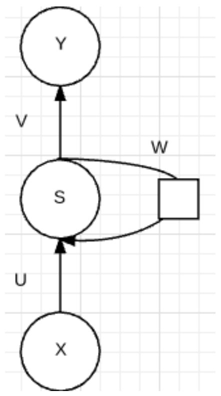


Figure 3.4 Elman RNN architecture

Where x is input at the time step t. S is hidden layer node with a looped connection to itself and y is an output node. U, V, W are connection weights.

The difference in architecture makes the learning process of such neural networks a bit different. It is still based on the back-propagation algorithm, however it is modified and called 'back-propagation through time' (BPTT).

3.3.3.3 Back-propagation through time algorithm

In order to be able to apply back-propagation to RNN, it can be unfolded in the way shown in Figure *3.5*.

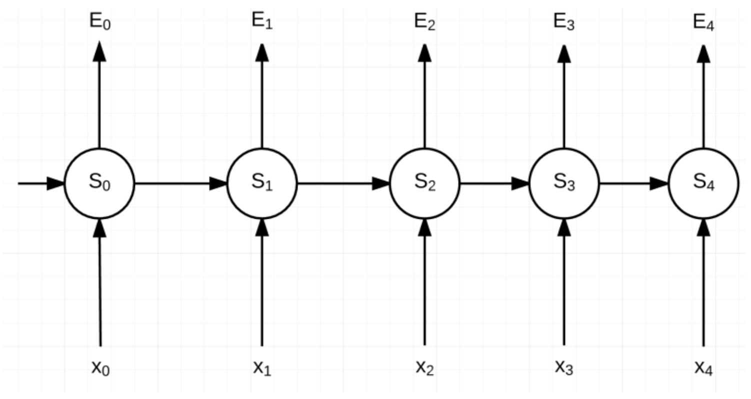


Figure 3.5 Unfolded RNN

The idea of learning in BPTT is the same: gradient error calculation with respect to weights U, V, W and further weights correction. After a forward pass, backward propagation of the error starts. According to the analogy with summation of the output layer error in traditional BP, gradients at each time step for one training example are summed up:

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

In order to calculate the sum, we use the chain rule of differentiation. That is exactly what was used during backward propagation of the error in traditional BP. For example, for E_3 the formula will be:

$$\frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y_3}} \frac{\partial \hat{y_3}}{\partial V}$$

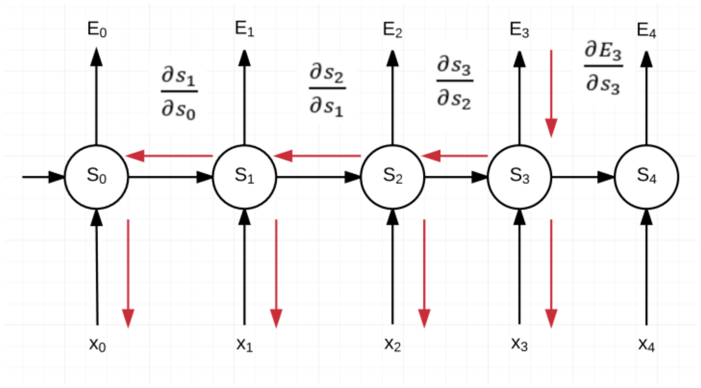
$$= \frac{\partial E_3}{\partial \hat{y_3} \partial z_3} \frac{\partial \hat{y_3} \partial z_3}{\partial V}$$

$$= (\hat{y_3} - y_3) \oplus s_3$$

where $z_3 = V s_3$

For the weight W, the gradient will be (Figure 3.6):

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y_3}} \frac{\partial \hat{y_3}}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$



E0
E1
E2
E3
E4
$\frac{\partial s_1}{\partial s_0}$
$\frac{\partial s_2}{\partial s_1}$
$\frac{\partial s_3}{\partial s_2}$
$\frac{\partial E_3}{\partial s_3}$
S0
S1
S2
S3
S4
X0
X1
X2
X3
X4

Figure 3.6 Gradient calculation for weight *W*

It should be highlighted that the principle behind BPTT is exactly the same as in the traditional BP case. The only difference is in the summation of gradients for the loop connection weight on every time step.

The state of a complicated environment, such as a computer host (where the penetration testing system acts), is usually described by a set of features. If the features are defined and their values have been obtained, then they can be formed in the vector. The chosen action in that current state can also be formed in the vector joint with the 'state' vector. This joint vector can be fed to the neural network as input, one element to one input node. As a result, the network will calculate the Q-value for a given pair.

Features describing state and used as a neural network input can be chosen by the researcher. However, the features space might be too large. In this case, the quantity of neurons will be increased as well. That will lead to performance problems. In addition, it might not be clear what features are really needed. In order to address this problem, the other type of neural network can be used – an autoencoder. This has the ability to find correlations between input data that can be used to decrease feature space.

3.3.3.4 Autoencoder

An 'autoencoder' is a particular architecture of ANN consisting of an input layer, hidden layer and output layer (Figure 3.7).

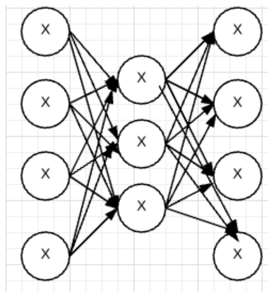


Figure 3.7 Autoencoder architecture

The input (x) and output (x') layer have the same size. The main advantage of an autoencoder is the ability to use unsupervised learning. Its main principle of operation and learning consists of adjusting the internal weights of ANN to recreate the input as an output with minimal error. This property can be used to reduce the input layer to the smaller hidden layer with minimal information loss. As a result, the autoencoder learns to extract the correlation of input data automatically and maps it in a smaller space. The learning process of the autoencoder uses the traditional back-propagation algorithm (Table 3.1).

Table 3.1 Learning process of autoencoder

For each input X	
1.	Do a feed forward pass to compute activations at all hidden layers
2.	Compute activations at the output layer to obtain an output X`
3.	Measure the deviation of X` from the input X (square error used here)
4.	Back-propagate the error through the net and perform weight updates
end for	

The main function of the autoencoder is a reconstruction of its input X. It consists of encoder and decoder parts. The encoder is a transition Φ and the decoder is a transition ψ :

$$\Phi : \mathrm{X} \rightarrow \mathrm{F}$$

$$\phi : \mathrm{F} \rightarrow \mathrm{X}$$

$$arg \min_{\Phi, \phi} \left\| X - (\Phi \circ \phi) X \right\|^2$$

In the case of only one hidden layer, the autoencoder maps its input:

$$x \in R^d = X_{into} z \in R^p = F : z = \sigma_1(Wx+)$$

where σ_1 is an activation function for each element. The experiments here used sigmoid, tangent and rectified linear unit. After z was calculated, it is reconstructed into X`:

$$x' = \sigma_2(W'z + b')$$

During the autoencoder training process, autoencoder minimizes reconstruction errors. Square error was used here:

$$\mathcal{L}(x, x') = \|x - x'\|^2 = \left\|x - \sigma_2(W'(\sigma_1(Wx + b)) + b')\right\|^2$$

This project uses the autoencoder to decrease the features space representing the target host.

The penetration testing system consists of two agents communicating via a network. One attacks the target and the second resides there to give environmental feedback for the first agent. The second agent collects the set of features to define the state space. The state space consists of features describing the target host and its current status: hacked (admin privileges), hacked (ordinary user privileges), attack failed, OS checked. Those flag-type features are combined with the features describing the host from the software point of view. This is related to the fact that most exploits compromise the particular service/process. Therefore, features will be able to indicate the vulnerable process presence on the host. It will give an opportunity for the learning algorithm to correlate information about sets of processes and successful exploits. The features vector is created in the following way:

- The target host gets a list of processes and their dll modules.
- The unique list of dll modules is created.
- Every process is checked against this list to form a vector of 0s and 1s, showing the presence/absence of each dll.
- The result of extraction is saved as a vector of 0s (dll module is presented) and 1s (dll module is not presented), describing the state of the target host (Figure 3.8).

'Hacked/not hacked' flag features are not yet added to the features vector. This process will happen after the features set is reduced.

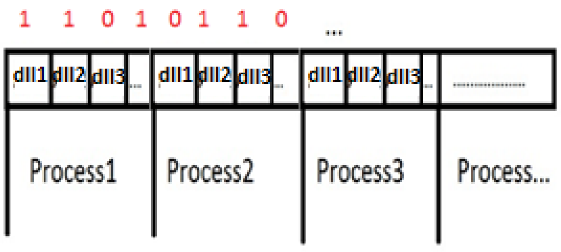


Figure 3.8 Feature extraction result

During the first attempts at modelling, the host configuration API functions were included instead of dlls. Although the APIs list better represents host functionality, it is very complicated to derive such a list and it is almost impossible to get a full list of APIs used in a particular system. In contrast, the list of dlls for every process can be easily obtained.

The resulting vector is an input for the autoencoder. It is a feature space vector and the goal is to reduce it. The learning process of the autoencoder starts. It repeats a number of iterations. During those iterations, the autoencoder continuously encodes and decodes the inputs to minimize the difference between input and output. Output is the recovered input. The autoencoder can be connected with another, which gives the opportunity to use deeper, more effective architecture (Figure 3.9).

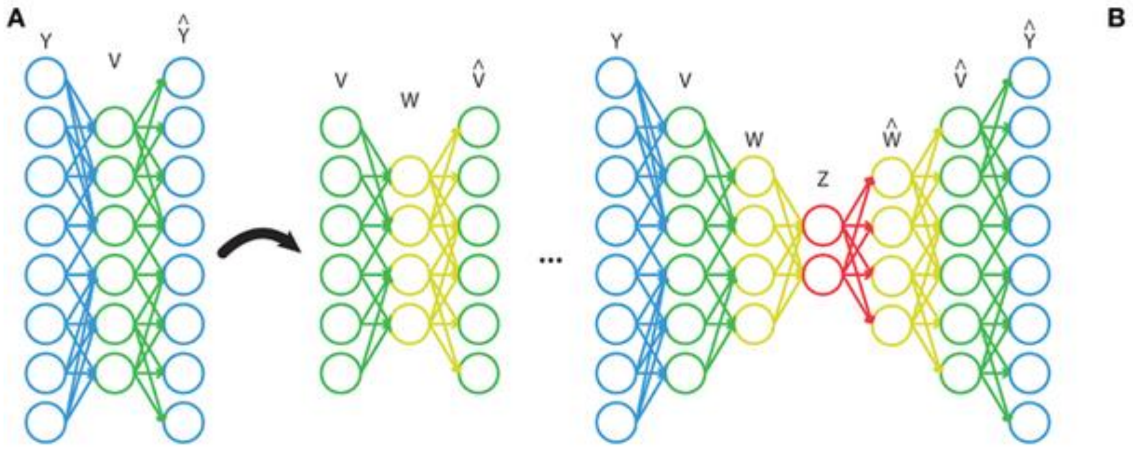


Figure 3.9 Deep autoencoder architecture

The experiment was performed without a second autoencoder layer. More details are presented in the Results section. In case a second layer of autoencoder is used, then the process of learning repeats for the second autoencoder as well. The only difference is that the input for the second autoencoder is the value of hidden layer neurons of the first. This can slow down the learning process in general. However, the second autoencoder learns much faster. An example of the learned autoencoder is presented in Figure 3.10. Autoencoder is used for feature extraction before actual penetration tests. It needed for agent to be able to describe a system under attack and interpret its feedback after penetration testing phase started.

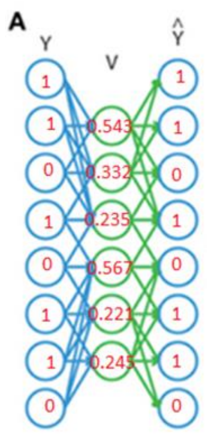
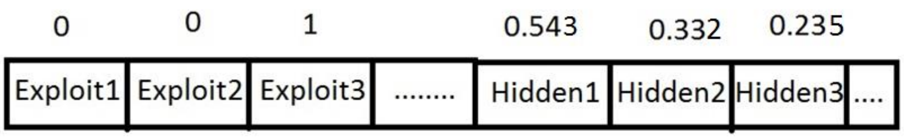


Figure 3.10 Learned autoencoder

After the autoencoder learning process is finished, the audit system derives a vector of its hidden layer values and adds 'hacked/not hacked' flag features. It is the vector representing the new decreased features set, describing a state of the target host. This vector, combined with the action vector, will be fed to the Elman RNN approximator in order to determine the Q-function value. Therefore, a deep learning architecture is achieved, consisting of two neural networks combined together: the autoencoder to decrease the quantity of features describing state space and the Elman RNN used as an approximator in Q-learning.

Further, the program chooses the action according to the Q-learning policy. The policy is an ε-greedy one. It means that the program sometimes acts randomly, and sometimes according to its experience (Q-value). Acting according to experience is described below. Let us consider when there are three possible actions to choose from: execute exploit1, exploit2 or exploit3.

Let us assume that, during its first iteration, the program acted randomly. In this case, the program has chosen exploit3. The vector shown in Figure 3.11 will be created.



0	0	1		0.543	0.332	0.235	
Exploit1	Exploit2	Exploit3	……..	Hidden1	Hidden2	Hidden3	….

Figure 3.11 The action vector

This vector is the input to the RNN. After the RNN processes this input, it will calculate the Q-value for that particular state-action pair. The Q-value is a prediction of how good this particular action is in this particular state from the point of view of reward accumulation. At the same time, the program will execute exploit3 and will receive the real reward, depending on how successfully exploit3 was executed. The error between Q-value and real reward will be calculated and this error will be propagated back, changing the weights in the RNN.

After exploit3 has been successfully executed, the system will move to another state: the flag feature 'hacked (admin privileges)' or 'hacked (user privileges)' will become 1, depending on the exploit3 result. The vector representing state will be changed. The feature vector describing the host will remain the same in that case, so the autoencoder will have no need to learn new features. It is very effective because the learning of the autoencoder takes time, which cannot be wasted after every action. However, if the action resulted in changing the target, the feature vector built by the autoencoder will be updated using the new target. This approach to the definition of 'state space' helps to include information not only on the status of a particular host, but also the attack trajectory through the network. In this project, all experiments were performed without host transitions (this is a possible direction for the future).

As a result of transition to another target, the vector which describes the new state of the host will be created, as indicated in Figure 3.12.

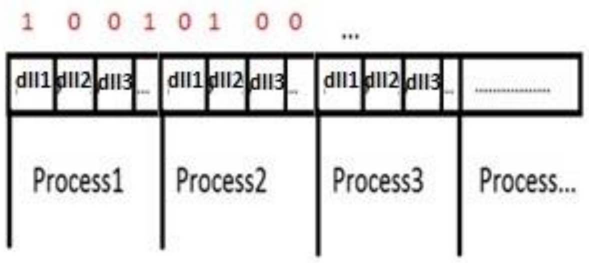


Figure 3.12 The new state of the host

This vector will be an input to the autoencoder to reduce features. The autoencoder will be relearned (Figure 3.13).

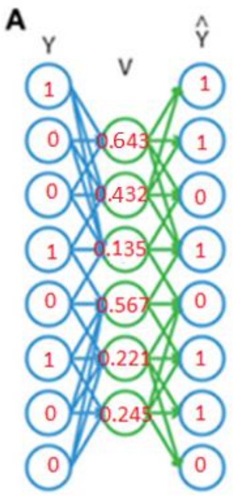


Figure 3.13 Relearned autoencoder

After the autoencoder has been relearned, the program will derive a vector of its hidden layer values. It is the vector describing a state. Now the program will choose the action according to the ε-greedy policy. Let us consider the case where it acts according to its experience (Q-value). The program has three actions: exploit1, exploit2 or exploit3. First of all, the program will calculate Q-value1 for the first action: exploit1. It will form the action vector '1 0 0' and will combine it with the vector from the autoencoder and status flag features, which describes the state (Figure 3.14).

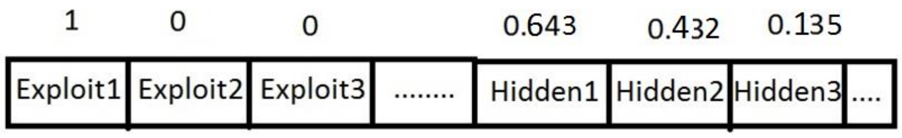


Figure 3.14 Action vector 1

This vector will be an input for the RNN. As a result, the Q-value1 for action1 (execute exploit1) will be received. It should be mentioned that this is just a calculation of Q-value: there is no error back-propagation yet.

Next, the Q-value2 for action2 (execute exploit2) will be calculated. The program will form the action vector '0 1 0' and will combine it with the vector from the autoencoder plus status flag features, which describes the state (Figure 3.15).

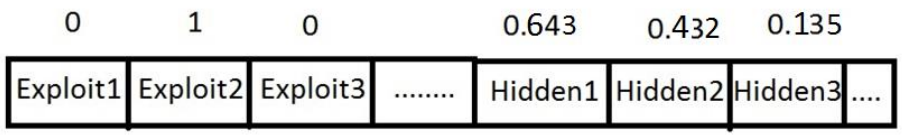


Figure 3.15 Action vector 2

This vector will be an input for RNN. As a result, the Q-value2 for action2 (execute exploit2) will be received.

In the same way, the Q-value3 for action3 (execute exploit3) will be calculated. The program will form the action vector '0 0 1' and will combine it with the vector from the autoencoder with host status flags, which describes the state (Figure 3.16).

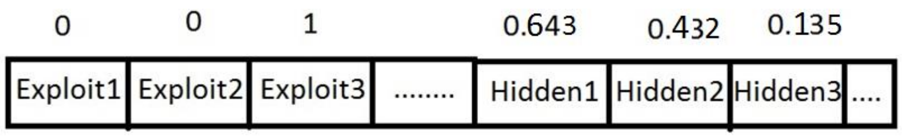


Figure 3.16 Action vector 3

This vector will be an input for RNN. As a result, the Q-value3 for action3 (execute exploit3) will be received.

The program will compare Q-value1, Q-value2 and Q-value3, and will choose the max. Let us consider the case when it is Q-value1.

In this case, the program will execute exploit1. It means that the vector shown in Figure 3.17) will be an input for the RNN.

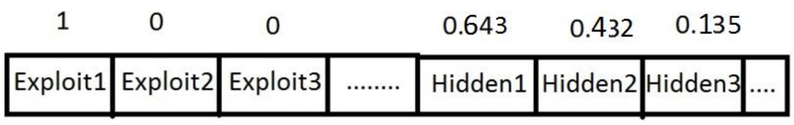


Figure 3.17 Final action vector

It should be mentioned that the action/pair vector (execute exploit3) from the first iteration will already be saved in the memory of the RNN (Figure 3.18).

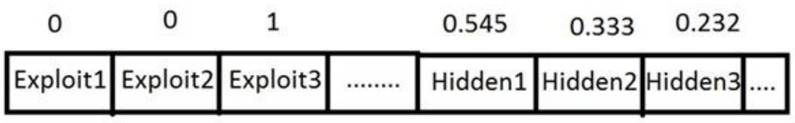


Figure 3.18 Previous action vector

This means that the actual input for the RNN will not only be the vector that was chosen during the second iteration, but the combined vectors from the first and second iterations. It is a representation of the sequence of actions.

The program will receive the real reward, depending on how the exploit1 worked. Finally, the error between Q-value1 and real reward will be calculated and this error will be propagated back, changing the weights in the RNN. The state of the host will be changed as a result.

This process will continue according to the Q-learning algorithm until the learning process has ended.

3.4 Environment architecture

In order to test an application of the developed approach to the real systems, the virtual environment was build. It replicates a typical corporate network. The only difference with real system is in hosts, represented as virtual units, instead of having their own hardware. Hosts run using resources of two real joint clusters. This fact does not influence on the experimentation results. Therefore, this configuration can be considered as real system testing environment. The system architecture includes 80+ hosts and represents a typical corporate network prototype, consisting of four parts: entry points, attacker segment, demilitarized zone (DMZ) and user segment (Figure 3.19). The communication between zones is controlled by firewalls and IDS. The main purpose of the entry points segment is to isolate the prototype and the City University network. These hosts have Windows 7 OS and are defended by Windows firewall to block all traffic going outside the network. Entry points can be used as an attacker host as well. The attacker and entry points segments are the hostile environment in terms of computer security. They represent segments from which the corporate network can be attacked. Attacker segment hosts have Kali Linux OS installed. It has a variety of computer security audit tools such as: Aircrack-ng, Burp suite, Cisco Global Exploiter, Ettercap, John the Ripper, Kismet, Maltego, Metasploit framework, Nmap, OWASP ZAP, Social engineering tools, Wireshark, Hydra, Reverse Engineering tools, Forensics tools like Binwalk, Foremost, Volatility etc. The next segment is the DMZ. It was set up in order to defend services, which need to be accessed from user and attacker network segments. The demilitarized zone hosts provide the basic network services: corporate mail server, file server, web server, SQL server, SSH server and so on. The connection between segments is defended using IPFW firewall and SNORT IDS. This software filters all internal and external traffic and raises the alarm in case of suspicious behavior. The main purpose of the DMZ is the creation of the primary barrier against malicious attacks on the User segment. Mail and web servers have been implemented using Courier and Apache, based on the OS FreeBSD. The most protected segment of the prototype is the segment of end-users. It is protected against external intrusions by two firewalls and the DMZ. End-users have access to the services presented in the DMZ. They can securely communicate with each other.

The infrastructure is built using virtual machines based on VMware technology. Physically, all virtual machines are stored on two clusters under VMSphere management.

The servers in DMZ zone and firewalls were not used in experiments. They were incorporated in the testing environment in order to build a network as close to reality as possible. Those hosts will be used for the future experimentation including penetration agent detection and server attacks.

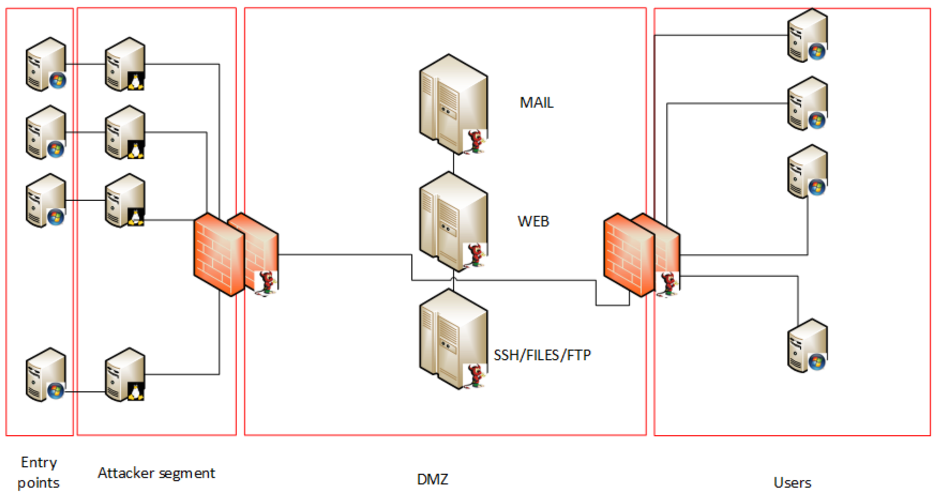


Figure 3.19 Physical scheme

3.5 Tool architecture

The system architecture consists of a client and a server (Figure 3.20). The server is installed on a host within the same network as the target. The target is the host to be attacked. During the computer security audit, the server performs multistep attacks by generating a sequence of actions and executing them, one by one, in a single attack. Actions include exploits and discovery actions such as port scan and OS detection. The server executes remote exploits itself. However, for local exploits, it sends the task for execution to the client, residing on the target. This functionality makes possible the scenario where a local exploit will be executed immediately after the remote one. After the server attacks a target, the client receives the notification that the attack was executed. The client checks the success of the exploit execution and then sends the result back to the server. It also restarts the services if needed. In the approach used here, all decisions concerned with the combination of exploits and the generation of the vulnerable string are learned using Q-learning on the server side. The client generates the reward signal. It should be noted that the client works using admin rights on the target machine. Such an architecture solution prevents the usage of the tool for malicious purposes. It will not let the operator of the tool break into anyone's system, except the one he owns. The client is installed on the target and provides environmental feedback to the Q-learning algorithm deployed on the server. This indicates whether the exploit has breached the target's security and updates a reward signal according to the actions of the server. The architecture assumes that the feedback agent can be installed on the target host.

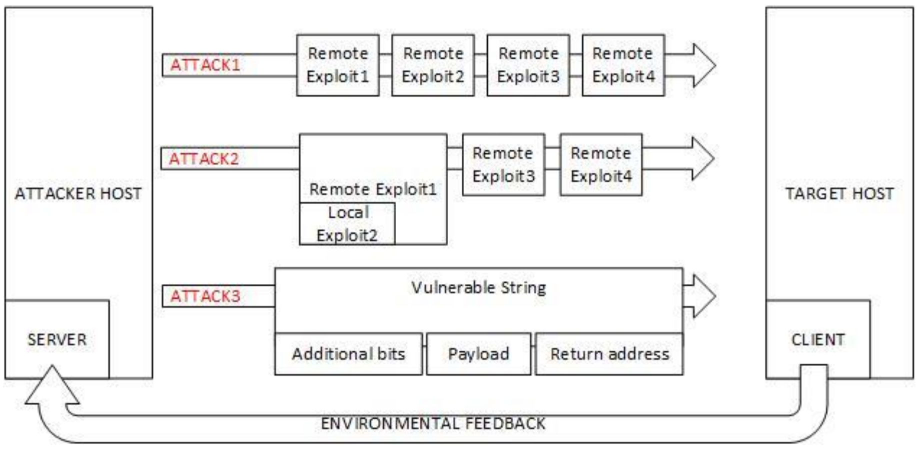


Figure 3.20 System architecture

3.6 Experiments

The experiments were intended to prove that Q-learning could be successfully applied to the penetration testing process in order to make it learn the attack strategy itself. This application was based on a model free approach. The penetration testing system did not have a model of the environment it was acting in. It was just executing actions and was learning the successful attack strategy by trial and error. All experiments involved an attacking agent and an environment agent. The attacking agent represented the computer security audit system. It performed attacks on the remote targets. The environment agent resided on the target in order to provide environmental feedback to the attacking agent. Based on this feedback, the agent accumulated knowledge and learned the optimal attack strategy. Experiments were divided into five groups: MDP experiment, Field experiment, New vulnerabilities experiment, Deep Architecture ANN experiment, Deep architecture RNN experiment.

In case of an MDP process, a model is described (apart of the state/action space) by definition of transitions between states and predefined rewards for every action in a particular state. This approach is called Model-dependent. In cases when the agent (acting in MDP state space) doesn't know transitions between states and predefined reward for its actions, it learns the model on the go by trial and error. This approach is named model free approach. Therefore, in order to apply the model free approach, two requirements have been relaxed: predefined reward function and predefined transition function for all possible states.

The first group of experiments was intended to test the basic hypothesis of the model free approach, that the penetration testing system based on the Q-learning algorithm would be able to learn the attack strategy itself without a predefined reward function. The relaxation of this requirement leads to the receiving of rewards directly from the environment. Since experiments were designed to test the basic concept, the penetration testing process was defined as a specific MDP with state and action spaces. The case was generalized later in Deep Architecture ANN experiment and Deep architecture RNN experiment. The transition function was defined as well at that stage of the experiment. The agent attacked the target and learned the optimal attack strategy. In order to check the performance of the algorithm, different hyperparameters were tested such as: learning rate, reward discount parameter, quantity of episodes. Different acting policies were compared as well: random, ε-greedy, softmax policies. The dynamics of convergence to the optimal attack strategy was graphically assessed. More information about those experiments can be found in the MDP experiment section.

The second group of experiments was also related to the investigation of the model free approach. Additionally to the relaxation of the predefined reward function, the transition function definition requirement was relaxed as well. The penetration testing system was attacking the target while knowing the state/action space definitions only. Rewards were derived from the environment. The transition function was not predefined. As in the previous case, the hyper-parameters performance for the Q-learning algorithm was tested, such as learning rate, reward discount factor and different policies. The criteria of optimality was compared graphically. More details can be found in the Field experiment section.

The third group of experiments was intended to prove that the Q-learning model-free approach could be successfully applied to the new vulnerabilities identification, the buffer overflow in particular. The attacking agent was sending different inputs to the remote target machine's service waiting for the input. The network service had a predefined buffer overflow vulnerability. The goal of the penetration testing system was to learn the sequence of actions leading to the successful vulnerability exploitation. The experiment was based on the particular MDP. The reward and transition function were not predefined. The performance of the algorithm was tested according to different hyperparameters: learning rate, reward discount parameter, quantity of episodes. Different acting policies were compared as well: random, ε-greedy, softmax policies. The generalisation of the experiment is considered as future work. More information can be found in the New vulnerabilities experiment section.

The goal of the fourth group of experiments was to generalise the Q-learning model free approach for real world scenarios with huge state spaces. The algorithm was modified by using deep artificial neural network architecture as a state space approximator. It consisted of an autoencoder joined with an artificial neuro network (ANN). The autoencoder needed for the target host features extraction. ANN was used for actual approximation of the learning process. Due to deep architecture, the penetration testing system received the opportunity not only to learn the attack strategy while acting, but to extend the accumulated knowledge to the cases it never experienced before. The experiments were divided in two phases.

The first phase consisted of attacking a remote target to learn the successful attack strategy. During this phase, the optimal hyperparameters for the Q-learning algorithm were used, identified during the MDP experiment and the Field experiment, such as learning rate and reward discount factor. However, additional approximator architecture related hyperparameters were tested in order to check penetration testing learning performance. The autoencoder was tested with different values of weight decay, quantity of iterations, neurons per hidden layer, sparsity penalty term. The ANN hyperparameters were tested as well, such as: quantity of iterations, weight decay, learning rate. Different acting polices were used: random, ε-greedy and softmax.

During the second phase, the penetration testing system was executed against a remote host it had never experienced in the past. This experiment was intended to prove the ability to approximate the previously accumulated knowledge for the real case scenario with huge state space. More information can be found in the Deep Architecture ANN experiment section.

The fifth group of experiments was intended to test the adaptivity of the penetration testing architecture and better approximator functionality. A model free approach based on Q-learning application was used. The reward and transition functions were not predefined for every state and were learned while acting. The deep architecture of the approximator was changed. ANN was substituted by a recurrent neural network (RNN), an Elman network in particular.

Experiments consisted of two phases. During the first phase, the penetration testing system was learning the attack strategy by attacking the remote target. The optimal hyperparameters were identified during previous experiments. Different action strategies were used: random, ε-greedy, softmax policies.

During the second phase, the penetration testing system was attacking the target, which had not been experienced earlier, using previously accumulated knowledge. After successful penetration, the case was emulated when the configuration of the target was changed (one of the vulnerabilities was patched). In other words, the attack strategy, learned earlier, was no longer leading to successful penetration. In those circumstances, the penetration testing system should have been relearned automatically to find the new successful attack strategy itself. More information can be found in the Deep architecture RNN experiment section.

The next chapter represents the information on the full set of experiments performed in order to achieve the goals of the project.

4 Results

This section describes the experimental results, including Q-learning policy performance and learned attack strategies. The interpretation of the results of the experimentation is presented in the Conclusion and future chapter. The outline of the experiments is described in the Experiments section of the Methodology chapter.

This research on computer security, specifically on the audit process, is related to the application of Q-learning algorithms in order to make a penetration testing system learn attack strategies and adapt to the changing environment. The attacking agent performs different attack sequences in order to find the vulnerabilities of the target system. It receives feedback from the environment and corrects its attack strategy accordingly. As a result, accumulated knowledge provides the ability to make decisions during the penetration testing process automatically. This methodology allows the attack agent to learn new attack strategies if the host configuration under attack is changed.

First, the computer security audit was modelled as a Markov Decision Process (MDP) in order to test a number of decision-making strategies and to compare their optimality. This model will be able to describe a behavior of the penetration testing system. It also can be tuned up to deal with real-life penetration testing constraints such as: attack detection rate, level of network traffic etc. In order to do so, additional features, defining state of the system can be added. In current research, exploit properties only were considered as constrains for the decision making such as: level of harm for system under attack, date of release, level of privileges etc.

Second, a novel decision-making approach to the audit process was used, based on a modified Q-learning application, which was implemented with three different policies: Q-learning random policy, Q-learning ε-greedy policy and Q-learning softmax policy. At the end, the optimality of these policies was compared.

Third, a deep learning architecture for the approximator was developed in order to generalize the implemented approach for the real-life scenario. The proof-of-concept tool and testing architecture was built.

The following subsections describe the experimental hypothesis, setup, result and other related information.

4.1 MDP experiments

4.1.1 Hypothesis

This group of experiments is intended to test the hypothesis that the Q-learning algorithm can be used to make a penetration testing system learn attack strategies by itself on the go, without a predefined reward function. The agent derives the reward from the environmental feedback directly. Additionally, it will show that the acting agent will be able to learn multistep attacks.

4.1.2 Experiment setup

The experiment uses the testing infrastructure described in the Environment architecture section of the Methodology chapter (Figure 4.1)

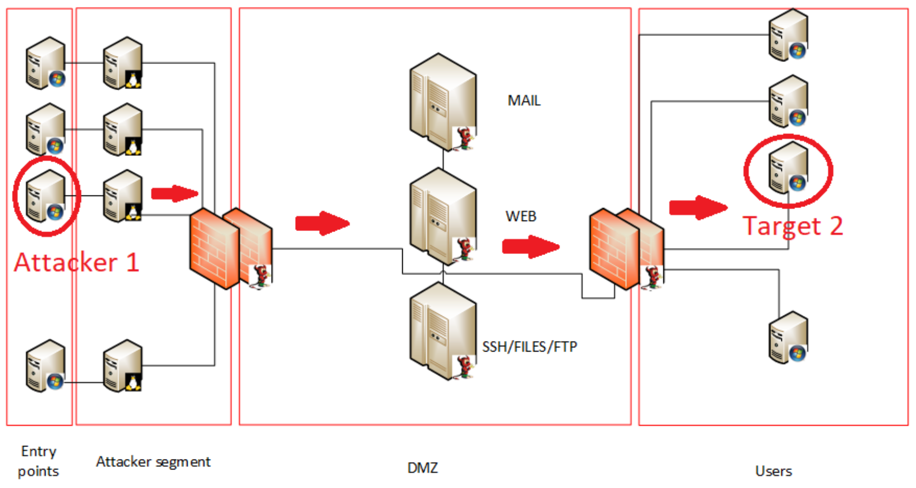


Figure 4.1: The experiment setup

The first host represents the penetration testing system. It has the residing agent program with the capability to attack the target. By ‘attack’, we mean the ability to execute exploits and probe the target (check the OS, scan ports, etc.). The second host is the target under attack. It has the second agent residing (on Target 2), which will give environmental feedback to the penetration testing agent, such as: exploit worked successfully with admin rights; exploit worked successfully with user rights; exploit failed; returns the OS name; particular port is open; particular service is presented. Both agents communicate via sockets. The target host has remote code execution and privilege escalation vulnerabilities.

During an experiment, the penetration testing agent attacks the target host by executing different actions – exploits and probe actions. The target agent sends back environmental feedback. Based on that process, the attacking agent's machine learning algorithm learns the optimal sequence of actions (attack strategy).

After hypothesis validation, the experiments related to learning performance were realized by changing different hyper-parameters of the learning algorithm. First, the quantity of episodes was changed. Second, the values of learning rate and discount factor were tested.

In order to have a learning ability, the machine learning algorithm should know what process it is interacting with. For that purpose, the penetration testing needs to be modelled as an MDP. This MDP is not universal and describes only that particular experimental situation. The goal of the experiment was to validate proof-of-concept of the idea, stated in the hypothesis – the generalization problem was addressed in the later Deep Architecture ANN experiment by applying Q-learning approximators.

In order to make the penetration testing system adaptive, a model-free approach was used. It does not have a predefined model of the environment. As opposed to the model-based approach, the requirement for transition and reward functions can be withdrawn. The current experiment is intended to test the hypothesis without a predefined reward function, describing all possible rewards for all actions in all states. Rewards are received directly from the environmental feedback as a result of action. Transition function requirement is not withdrawn for the current experiment. It is defined as a table describing every state-action pair transition. This requirement is later withdrawn in the Field experiment.

Actions, transition, states and MDP itself are presented in the next subsection, on the 'The MDP process'.

4.1.3 The MDP process

The penetration testing process was modelled as an MDP with states/ action defined below. Only exploits presented in the table were used during experiment.

The list of states and the list of actions are presented in Table 4.1 and Table 4.2, respectively.

Table 4.1 MDP states

0.	Initial state – The state from which the audit starts
1.	Win7 – The state where it is checked whether the host has a Windows OS

2.	Lx – A state when it is checked whether the host has a Linux OS
3.	P445 – A state when it is checked whether the host has Port 445 open
4.	P135 – A state when it is known that the host has Port 135 open
5.	P22 – A state when it is known that the host has Port 22 open
6.	P2525 – A state when it is known that the host has Port 2525 open
7.	Serv1 – Checked that Service1 exists on the host
8.	Serv2 – Checked that Service2 exists on the host
9.	SSH – Checked that SSH service exists on the host
10.	Rm_Ex_1 – Remote Exploit 1 (admin) executed
11.	Rm_Ex_2 – Remote Exploit 2 (user) executed
12.	Lc_Ex_3 – Local Exploit 3 executed
13.	SSH_prob - SSH probing
14.	Rm_Ex_1_2 – Remote Exploit 2 (admin) executed as a second exploit in the attack chain
15.	Rm_Ex_2_2 – Remote Exploit 1 (user) executed as a second exploit in the attack chain
16.	Lc_Ex_3_2 – Local Exploit 3 executed as a second exploit in the attack chain
17.	Fail_terminal – Final state – dead end, such as exploit failed
18.	Suc_terminal – Final state, target was reached, such as host hacked (admin privileges on the host)

Table 4.2 MDP actions

0.	OS_Win_Check – Checks if OS is Windows 7
1.	OS_Lin_Check – Checks if OS is Linux
2.	Port445_Check – Checks if Port 445 is open
3.	Port135_Check – Checks if Port 135 is open
4.	Port22_Check – Checks if Port 22 is open
5.	Port2525_Check – Checks if Port 2525 is open

6.	Serv1_Check – Checks if Service1 works
7.	Serv2_Check – Checks if Service2 works
8.	SSH_Check – Checks if SSH service works
9.	Rm_Ex_1_Check – Execute remote Exploit 1 (admin)
10.	Rm_Ex_2_Check – Execute remote Exploit 2 (user)
11.	Lc_Ex_3_Check – Execute Local exploit 3

The example of MDP transactions from the initial state is presented in Figure 4.2. The bigger size of the figure is presented in Appendix A.

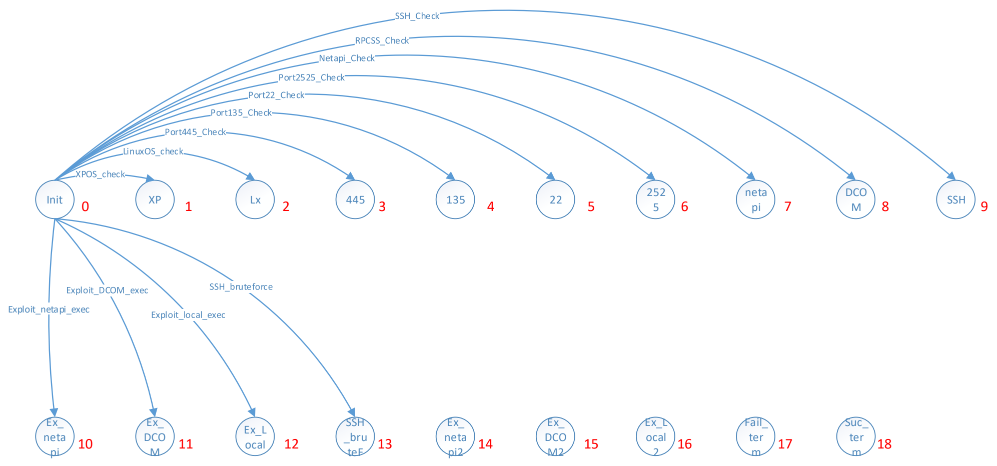


Figure 4.2 Transitions from initial state

Actions and states are defined as numbers 0-18 in the Q-table of the Q-learning algorithm.

4.1.4 Reward

A ‘reward’ is the numerical value received by the agent after an attack/probe action was executed. It is the agent motivator to learn the successful attack strategy. Since the methodology described here is based on a model-free approach, the reward function is not predefined for all possible actions in all possible states. The reward is received directly from the environmental feedback. For example, if the last action in the sequence resulted in successful penetration, the agent will receive a high reward for the last transition to the successful final state. The Q-learning algorithm will propagate this reward to the previous transitions itself. This mechanism will form the agent’s experience, identifying the sequence of transitions leading to the highest possible reward. The task of the agent is to learn the sequence of actions guaranteeing the highest cumulative reward.

The reward is defined as follows:

$$R = R + R1 + R2 + R3$$

where the main reward is defined as R. This reward indicates successful penetration for the agent. In cases where an agent got admin rights on a target host, it receives 100 as a main reward, otherwise it receives 0:

R1 – Represents exploit effectiveness according to the metasploit rating (great, good and so on). The more effective the exploit, the higher the reward.

R2 – Represents the possible OS/service harm after the exploit is used, according to the Common Vulnerabilities and Exposures (CVE) database. The more likely it is that the system service will be crashed, the less reward the audit agent will be awarded.

R3 – Additional reward for testing actions (portscan, OS detection and so on), defined as 20/n (where n is the number of iterations per episode). This reward motivates an agent to explore the environment and reduces the state space before exploitation.

In the current experiment, the agent can choose from three exploits. Remote exploit 2 guarantees user privileges on the target host. Remote exploit 1 guarantees admin privileges on the host. Local exploit 3 escalates the privileges locally from user to admin. Remote exploit 1 is more harmful for the target system and it has a lower reliability rating, according to the CVE. It means that, from the point of view of optimality, it is considered to be less optimal to use. Remote exploit 2 is more reliable and less harmful. It means that the agent receives more reward when it successfully uses remote exploit 2 compared to remote exploit 1. More harmful exploit cases the crash of the system under attack with higher probability. That is the reason why it receives less reward. As the goal of the attack is to get admin access on the Target 2, the agent should not cause DOS for system under attack. The optimal sequence of actions (attack strategy) leading to successful penetration guarantees the highest cumulative reward. In current research, reward does not reflect the value of the target. It will be added in the future work.

4.1.5 Results assessment

The attack agent can learn a number of attack strategies. They are represented by different sequences from the set of available actions. The learning ability of the agent will be demonstrated if any of those successful attack strategies are learned. According to the experiment design, the agent can find a number of available solutions leading to the successful penetration of the target host. Those solutions differ by optimality. It is assessed by comparing the cumulative reward received after the attack strategy has been fully executed. The higher the value, the more optimal the solution. Possible successful attack strategies that can be learned by the agent are presented below. The list is sorted by optimality (1 being the highest):

1. Probe action (Check the OS/ Check Service/ Port scan) + execute a remote exploit with user privileges + execute local exploit for privilege escalation.
2. Execute a remote exploit with user privileges + execute local exploit for privilege escalation.
3. Probe action (Check the OS/ Check Service/ Port scan) + execute a remote exploit with admin privileges.
4. Execute a remote exploit with admin privileges.
5. Sequence of actions leading to Fail state (penetration fail).

Solution 1 will generate the highest cumulative reward as optimal.

The performance of the learning algorithm is measured by cumulative rewards graphically. The higher the graphic curve rises of the cumulative reward across episodes, the better the performance of the learning algorithm. The faster it grows, the faster an acting agent learns.

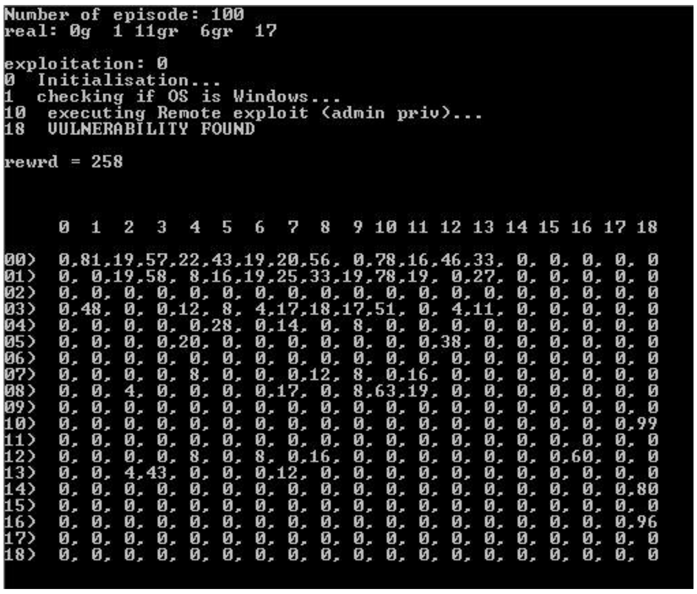
4.1.6 Experiment outcome

This section presents the actual results of the described experimental setup. There were different acting policies used during the learning process:

- Random – Actions are chosen randomly.
- ε-greedy – Actions are chosen randomly or according to experience with 1-ε probability.
- Softmax – Actions are selected randomly but each action has a different probability of being chosen. This probability is calculated according to the experience of the learning agent. The higher the potential reward, the more probable it is that the action will be chosen. In other words, softmax policy selects the actions according to the probability distribution of a number of different possible actions to choose in a particular state.

4.1.6.1 Q-learning with random policy

Initially, Q-learning with random policy was applied. The MDP solution after 100 episodes of Q-learning is presented in Figure 4.3.



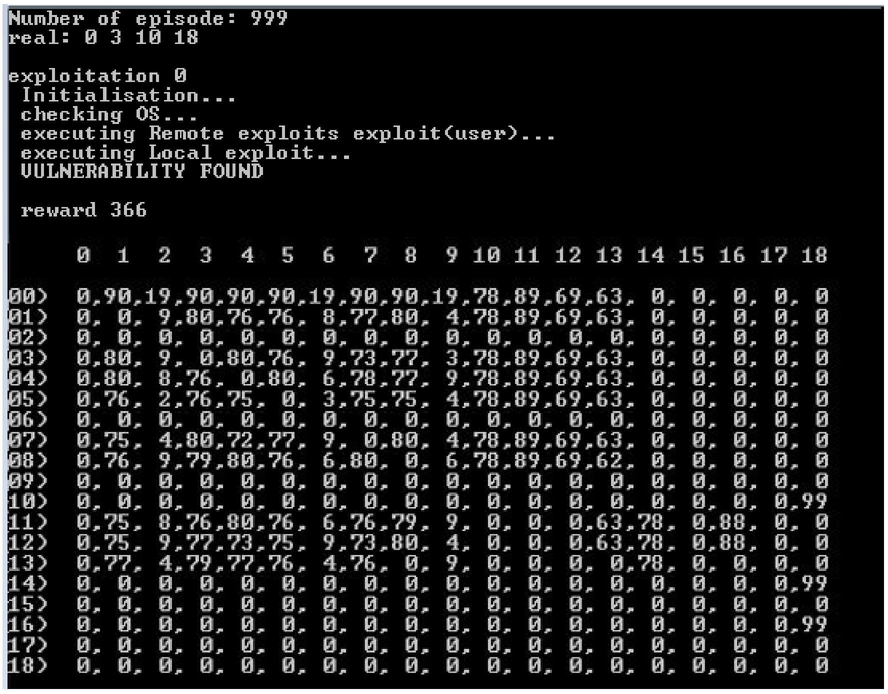
```
Number of episode: 100
real: 0g  1 11gr  6gr  17

exploitation: 0
0  Initialisation...
1  checking if OS is Windows...
10  executing Remote exploit (admin priv)...
18  VULNERABILITY FOUND

rewrd = 258


     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00)  0,81,19,57,22,43,19,20,56, 0,78,16,46,33, 0, 0, 0, 0, 0
01)  0, 0,19,58, 8,16,19,25,33,19,78,19, 0,27, 0, 0, 0, 0, 0
02)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03)  0,48, 0, 0,12, 8, 4,17,18,17,51, 0, 4,11, 0, 0, 0, 0, 0
04)  0, 0, 0, 0, 0,28, 0,14, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0
05)  0, 0, 0, 0,20, 0, 0, 0, 0, 0, 0,38, 0, 0, 0, 0, 0, 0, 0
06)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07)  0, 0, 0, 0, 8, 0, 0, 0,12, 8, 0,16, 0, 0, 0, 0, 0, 0, 0
08)  0, 0, 4, 0, 0, 0, 0,17, 0, 8,63,19, 0, 0, 0, 0, 0, 0, 0
09)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
12)  0, 0, 0, 0, 8, 0, 8, 0,16, 0, 0, 0, 0, 0, 0, 0,60, 0, 0
13)  0, 0, 4,43, 0, 0, 0,12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
14)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,80
15)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,96
17)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.3 MDP solution using random policy Q-learning 100 episodes

This output shows an MDP solution, cumulative reward and Q-matrix with Q-values. The optimality of the solution is defined by the cumulative reward value. The bigger it is, the more optimal the solution that was found. The sequence of steps to trigger vulnerability was revealed:

- Initialization
- Check if OS is Windows
- Execute Remote Exploit 1 (admin) exploit
- Successful vulnerability exploitation

After increasing the quantity of learning episodes to 1000, a new MDP solution was found (Figure 4.4).



```
Number of episode: 999
real: 0 3 10 18

exploitation 0
 Initialisation...
 checking OS...
 executing Remote exploits exploit(user)...
 executing Local exploit...
 VULNERABILITY FOUND

 reward 366

      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00)  0,90,19,90,90,90,19,90,90,19,78,89,69,63, 0, 0, 0, 0, 0
01)  0, 0, 9,80,76,76, 8,77,80, 4,78,89,69,63, 0, 0, 0, 0, 0
02)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03)  0,80, 9, 0,80,76, 9,73,77, 3,78,89,69,63, 0, 0, 0, 0, 0
04)  0,80, 8,76, 0,80, 6,78,77, 9,78,89,69,63, 0, 0, 0, 0, 0
05)  0,76, 2,76,75, 0, 3,75,75, 4,78,89,69,63, 0, 0, 0, 0, 0
06)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07)  0,75, 4,80,72,77, 9, 0,80, 4,78,89,69,63, 0, 0, 0, 0, 0
08)  0,76, 9,79,80,76, 6,80, 0, 6,78,89,69,62, 0, 0, 0, 0, 0
09)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11)  0,75, 8,76,80,76, 6,76,79, 9, 0, 0, 0,63,78, 0,88, 0, 0
12)  0,75, 9,77,73,75, 9,73,80, 4, 0, 0, 0,63,78, 0,88, 0, 0
13)  0,77, 4,79,77,76, 4,76, 0, 9, 0, 0, 0, 0,78, 0, 0, 0, 0
14)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
17)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.4 MDP solution using random policy Q-learning 1000 episodes

The solution to trigger the vulnerability after 1000 episodes is different:

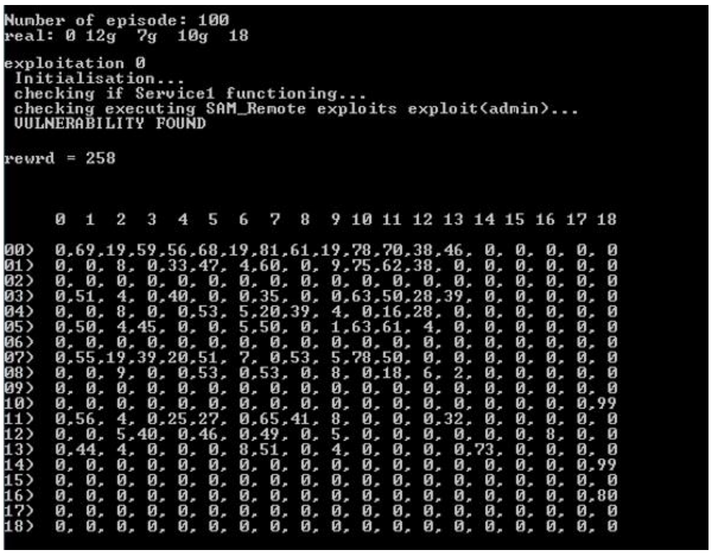
- Initialization
- Check if OS is Windows
- Execute Remote Exploit (user)
- Execute Local Exploit
- Successful vulnerability exploitation

The further increase of episodes did not change this result.

The next experiment was performed for the ε-greedy acting policy.

4.1.6.2 Q-learning with ε-greedy policy

At the next stage of the experiment, Q-learning with ε-greedy policy was applied. The MDP solution after 100 episodes using Gamma=0.8, Alpha=0.8, 1-ε=0.15 is shown in Figure 4.5.



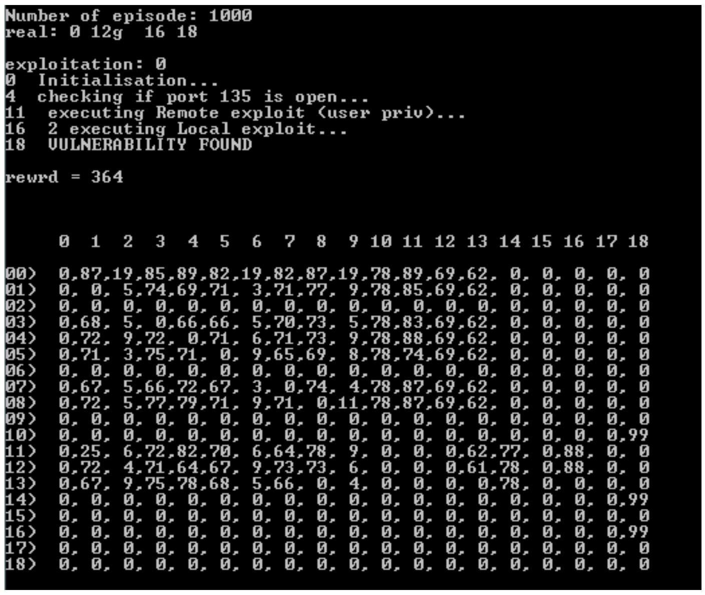
```
Number of episode: 100
real: 0 12g  7g  10g  18

exploitation 0
 Initialisation...
 checking if Service1 functioning...
 checking executing SAM_Remote exploits exploit(admin)...
 VULNERABILITY FOUND

rewrd = 258


     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00)  0,69,19,59,56,68,19,81,61,19,78,70,38,46, 0, 0, 0, 0, 0
01)  0, 0, 8, 0,33,47, 4,60, 0, 9,75,62,38, 0, 0, 0, 0, 0, 0
02)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03)  0,51, 4, 0,40, 0, 0,35, 0, 0,63,50,28,39, 0, 0, 0, 0, 0
04)  0, 0, 8, 0, 0,53, 5,20,39, 4, 0,16,28, 0, 0, 0, 0, 0, 0
05)  0,50, 4,45, 0, 0, 5,50, 0, 1,63,61, 4, 0, 0, 0, 0, 0, 0
06)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07)  0,55,19,39,20,51, 7, 0,53, 5,78,50, 0, 0, 0, 0, 0, 0, 0
08)  0, 0, 9, 0, 0,53, 0,53, 0, 8, 0,18, 6, 2, 0, 0, 0, 0, 0
09)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11)  0,56, 4, 0,25,27, 0,65,41, 8, 0, 0, 0,32, 0, 0, 0, 0, 0
12)  0, 0, 5,40, 0,46, 0,49, 0, 5, 0, 0, 0, 0, 0, 0, 8, 0, 0
13)  0,44, 4, 0, 0, 0, 8,51, 0, 4, 0, 0, 0, 0,73, 0, 0, 0, 0
14)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,80
17)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.5 MDP solution using ε-greedy policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.15

A solution was found:

- Initialization
- Check Service 1 existence
- Execute Remote Exploit (admin)
- Successful vulnerability exploitation

However, it is clear from the random policy testing that a more optimal solution exists. The quantity of episodes was increased to 1000 (Figure 4.6).



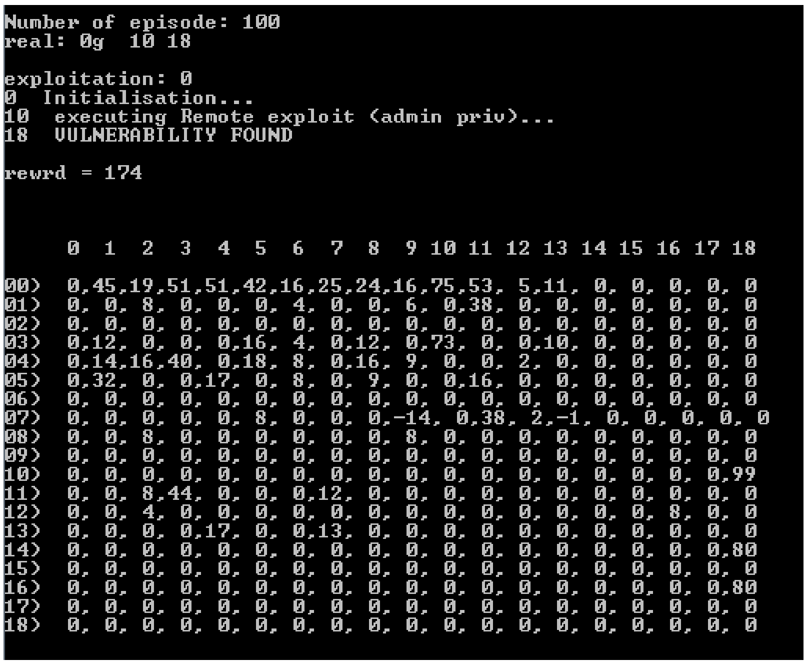
```
Number of episode: 1000
real: 0 12g  16 18

exploitation: 0
0  Initialisation...
4  checking if port 135 is open...
11  executing Remote exploit (user priv)...
16  2 executing Local exploit...
18  VULNERABILITY FOUND

rewrd = 364


     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00)  0,87,19,85,89,82,19,82,87,19,78,89,69,62, 0, 0, 0, 0, 0
01)  0, 0, 5,74,69,71, 3,71,77, 9,78,85,69,62, 0, 0, 0, 0, 0
02)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03)  0,68, 5, 0,66,66, 5,70,73, 5,78,83,69,62, 0, 0, 0, 0, 0
04)  0,72, 9,72, 0,71, 6,71,73, 9,78,88,69,62, 0, 0, 0, 0, 0
05)  0,71, 3,75,71, 0, 9,65,69, 8,78,74,69,62, 0, 0, 0, 0, 0
06)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07)  0,67, 5,66,72,67, 3, 0,74, 4,78,87,69,62, 0, 0, 0, 0, 0
08)  0,72, 5,77,79,71, 9,71, 0,11,78,87,69,62, 0, 0, 0, 0, 0
09)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11)  0,25, 6,72,82,70, 6,64,78, 9, 0, 0, 0,62,77, 0,88, 0, 0
12)  0,72, 4,71,64,67, 9,73,73, 6, 0, 0, 0,61,78, 0,88, 0, 0
13)  0,67, 9,75,78,68, 5,66, 0, 4, 0, 0, 0, 0,78, 0, 0, 0, 0
14)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
17)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.6 MDP solution using ε-greedy policy Q-learning 1000 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.15

The solution after 1000 episodes was different:

- Initialization
- Check if Port 135 is open
- Execute Remote Exploit (user privileges)
- Execute Local Exploit
- Successful vulnerability exploitation

The further increase of episodes did not change this result. At the next stage of the experiment, 1-ε was increased to 0.55 and the quantity of episodes decreased back to 100 (Figure 4.7).



```
Number of episode: 100
real: 0g  10 18

exploitation: 0
0  Initialisation...
10  executing Remote exploit (admin priv)...
18  VULNERABILITY FOUND

rewrd = 174



     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00)  0,45,19,51,51,42,16,25,24,16,75,53, 5,11, 0, 0, 0, 0, 0
01)  0, 0, 8, 0, 0, 0, 4, 0, 0, 6, 0,38, 0, 0, 0, 0, 0, 0, 0
02)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03)  0,12, 0, 0, 0,16, 4, 0,12, 0,73, 0, 0,10, 0, 0, 0, 0, 0
04)  0,14,16,40, 0,18, 8, 0,16, 9, 0, 0, 2, 0, 0, 0, 0, 0, 0
05)  0,32, 0, 0,17, 0, 8, 0, 9, 0, 0,16, 0, 0, 0, 0, 0, 0, 0
06)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07)  0, 0, 0, 0, 0, 8, 0, 0, 0,-14, 0,38, 2,-1, 0, 0, 0, 0, 0
08)  0, 0, 8, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0
09)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11)  0, 0, 8,44, 0, 0, 0,12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
12)  0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0
13)  0, 0, 0, 0,17, 0, 0,13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
14)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,80
15)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,80
17)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.7 MDP solution using ε-greedy policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.55

A solution was found:

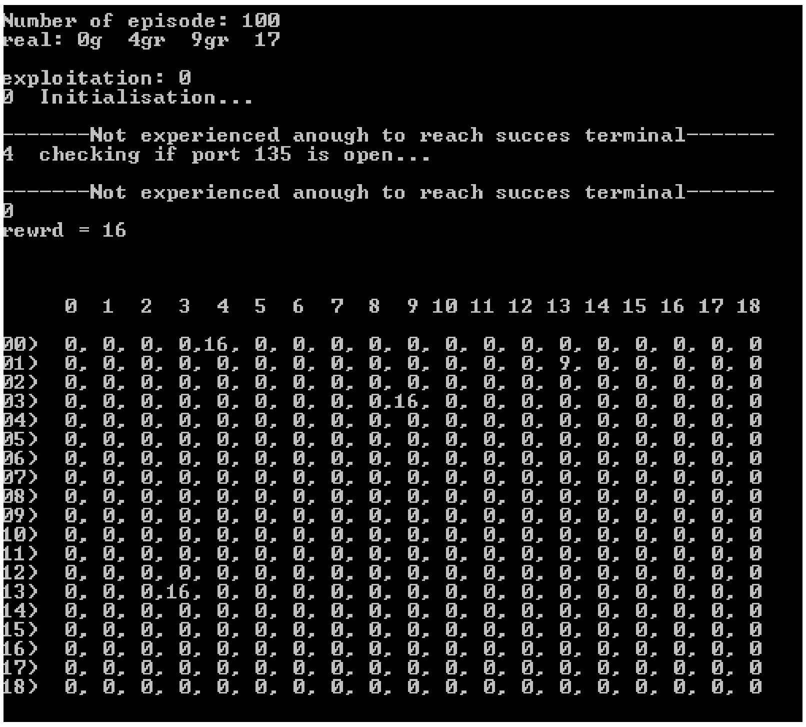
- Initialization
- Remote Exploit (admin) execution
- Successful vulnerability exploitation

This solution is not optimal, since Remote Exploit (admin) is not the most reliable from the set of available exploits and the agent did not even try to decrease the state space by probing actions. After the quantity of episodes was increased to 1000, the result became more optimal. A more reliable chain of exploits was found:

- Initialization
- Remote Exploit (user) execution
- Local Exploit execution
- Successful vulnerability exploitation

However, the agent still did not try to decrease the state space before exploit execution. Nothing changed at 3000 episodes, the solution was the same.

At the next stage of the experiment, 1- ε was increased to 0.98 and the quantity of episodes was decreased back to 100 (Figure 4.8).



```
Number of episode: 100
real: 0g  4gr  9gr  17

exploitation: 0
0  Initialisation...

-------Not experienced anough to reach succes terminal-------
4  checking if port 135 is open...

-------Not experienced anough to reach succes terminal-------
0
rewrd = 16


      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18

00)   0,  0,  0,  0,16,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
01)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  9,  0,  0,  0,  0,  0
02)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
03)   0,  0,  0,  0,  0,  0,  0,  0,  0,16,  0,  0,  0,  0,  0,  0,  0,  0,  0
04)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
05)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
06)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
07)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
08)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
09)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
10)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
11)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
12)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
13)   0,  0,  0,16,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
14)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
15)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
16)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
17)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
18)   0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0
```

Figure 4.8 MDP solution using ε-greedy policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.98

A solution was not found. It was found that 100 episodes was not enough for the agent to learn a successful strategy. The situation was the same after 1000 episodes.

After the quantity of learning episodes was increased to 3000, a solution was found but was not optimal:

- Initialization
- Probe action
- Remote Exploit (admin) execution

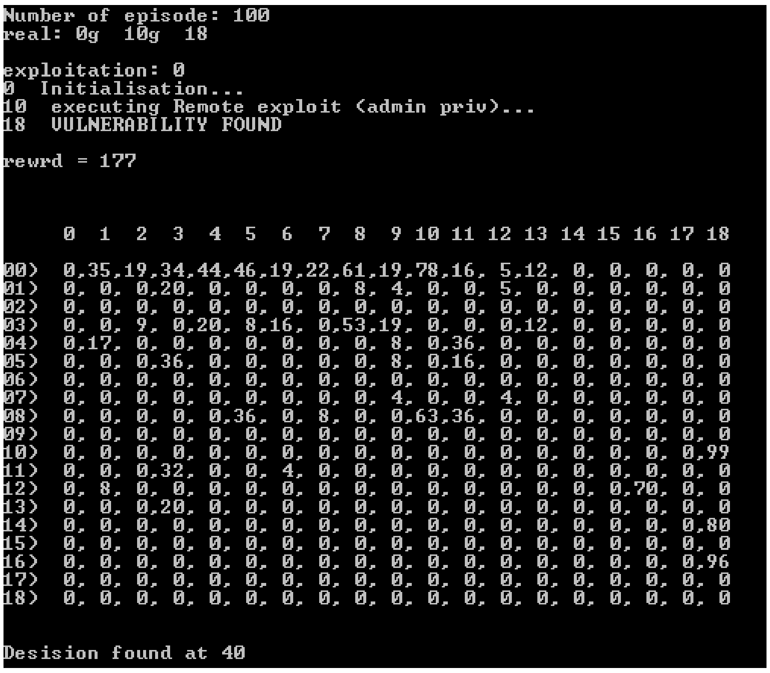
- Successful vulnerability exploitation

Even after 10000 episodes, the situation remained the same.

The next experiments were performed using the softmax acting policy.

4.1.6.3 Q-learning with softmax policy

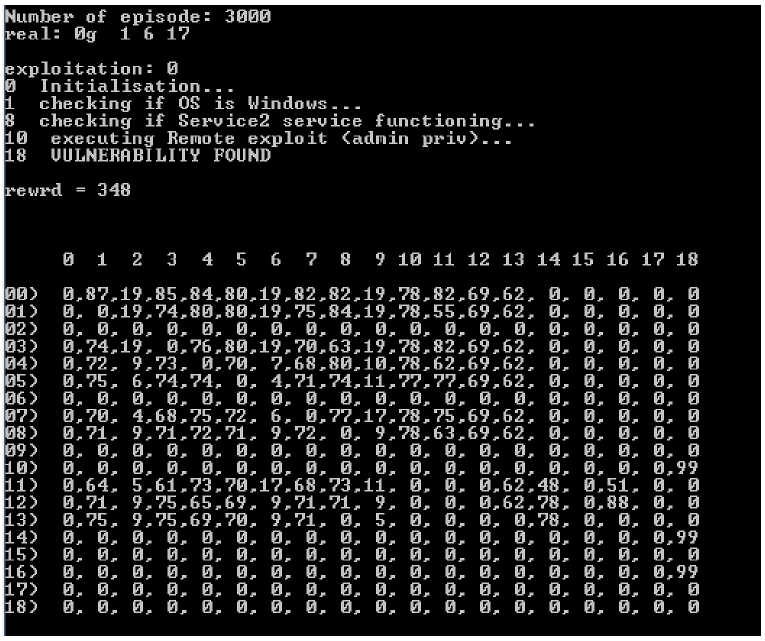
At the next stage of the experiment, Q-learning with softmax policy was applied. The solution after 100 episodes with Gamma=0.8, Alpha=0.8 is shown in Figure 4.9.



```
Number of episode: 100
real: 0g  10g  18

exploitation: 0
0  Initialisation...
10  executing Remote exploit (admin priv)...
18  VULNERABILITY FOUND

rewrd = 177


     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00)  0,35,19,34,44,46,19,22,61,19,78,16, 5,12, 0, 0, 0, 0, 0
01)  0, 0, 0,20, 0, 0, 0, 0, 8, 4, 0, 0, 5, 0, 0, 0, 0, 0, 0
02)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03)  0, 0, 9, 0,20, 8,16, 0,53,19, 0, 0, 0,12, 0, 0, 0, 0, 0
04)  0,17, 0, 0, 0, 0, 0, 0, 0, 8, 0,36, 0, 0, 0, 0, 0, 0, 0
05)  0, 0, 0,36, 0, 0, 0, 0, 0, 8, 0,16, 0, 0, 0, 0, 0, 0, 0
06)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07)  0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 4, 0, 0, 0, 0, 0, 0
08)  0, 0, 0, 0, 0,36, 0, 8, 0, 0,63,36, 0, 0, 0, 0, 0, 0, 0
09)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11)  0, 0, 0,32, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
12)  0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,70, 0, 0
13)  0, 0, 0,20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
14)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,80
15)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,96
17)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0


Desision found at 40
```

Figure 4.9 MDP solution using softmax policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8

Usage of Remote Exploit 1 (admin priv) is not an optimal solution. The state space was not decreased. Increasing the number of episodes to 1000 produced the same result. However, 3000 episodes leads to a more optimal solution (Figure 4.10). Before the execution of Remote Exploit 1(admin priv), the agent checks if the service exists. It decreases the state space. However, the more optimal Remote Exploit 2(userpriv) was not applied. After 10000 episodes, the solution stayed the same.



```
Number of episode: 3000
real: 0g  1 6 17

exploitation: 0
0  Initialisation...
1  checking if OS is Windows...
8  checking if Service2 service functioning...
10  executing Remote exploit (admin priv)...
18  VULNERABILITY FOUND

rewrd = 348


     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00)  0,87,19,85,84,80,19,82,82,19,78,82,69,62, 0, 0, 0, 0, 0
01)  0, 0,19,74,80,80,19,75,84,19,78,55,69,62, 0, 0, 0, 0, 0
02)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03)  0,74,19, 0,76,80,19,70,63,19,78,82,69,62, 0, 0, 0, 0, 0
04)  0,72, 9,73, 0,70, 7,68,80,10,78,62,69,62, 0, 0, 0, 0, 0
05)  0,75, 6,74,74, 0, 4,71,74,11,77,77,69,62, 0, 0, 0, 0, 0
06)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07)  0,70, 4,68,75,72, 6, 0,77,17,78,75,69,62, 0, 0, 0, 0, 0
08)  0,71, 9,71,72,71, 9,72, 0, 9,78,63,69,62, 0, 0, 0, 0, 0
09)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11)  0,64, 5,61,73,70,17,68,73,11, 0, 0, 0,62,48, 0,51, 0, 0
12)  0,71, 9,75,65,69, 9,71,71, 9, 0, 0, 0,62,78, 0,88, 0, 0
13)  0,75, 9,75,69,70, 9,71, 0, 5, 0, 0, 0, 0,78, 0, 0, 0, 0
14)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
17)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18)  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.10 MDP solution using softmax policy Q-learning 10000 episodes: Gamma=0.8, Alpha=0.8

Next section describes the experiments related to the performance of the learning algorithm in regards of different parameters and acting policies.

4.1.7 Learning performance

The performance of the learning algorithm depending on the parameters and policies is presented below.

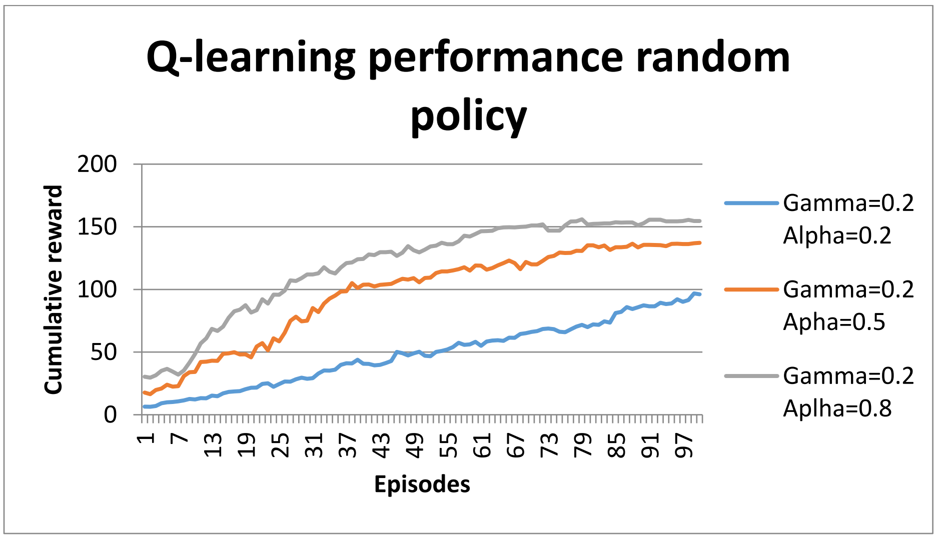


Figure 4.11 Random policy Q-learning performance with Gamma=0.2

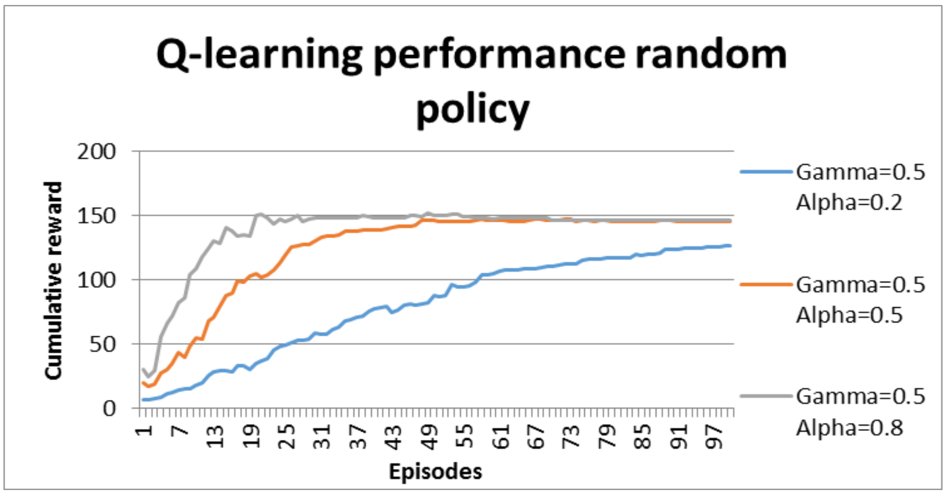


Figure 4.12 Random policy Q-learning performance with Gamma=0.5

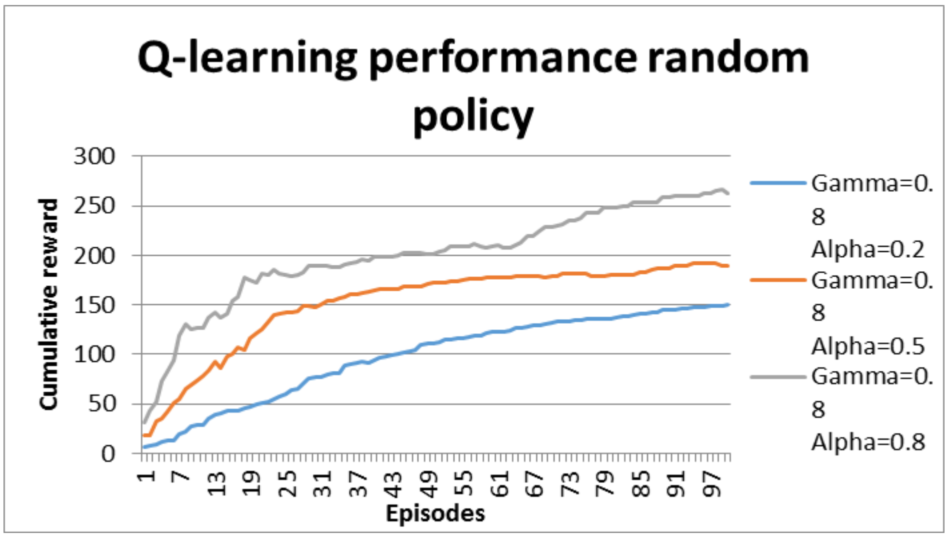


Figure 4.13 Random policy Q-learning performance with Gamma=0.8

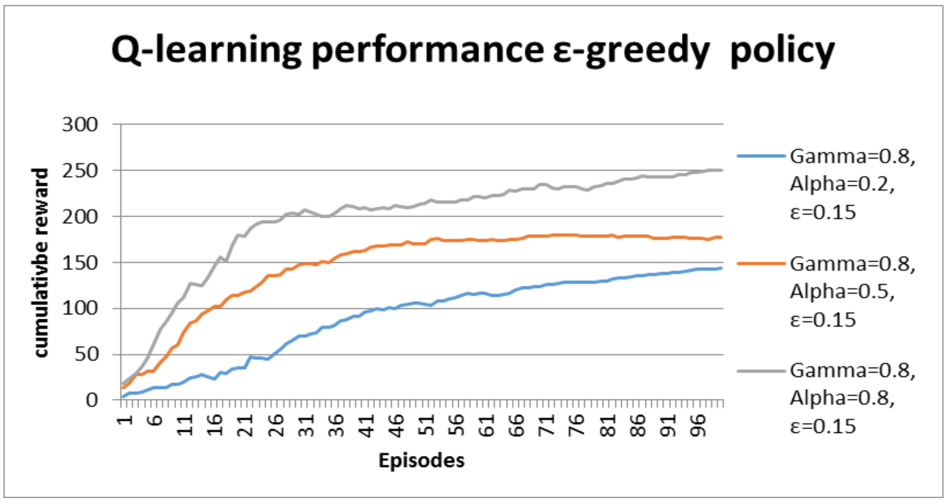


Figure 4.14 *ε-greedy policy Q-learning performance with Gamma=0.8, Alpha=0.2/0.5/0.8, 1- ε=0.15*

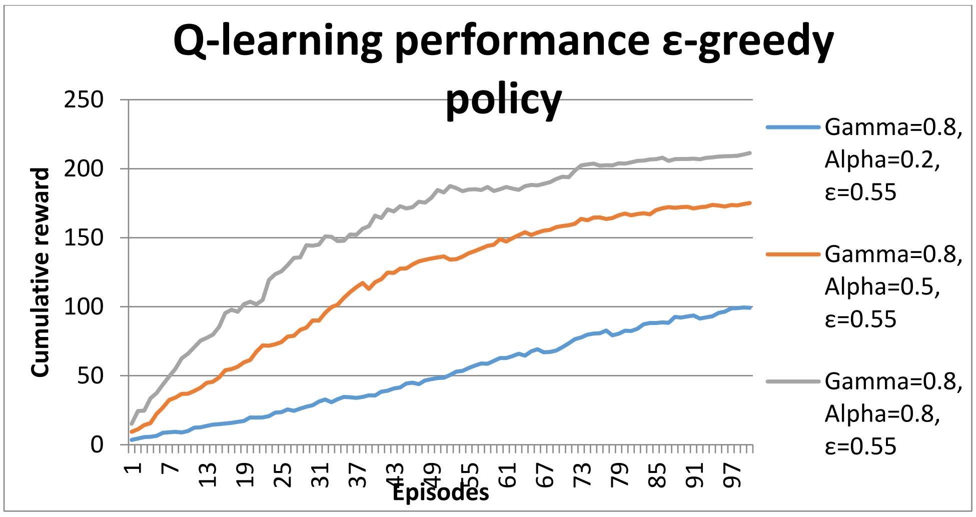


Figure 4.15 ε-greedy policy Q-learning performance with Gamma=0.8, Alpha=0.2/0.5/0.8, 1- ε=0.55

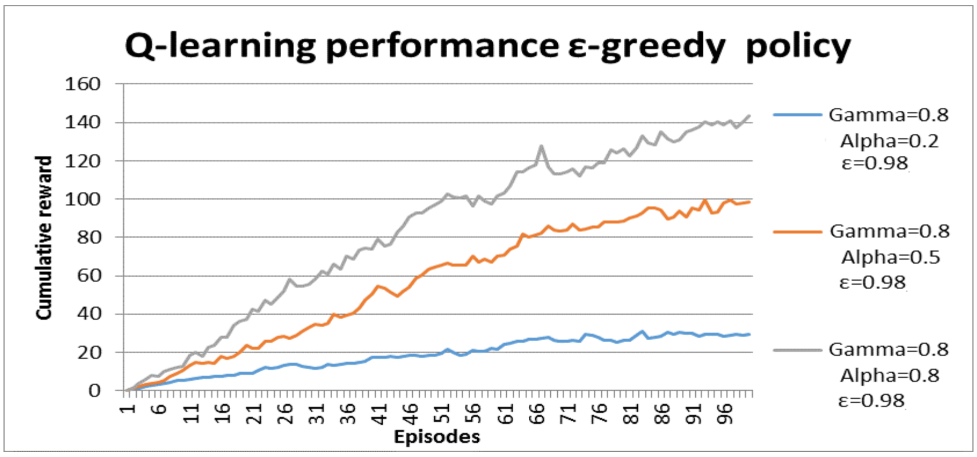


Figure 4.16 ε-greedy policy Q-learning performance with Gamma=0.8, Alpha=0.2/0.5/0.8, 1-ε=0.98

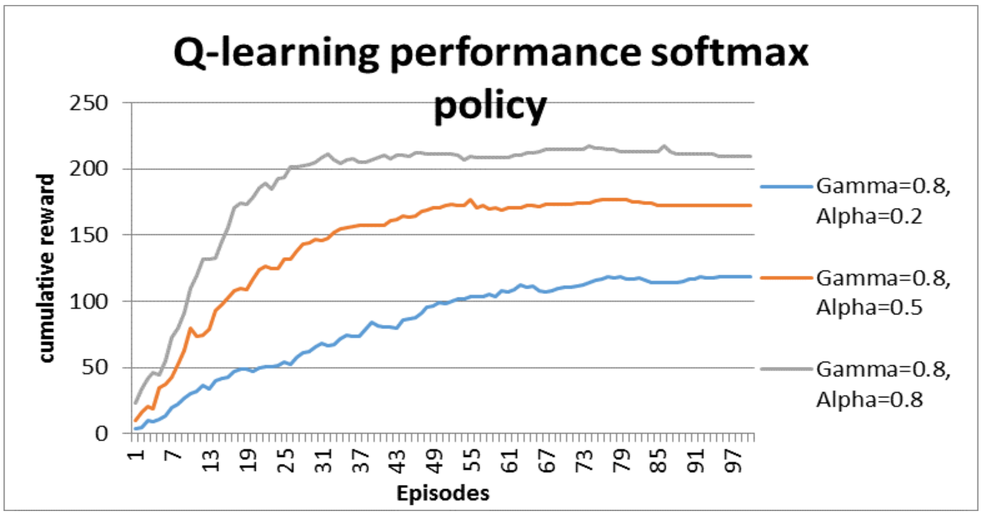


Figure 4.17 *Softmax policy Q-learning performance with Gamma=0.8, Alpha=0.2/0.5/0.8*

The performance of the Q-learning algorithm with random policy was tested using Gamma=0.2 and Alpha=0.2/0.5/0.8. The quantity of episodes is 100 averaged by 100 tests (Figure 4.11). None of the three curves converge to their maximum reward smoothly. This fact indicates stabilization problems. However, the Q-learning algorithm with parameters Gamma=0.2 and Alpha=0.8 has the best performance, since its curve is consistently above the others and the value of the maximum cumulative reward is bigger.

At the next step, Gamma was increased to 0.5 and Alpha=0.2/0.5/0.8. The quantity of episodes was the same: 100 averaged by 100 tests (Figure 4.12). The curves become a bit smoother. The algorithms using Gamma=0.5, Alpha=0.5/0.8 converge to their maximum rewards much faster. The algorithm with Gamma=0.5, Alpha=0.8 has the best performance.

Finally, Gamma was increased to 0.8, Alpha=0.2/0.5/0.8 (Figure 4.13). The curves become much smoother. The maximum reward is significantly higher. The most effective algorithm with Gamma=0.8, Alpha=0.8 converges to the cumulative reward value 267, which is almost double the reward value of the algorithm with Alpha=0.2.

In order to test the performance of the Q-learning algorithm with the ε-greedy policy, Gamma was set at 0.8, Alpha=0.2/0.5/0.8 and 1- ε=0.15 (Figure 4.14). The optimal parameters for 1-ε=0.15 are Gamma=0.8, Alpha=0.8, since the curve representing these parameters is above the others. It converges to a much higher cumulative reward, with the value at 250 compared to the other curves at 175 and 142. It is quite sharp at the beginning and its value increases faster.

The same parameters were tested for 1-ε=0.55 (Figure 4.15). The best performance was shown with 1-ε=0.55, Gamma=0.8, Alpha=0.2. All curves are flat and convergence to the max cumulative reward became smoother.

The next test was performed with 1-ε=0.98, Gamma=0.8, Alpha=0.2/0.5/0.8 (Figure 4.16). Curves become flat but less stabilized. There are a large number of small peaks. The best performance is the algorithm with parameters Gamma=0.8, Alpha=0.8, 1-ε=0.98.

The performance of Q-learning with softmax policy with Gamma=0.8 and Alpha=0.2/0.5/0.8 is shown in Figure 4.17. The best performance was shown with Gamma=0.8, Alpha=0.8. In the case where the learning rate is less than 0.8, the curve becomes flatter and convergence to the max cumulative reward becomes slower.

4.1.8 Conclusion

The hypothesis that Q-learning can be used to make the penetration testing system learn attack strategies itself on the go without predefined reward function was demonstrated. The acting agent was able to learn multistep attacks as well, consisting of probing actions and exploit execution, external + local in a sequence.

The performance of the learning algorithm demonstrated convergence to optimality with the number of episodes increased. The best Q-learning hyper-parameters were identified: learning rate = 0.8, discount factor = 0.8, and the best acting policy was ε-greedy with 1-ε=0.15.

This experiment partly proved the application of a model-free approach to the penetration testing. It showed that the agent can successfully learn the optimal attack strategy without a predefined reward function, describing all possible rewards for all possible states-actions pairs. The reward was derived directly from the environmental feedback. Therefore, the penetration testing system is one step closer to the automation, since the reward for the state/action pair is derived by the audit agent itself.

The transition function for the MDP was predefined. In order to fully approve the application of the model-free approach, the predefined transition function should be withdrawn in addition to the reward function. It will let the agent derive the model of the environment directly from acting. It will itself learn expected rewards and possible transitions. The Field experiment section is intended to test such a hypothesis.

4.2 Field experiment

4.2.1 Hypothesis

This experiment is intended to address a real-life uncertainty and to prove that, in this scenario, the penetration testing system will be able to learn attack strategies on the go, and, in addition, will learn multistep attacks. The uncertainty of a real-life scenario is defined by the agent's lack of knowledge about which actions lead to the states and actions that guarantee the highest cumulative reward. In other words, there are no predefined transition and reward functions for the agent. It should learn this information during the experiment.

4.2.2 Experiment setup

The experiment uses the same testing infrastructure as that in the MDP experiment described in the Environment architecture section of the Methodology chapter.

The context of the Field experiment differs from that of the MDP experiment. In real life, the result of the penetration testing action is not known in advance since the tested system is always different. That is why it is not possible to define rewards and transitions in the R-matrix for the Q-learning algorithm statically. However, this information can be derived from the environmental feedback while acting. After the action is executed and feedback from the environment has been received, the knowledge that the agent has about the environment will be updated. It is accumulated in the Q-matrix representing agent experience.

The field experiment differs from the MDP experiment by additional requirement withdrawal. It does not have a predefined transitions function, and, furthermore, it does not have a predefined reward function. It is the representation of a free-model approach to the real-world scenario. The model of the environment is not given to the agent. The only knowledge it has is state and action spaces. It does not know which actions are better (which guarantees the highest cumulative reward). Furthermore, the agent has no information about which actions lead to which states. This experiment is aimed at validating the project hypothesis while addressing the uncertainty of a real-life scenario. After hypothesis validation, the performance of the learning algorithm was tested by changing different hyper-parameters such as the: quantity of episodes, learning rate and discount factor.

The MDP process is the same as in the MDP experiment with the only difference being that transitions are unknown to the agent. The agent has knowledge about state and action space only.

The reward is defined in the same way as in the MDP experiment. Additionally, the agent is punished by receiving negative reward while it’s stuck in the same state.

A result assessment is done as same way as in the MDP experiment, the learning algorithm performance is compared by cumulative rewards graphically. The higher the graphic curve rises of the cumulative reward across episodes, the better the performance of the learning algorithm. The faster it grows, the faster an acting agent learns.

4.2.3 Experiment outcome

This section describes the outcome of the Field experiment. The experimental conditions are described in the section Experiment setup. Different acting policies were tested:

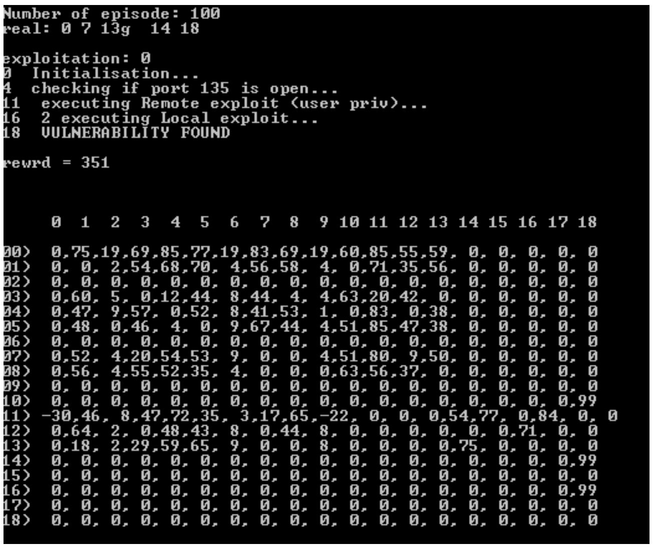
- Random policy – Actions are chosen randomly.
- ε-greedy policy – Actions are chosen randomly or according to experience with a certain probability.

- Softmax policy – Actions are chosen randomly but some actions are chosen more often with the higher potential reward. This policy tries to avoid the states with a very low potential reward.

In order to check the learning algorithm performance for every policy, a variety of hyper-parameters were tested such as: iteration quantity, learning rate and discount factor.

4.2.3.1 Q-learning with random policy

The field experiment with random policy after 100 episodes with Gamma=0.8, Alpha=0.8 is shown in Figure 4.18.



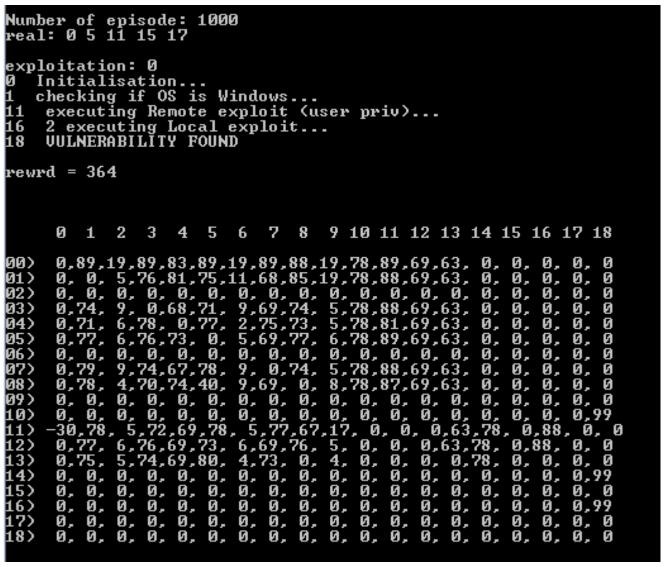
```
Number of episode: 100
real: 0 7 13g  14 18

exploitation: 0
0  Initialisation...
4  checking if port 135 is open...
11  executing Remote exploit (user priv)...
16  2 executing Local exploit...
18  VULNERABILITY FOUND

rewrd = 351


      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00)   0,75,19,69,85,77,19,83,69,19,60,85,55,59, 0, 0, 0, 0, 0
01)   0, 0, 2,54,68,70, 4,56,58, 4, 0,71,35,56, 0, 0, 0, 0, 0
02)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03)   0,60, 5, 0,12,44, 8,44, 4, 4,63,20,42, 0, 0, 0, 0, 0, 0
04)   0,47, 9,57, 0,52, 8,41,53, 1, 0,83, 0,38, 0, 0, 0, 0, 0
05)   0,48, 0,46, 4, 0, 9,67,44, 4,51,85,47,38, 0, 0, 0, 0, 0
06)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07)   0,52, 4,20,54,53, 9, 0, 0, 4,51,80, 9,50, 0, 0, 0, 0, 0
08)   0,56, 4,55,52,35, 4, 0, 0, 0,63,56,37, 0, 0, 0, 0, 0, 0
09)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11) -30,46, 8,47,72,35, 3,17,65,-22, 0, 0, 0,54,77, 0,84, 0, 0
12)   0,64, 2, 0,48,43, 8, 0,44, 8, 0, 0, 0, 0, 0, 0,71, 0, 0
13)   0,18, 2,29,59,65, 9, 0, 0, 8, 0, 0, 0, 0,75, 0, 0, 0, 0
14)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
17)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18)   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.18: Field test random policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8

Figure 4.18 demonstrates the sequence of actions leading to the successful penetration of the target and Q-matrix defining agent experience. After 100 episodes, the optimal solution was found. The attack strategy was the following:

- Initialization.
- Check if Port 135 is open.
- Execute Remote Exploit (user privileges)
- Execute Local exploit to escalate privileges.

After increasing the quantity of learning episodes to 1000, the same optimal solution was found but with different probe action (Figure 4.4).



```
Number of episode: 1000
real: 0 5 11 15 17

exploitation: 0
0  Initialisation...
1  checking if OS is Windows...
11  executing Remote exploit (user priv)...
16  2 executing Local exploit...
18  VULNERABILITY FOUND

rewrd = 364


      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00>   0,89,19,89,83,89,19,89,88,19,78,89,69,63, 0, 0, 0, 0, 0
01>   0, 0, 5,76,81,75,11,68,85,19,78,88,69,63, 0, 0, 0, 0, 0
02>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03>   0,74, 9, 0,68,71, 9,69,74, 5,78,88,69,63, 0, 0, 0, 0, 0
04>   0,71, 6,78, 0,77, 2,75,73, 5,78,81,69,63, 0, 0, 0, 0, 0
05>   0,77, 6,76,73, 0, 5,69,77, 6,78,89,69,63, 0, 0, 0, 0, 0
06>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07>   0,79, 9,74,67,78, 9, 0,74, 5,78,88,69,63, 0, 0, 0, 0, 0
08>   0,78, 4,70,74,40, 9,69, 0, 8,78,87,69,63, 0, 0, 0, 0, 0
09>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11> -30,78, 5,72,69,78, 5,77,67,17, 0, 0, 0,63,78, 0,88, 0, 0
12>   0,77, 6,76,69,73, 6,69,76, 5, 0, 0, 0,63,78, 0,88, 0, 0
13>   0,75, 5,74,69,80, 4,73, 0, 4, 0, 0, 0, 0,78, 0, 0, 0, 0
14>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
17>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.19 Field test random policy Q-learning 1000 episodes: Gamma=0.8, Alpha=0.8

After 1000 episodes, the most effective solution was found:

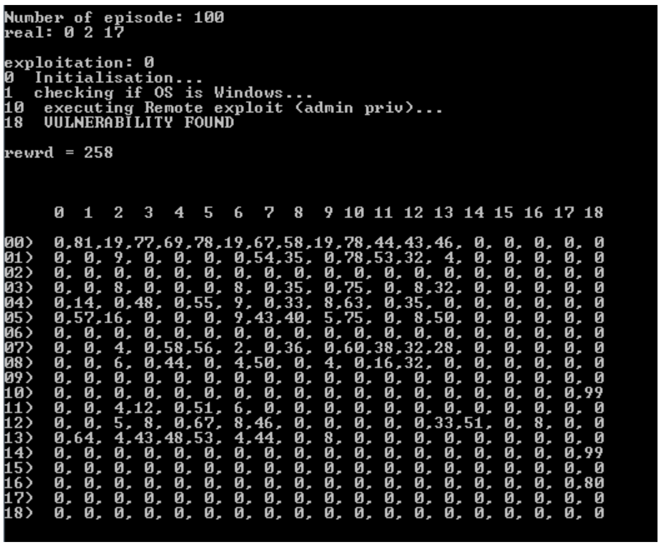
- Initialization.
- OS check.
- Remote exploit (user privileges) execution.
- Local exploit execution.

With a further increase of episodes, this result remained the same.

The next set of experiments is intended to test the ε-greedy acting policy. This policy makes the agent act randomly or according to its experience with a certain probability.

4.2.3.2 Q-learning with ε-greedy policy

The field experiment with ε-greedy acting policy after 100 episodes with Gamma=0.8, Alpha=0.8 is shown in Figure 4.20. The agent has chosen random actions with probability 0.15.



```
Number of episode: 100
real: 0 2 17

exploitation: 0
0  Initialisation...
1  checking if OS is Windows...
10  executing Remote exploit (admin priv)...
18  VULNERABILITY FOUND

rewrd = 258



     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00>  0,81,19,77,69,78,19,67,58,19,78,44,43,46, 0, 0, 0, 0, 0
01>  0, 0, 9, 0, 0, 0, 0,54,35, 0,78,53,32, 4, 0, 0, 0, 0, 0
02>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03>  0, 0, 8, 0, 0, 0, 8, 0,35, 0,75, 0, 8,32, 0, 0, 0, 0, 0
04>  0,14, 0,48, 0,55, 9, 0,33, 8,63, 0,35, 0, 0, 0, 0, 0, 0
05>  0,57,16, 0, 0, 0, 9,43,40, 5,75, 0, 8,50, 0, 0, 0, 0, 0
06>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07>  0, 0, 4, 0,58,56, 2, 0,36, 0,60,38,32,28, 0, 0, 0, 0, 0
08>  0, 0, 6, 0,44, 0, 4,50, 0, 4, 0,16,32, 0, 0, 0, 0, 0, 0
09>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11>  0, 0, 4,12, 0,51, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
12>  0, 0, 5, 8, 0,67, 8,46, 0, 0, 0, 0, 0,33,51, 0, 8, 0, 0
13>  0,64, 4,43,48,53, 4,44, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0
14>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,80
17>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

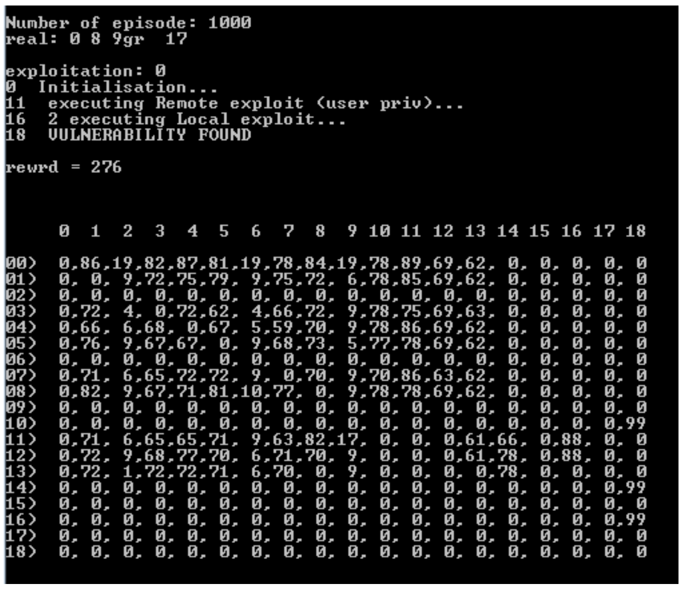
Figure 4.20 Field test ε-greedy policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.15

After 100 episodes, the solution was found:

- Initialization.
- OS check.
- Remote exploit (admin privileges) execution.

It is not the optimal sequence of actions. The advantage of this solution is that probe action was used, which is intended to decrease the state space. However, the more dangerous exploit was used, which has a risk of crashing the target system. The quantity of episodes was increased to 1000 (

Figure 4.21).



```
Number of episode: 1000
real: 0 8 9gr  17

exploitation: 0
0  Initialisation...
11  executing Remote exploit (user priv)...
16  2 executing Local exploit...
18  VULNERABILITY FOUND

rewrd = 276



     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00>  0,86,19,82,87,81,19,78,84,19,78,89,69,62, 0, 0, 0, 0, 0
01>  0, 0, 9,72,75,79, 9,75,72, 6,78,85,69,62, 0, 0, 0, 0, 0
02>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03>  0,72, 4, 0,72,62, 4,66,72, 9,78,75,69,63, 0, 0, 0, 0, 0
04>  0,66, 6,68, 0,67, 5,59,70, 9,78,86,69,62, 0, 0, 0, 0, 0
05>  0,76, 9,67,67, 0, 9,68,73, 5,77,78,69,62, 0, 0, 0, 0, 0
06>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07>  0,71, 6,65,72,72, 9, 0,70, 9,70,86,63,62, 0, 0, 0, 0, 0
08>  0,82, 9,67,71,81,10,77, 0, 9,78,78,69,62, 0, 0, 0, 0, 0
09>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11>  0,71, 6,65,65,71, 9,63,82,17, 0, 0, 0,61,66, 0,88, 0, 0
12>  0,72, 9,68,77,70, 6,71,70, 9, 0, 0, 0,61,78, 0,88, 0, 0
13>  0,72, 1,72,72,71, 6,70, 0, 9, 0, 0, 0, 0,78, 0, 0, 0, 0
14>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
17>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.21 *Field test ε-greedy policy Q-learning 1000 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.15*

The solution after 1000 episodes was better:

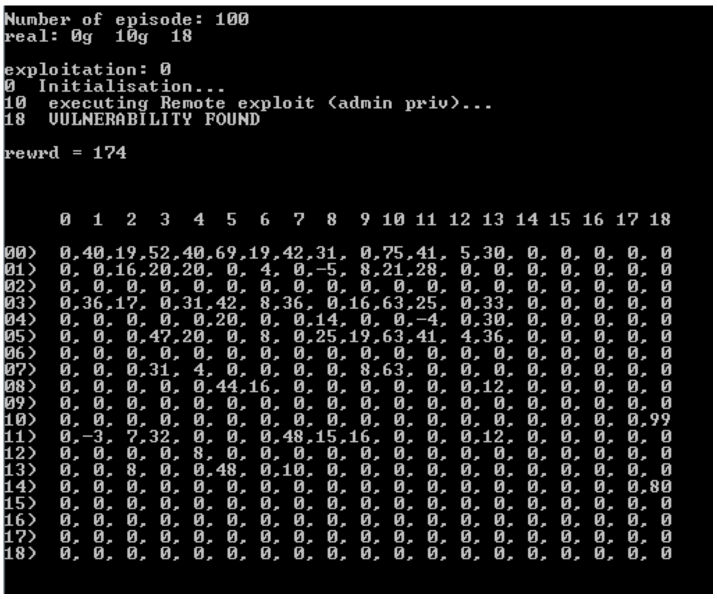
- Initialization.
- Remote exploit (user privileges) execution.
- Local exploit (rights escalation).

However, it was not the optimal solution. The probe action was missed. A further increase of episodes resolved that problem and the solution converged to the optimal one.

The next set of experiments is related to the softmax acting policy. This policy makes actions that are chosen randomly but some actions are chosen more often with the higher potential reward. This policy tries to avoid the states with a very low potential reward.

4.2.3.3 Q-learning with softmax policy

The field experiment with softmax policy after 100 episodes with Gamma=0.8, Alpha=0.8 is shown in Figure 4.22. A solution was found but it was not the optimal one.



```
Number of episode: 100
real: 0g  10g  18

exploitation: 0
0  Initialisation...
10  executing Remote exploit (admin priv)...
18  VULNERABILITY FOUND

rewrd = 174



      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00>   0,40,19,52,40,69,19,42,31, 0,75,41, 5,30, 0, 0, 0, 0, 0
01>   0, 0,16,20,20, 0, 4, 0,-5, 8,21,28, 0, 0, 0, 0, 0, 0, 0
02>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03>   0,36,17, 0,31,42, 8,36, 0,16,63,25, 0,33, 0, 0, 0, 0, 0
04>   0, 0, 0, 0, 0,20, 0, 0,14, 0, 0,-4, 0,30, 0, 0, 0, 0, 0
05>   0, 0, 0,47,20, 0, 8, 0,25,19,63,41, 4,36, 0, 0, 0, 0, 0
06>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07>   0, 0, 0,31, 4, 0, 0, 0, 0, 8,63, 0, 0, 0, 0, 0, 0, 0, 0
08>   0, 0, 0, 0, 0,44,16, 0, 0, 0, 0, 0, 0,12, 0, 0, 0, 0, 0
09>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11>   0,-3, 7,32, 0, 0, 0,48,15,16, 0, 0, 0,12, 0, 0, 0, 0, 0
12>   0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
13>   0, 0, 8, 0, 0,48, 0,10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
14>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,80
15>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
17>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18>   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

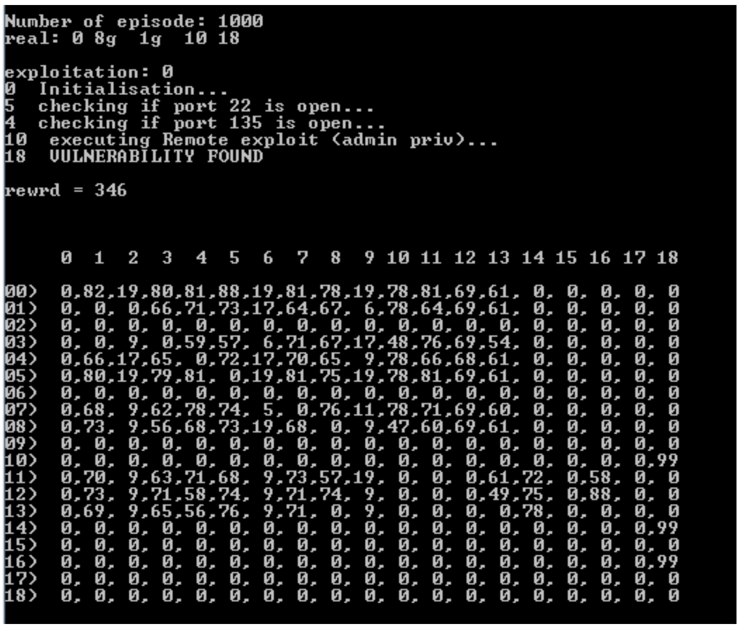
Figure 4.22 Field test softmax policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8

A solution was found after 100 episodes:

- Initialization.

- Remote exploit (admin privileges) execution.

The sequence of actions was not the most effective. There was no probe action used and remote exploit (admin) was executed, which is more harmful for the host. The quantity of episodes was increased to 1000. The solution is presented in Figure 4.23.



```
Number of episode: 1000
real: 0 8g  1g  10 18

exploitation: 0
0  Initialisation...
5  checking if port 22 is open...
4  checking if port 135 is open...
10  executing Remote exploit (admin priv)...
18  VULNERABILITY FOUND

rewrd = 346



     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

00>  0,82,19,80,81,88,19,81,78,19,78,81,69,61, 0, 0, 0, 0, 0
01>  0, 0, 0,66,71,73,17,64,67, 6,78,64,69,61, 0, 0, 0, 0, 0
02>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
03>  0, 0, 9, 0,59,57, 6,71,67,17,48,76,69,54, 0, 0, 0, 0, 0
04>  0,66,17,65, 0,72,17,70,65, 9,78,66,68,61, 0, 0, 0, 0, 0
05>  0,80,19,79,81, 0,19,81,75,19,78,81,69,61, 0, 0, 0, 0, 0
06>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
07>  0,68, 9,62,78,74, 5, 0,76,11,78,71,69,60, 0, 0, 0, 0, 0
08>  0,73, 9,56,68,73,19,68, 0, 9,47,60,69,61, 0, 0, 0, 0, 0
09>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
10>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
11>  0,70, 9,63,71,68, 9,73,57,19, 0, 0, 0,61,72, 0,58, 0, 0
12>  0,73, 9,71,58,74, 9,71,74, 9, 0, 0, 0,49,75, 0,88, 0, 0
13>  0,69, 9,65,56,76, 9,71, 0, 9, 0, 0, 0, 0,78, 0, 0, 0, 0
14>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
15>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
16>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,99
17>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
18>  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.23 Field test softmax policy Q-learning 1000 episodes: Gamma=0.8, Alpha=0.8

The solution after 1000 episodes was better:

- Initialization.
- Checking Port 22.
- Checking Port 135.
- Remote exploit (admin privileges) execution.

However, it was not the optimal solution. There were two similar probe actions used. The further increase of episodes resolved that problem and the solution converged to the optimal one. After 5000 episodes, the solution was:

- Initialization.
- Checking OS.
- Remote exploit (user privileges) execution.
- Local exploit (escalate privileges).

Next section describes the experiments related to the performance of the learning algorithm in regards of different parameters and acting policies.

4.2.4 Learning performance

The performance of the learning algorithm depending on the parameters and policies is presented below.

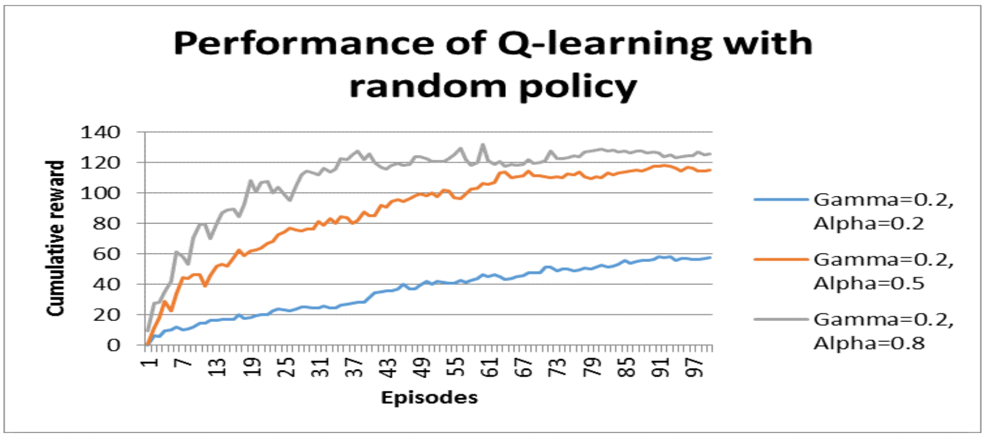


Figure 4.24 Performance of random policy Q-learning 100 episodes: Gamma=0.2, Alpha = 0.2/0.5/0.8

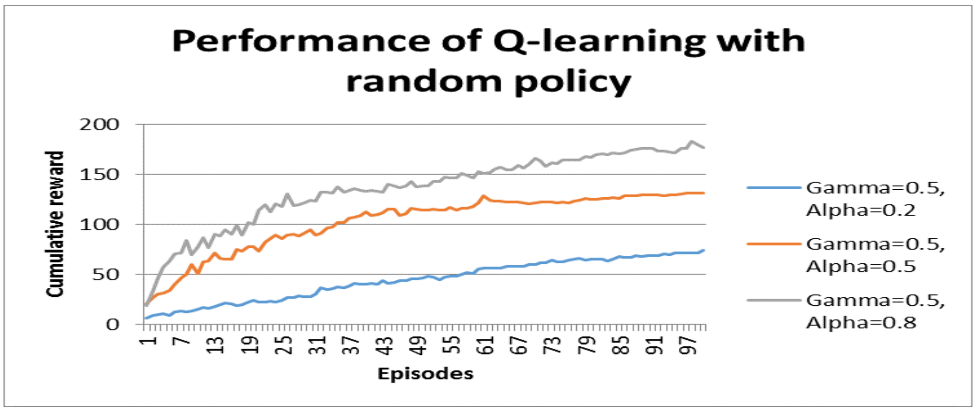


Figure 4.25 Performance of random policy Q-learning 100 episodes: Gamma=0.5, Alpha = 0.2/0.5/0.8

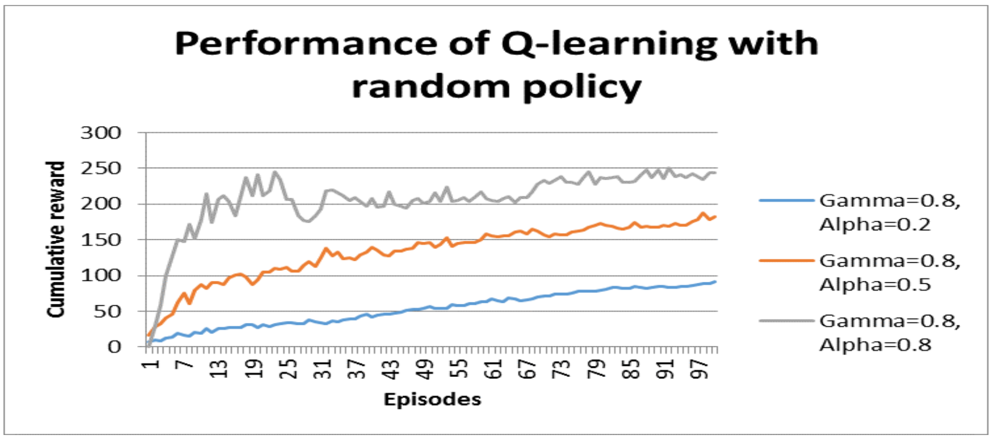


Figure 4.26 Performance of random policy Q-learning 100 episodes: Gamma=0.8, Alpha = 0.2/0.5/0.8

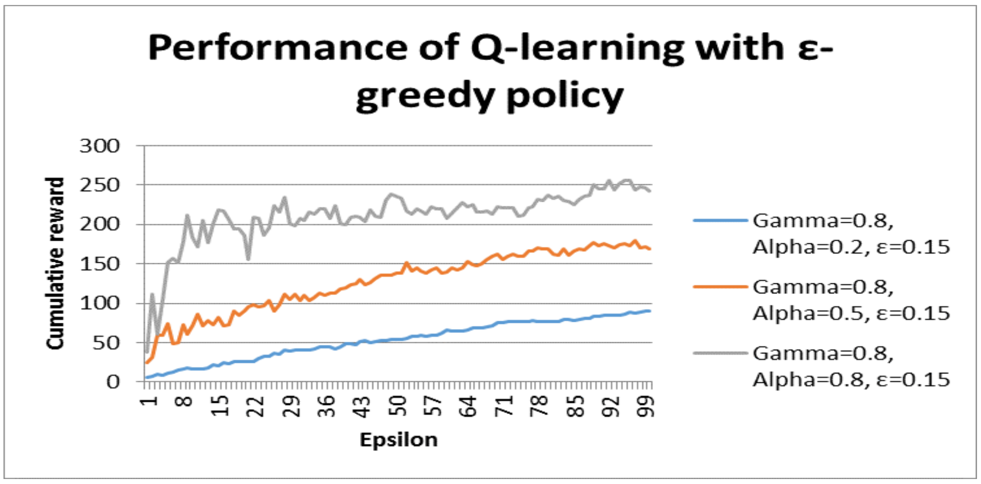


Figure 4.27 Performance of ε-greedy policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.2/0.5/0.8

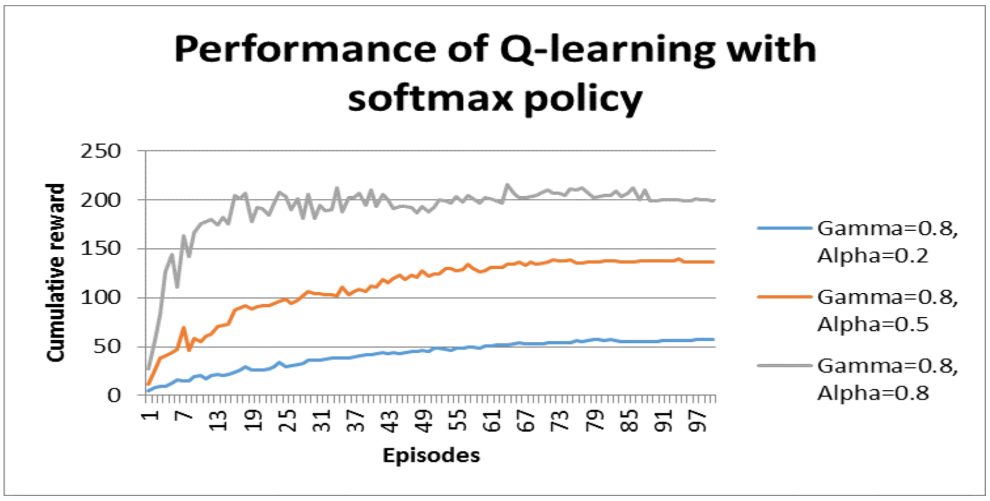


Figure 4.28 Performance of softmax policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.2/0.5/0.8

The performance of the Q-learning algorithm with random policy was tested using Gamma=0.2 and Alpha=0.2/0.5/0.8. The quantity of episodes is 100 averaged by 100 tests. The performance of Q-learning with random policy with different values of Gamma and Alpha is shown in Figure 4.24, and Figure 4.25.

In Figure 4.24, the curve with Gamma=0.2 and Alpha=0.2 smoothly converges to the solution. However, it does not accumulate a high cumulative reward. The rest of the curves are not smooth. The algorithm with parameters Gamma=0.2 and Alpha=0.5 has a better performance. The convergence is faster and the accumulated cumulative reward is higher. The Q-learning algorithm with parameters Gamma=0.2 and Alpha=0.8 has the best performance since its curve is fully above the others and the value of the maximum cumulative reward value is higher.

Figure 4.25 represents the increase of Gamma parameter (Gamma=0.5). Curves became smoother. The algorithm with Gamma=0.5 and Alpha=0.8 demonstrated the best performance.

In Figure 4.26, it is clear that the maximum reward is significantly higher for Gamma=0.8 and Alpha=0.8. Those values are the most effective for the Q-learning algorithm random acting policy.

Figure 4.27 represents the performance of Q-learning with ε-greedy policy with different values of Gamma and Alpha. The maximum reward is significantly higher for Gamma=0.8 and Alpha=0.8, ε = 0.15. Those hyper-parameters values are the most effective for the ε-greedy acting policy from the performance point of view. They make the algorithm learn faster and converge to the higher cumulative reward.

The performance of Q-learning with softmax policy is shown in Figure 4.28. The algorithm with Gamma=0.8, Alpha=0.8 showed the most effective learning process. The curve representing the best solution indicated some stabilization problems. However, it converged to the optimal solution.

4.2.5 Conclusion

This experiment was able to demonstrate the hypothesis that the penetration testing system will be able to learn attack strategies on the go. The multistep attack sequence was found. The algorithm was able to learn the optimal attack strategy in conditions where a model-free approach was applied. In other words, the experiment demonstrated the hypothesis in conditions where there is no transition and reward function provided, which describes all transitions and rewards from all states. Such information was learned by the agent on the go, while acting.

The current experiment validated the model-free approach for the particular MDP example. It motivated the extension of such an approach to the general case, so that it can be used in real-world scenarios. The generalization is presented in the later Deep Architecture ANN experiment and Deep architecture RNN experiment.

Importantly, the experiment demonstrated the core hypothesis about the possibility of developing self-learning penetration testing. The attack toolset of the penetration testing system is the set of exploits which can be executed against the target. Those exploits need to be renewed from time to time in order to keep the attack system up to date. This requirement motivated us to apply our model-free approach based on the Q-learning algorithm to make the penetration testing system learn new ways to exploit potential new vulnerabilities itself. As a result, the system should be able to create new exploits and add them to its toolset. It will make the penetration testing tool significantly autonomous and self-updating. It was decided to validate the approach on Buffer Overflow vulnerability. The experiment is presented in the New vulnerabilities experiment section.

4.3 New vulnerabilities experiment

4.3.1 Hypothesis

This experiment aims to demonstrate that the Q-learning algorithm can be used to make a penetration testing system learn vulnerability exploitation, in particular buffer overflow, which it has not experienced before. The approach is model-free. In other words, there are no transition and reward functions, predefined for all possible states.

4.3.2 Experiment setup

This experiment is intended to make a penetration system learn to exploit the buffer overflow vulnerability. The buffer overflow vulnerability allows the possibility of abnormal behavior by the program, so that it overruns the boundary of the buffer and rewrites additional memory locations. As a result, it will be able to execute malicious unauthorized actions.

The mechanism of the buffer overflow is simple. While the program runs, different functions are executed in a particular order. When a new function is to be executed, the running environment saves the current address (return address), puts the function parameters in the stack and jumps to the function code address. When the function is executed, the running environment uses the return address saved in the stack to restart the execution of the main program. However, in the case where the return address is rewritten by some unauthorized actions, the program will restart execution of the code stored at this new memory address. The buffer overflow vulnerability gives an opportunity to the attacker to overrun the buffer of the vulnerable function stored in the stack. As a result, the attacker can rewrite the return address stored in one of the stack registers after the buffer. The stack example is shown in Figure 4.29.

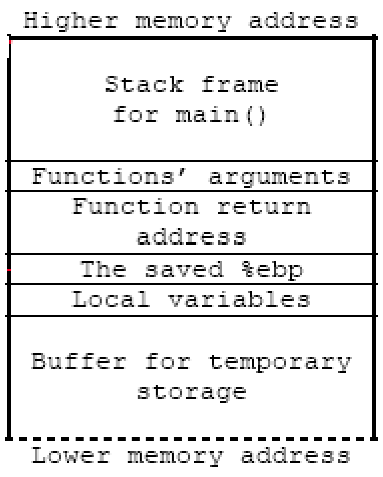


Figure 4.29 The memory storage of parameters in the stack

Similar to the 'MDP experiment', the experimental infrastructure consists of two hosts (Figure 4.30): attacker(1) and target(2).

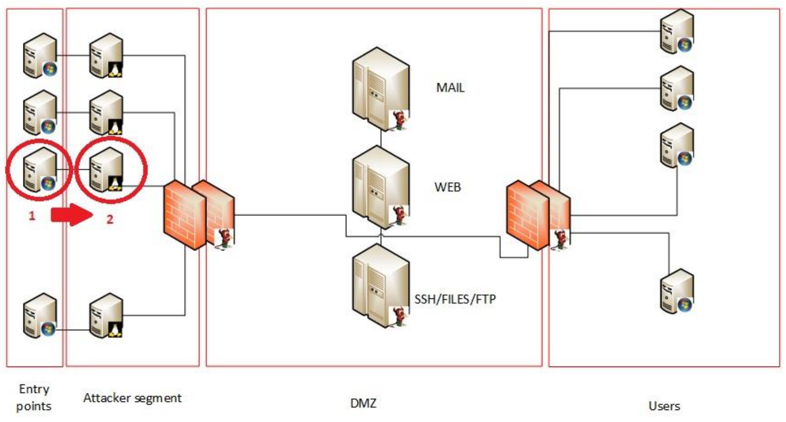


Figure 4.30 New vulnerability experiment infrastructure

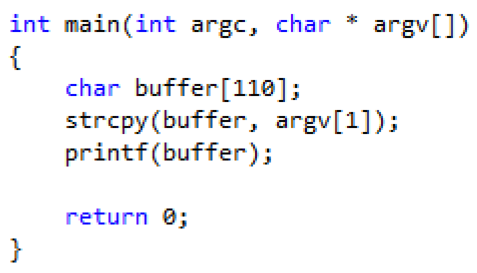
The attacker has an attack agent installed. The target has an environmental agent installed. The only difference is that the OS of the attack host is Windows 7, and the OS of the target host is Ubuntu Linux. The target machine has a vulnerable program residing there as well.

Instead of exploit execution, the attack agent tries to generate and send a string to the vulnerable program on the target host in order to overrun its buffer and rewrite the return address. The sequence of actions that the attacking agent should learn is:

- Add shellcode.
- Add the additional bytes needed in order to ensure that the desirable return address is shifted until it will not rewrite the old return address. For these reasons, NOP (No operation) operation was used.
- Add return address pointing to the shellcode.
- Send an input to the tested program.

Similar to the 'MDP experiment', a specific MDP was created in order to model that particular situation.

A simple vulnerable program, which the attacking agent tries to compromise, was created using C++ for the agent to attack (Figure 4.31). It resides on the target host.



```
int main(int argc, char * argv[])
{
    char buffer[110];
    strcpy(buffer, argv[1]);
    printf(buffer);

    return 0;
}
```

Figure 4.31 Program with stack overflow vulnerability

The program represents the vulnerable service working on the target side. It emulates the potential buffer overflow vulnerability that can be exploited. The penetration testing system should be able to identify such security breaches. As a result, it will learn the sequence of actions leading to the successful vulnerability exploitation.

The program reads the input parameter from the console and copies it to the buffer, using the vulnerable strcpy() function. The buffer has a particular size. Strcpy() does not check the length of the copied string. The audit agent does not have any information about the program, including the buffer size. It just knows that there is a program waiting for the input.

The audit agent tries to learn a correct sequence of data which will trigger the vulnerability. The environmental agent sends the string to the vulnerable program. It also restarts it in the case of a crash and checks if the shellcode was executed.

The audit agent uses Q-learning to combine (in a correct sequence) different shellcodes, a new return address (pointing at the shellcode), and the block of NOP operations.

Four different shellcodes were used for this purpose:

Shellcode1: Linux/x86-64: Connect Backdoor, date release: 14 Sep 2014.

```
‘\x31\xc0\x31\xd2\x31\xdb\x31\xc9\xb0\x02\xcd\x80\x83\xf8\x01\x7c\x02\xeb\
x62\x50\x6a\x01\x6a\x02\xb0\x66\xb3\x01\x89\xe1\xcd\x80\x89\xc3\x31\xc9\x
b0\x3f\xcd\x80\x41\x83\xf9\x04\x75\xf6\x68\x7f\x01\x01\x01\x66\x68\x1b\x39
\x66\x6a\x02\x89\xe1\x6a\x10\x51\x53\x89\xe1\xb0\x66\xcd\x80\x31\xc9\x29\x
c8\x75\x1b\xb0\x02\xcd\x80\x83\xf8\x01\x7c\x05\x31\xc0\x50\xeb\x0d\xb0\x0
b\xeb\x1f\x5e\x52\x56\x89\xe1\x89\xf3\xcd\x80\x31\xc0\xb0\x06\xcd\x80\xf3\x
90\x0f\x31\xf3\x90\xeb\x8b\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xdc\xff\xff\
xff\x2f\x62\x69\x6e\x2f\x62\x61\x73\x68’
```

Shellcode2: Linux/x86-64: Reads data from /etc/passwd, date release: 27 March 2014.

```
‘\x48\x31\xc0\xb0\x02\x48\x31\xff\xbb\x73\x77\x64\x00\x53\x48\xbb\x2f\x65\
x74\x63\x70\x61\x73\x53\x48\x8d\x3c\x24\x48\x31\xf6\x0f\x05\x48\x89\xc3\x
48\x31\xc0\x48\x89\xdf\x48\x89\xe6\x66\xba\xff\xff\x0f\x05\x49\x89\xc0\x48\
x89\xe0\x48\x31\xdb\x53\xbb\x66\x69\x6c\x65\x53\x48\xbb\x2f\x74\x6d\x70\x
6f\x75\x74\x53\x48\x89\xc3\x48\x31\xc0\xb0\x02\x48\x8d\x3c\x24\x48\x31\xf
6\x6a\x66\x66\x5e\x0f\x05\x48\x89\xc7\x48\x31\xc0\xb0\x01\x48\x8d\x33\x48\
x31\xd2\x4c\x89\xc2\x0f\x05’
```

Shellcode3: Linux/x86-64: console command execution (output to console), date release: 4 Oct 2015.

```
‘\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd\
x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f’
```

Shellcode4: Linux/x86-64: console command execution (output to console), date release: 14 Aug 2014.

```
‘\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd\
x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x61\x74\x74\x61\x63
\x6b’
```

The effectiveness of each shellcode is defined by two parameters: platform support and release date. The newer the release date, and the more platforms are supported, the more preferable is the shellcode. This information was used in the definition of the reward function in order to motivate the audit agent to use the most preferable shellcode. For this experiment, it is not important what the shellcode actually does. We are just interested in its successful execution, which is the indication of the vulnerability presence.

The next subsection defines an MDP for that particular example.

4.3.3 MDP process

The MDP state-action space and transitions were defined as shown in Table 4.3 and Table 4.4.

Table 4.3 MDP States

0	Initial state – The state from which the audit starts
1	SH1 – ‘Shellcode1: Linux/x86-64: Connect Backdoor’ was added
2	SH2 – ‘Shellcode2: Linux/x86-64: Reads data from /etc/passwd’ was added
3	SH3 – Shellcode3: Linux/x86-64: Shell echo command’ was added
4	SH4 – ‘Shellcode4: Linux/x86-64: Shell echo command’ was added
5	SH1_N1 – 223 bytes of \x90 (NOP) were added
6	SH1_N2 – 256 bytes of \x90 (NOP) were added
7	SH1_N3 – 138 bytes of \x90 (NOP) were added
8	SH1_N4 – 141 bytes of \x90 (NOP) were added
9	SH1_N5 – 115 bytes of \x90 (NOP) were added
10	SH2_N1 – 133 bytes of \x90 (NOP) were added
11	SH2_N2 – 57 bytes of \x90 (NOP) were added
12	SH2_N4 – 1 byte of \x90 (NOP) was added
13	SH2_N3 – 240 bytes of \x90 (NOP) were added
14	SH3_N1 – 123 bytes of \x90 (NOP) were added
15	SH3_N2 – 151 bytes of \x90 (NOP) were added
16	SH3_N3 – 205 bytes of \x90 (NOP) were added
17	SH3_N4 – 4 bytes of return address were added
18	Fail_terminal – Sequence failed
19	Suc_terminal – Sequence succeeded

Table 4.4 Actions

0	Add ‘Shellcode1: Linux/x86-64: Connect Backdoor’
1	Add ‘Shellcode2: Linux/x86-64: Reads data from /etc/passwd’
2	Add ‘Shellcode3: Linux/x86-64: Shell echo command’
3	Add ‘Shellcode4: Linux/x86-64: Shell echo command’
4	Add 223 bytes of \x90 (NOP)
5	Add 256 bytes of \x90 (NOP)
6	Add 138 bytes of \x90 (NOP)
7	Add 141 bytes of \x90 (NOP)
8	Add 115 bytes of \x90 (NOP)
9	Add 133 bytes of \x90 (NOP)
10	Add 57 bytes of \x90 (NOP)
11	Add 1 byte of \x90 (NOP)
12	Add 240 bytes of \x90 (NOP)
13	Add 123 bytes of \x90 (NOP)
14	Add 151 bytes of \x90 (NOP)
15	Add 205 bytes of \x90 (NOP)
16	Add 4 bytes of return address

Transitions for the first and second states are presented in Figure 4.32 as an example.

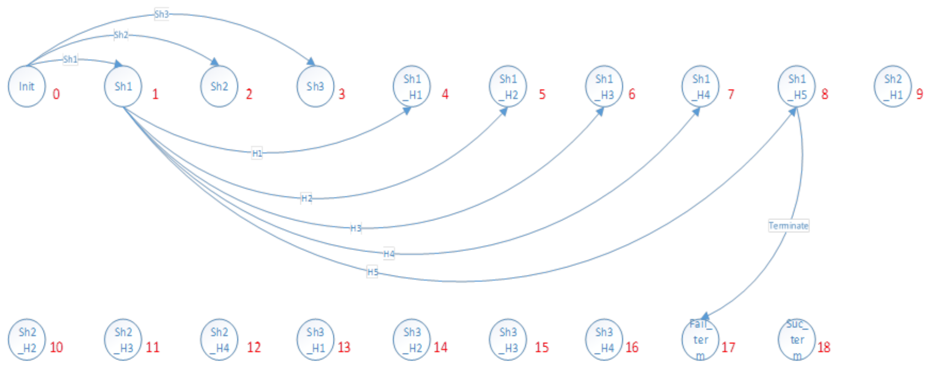


Figure 4.32 Transactions of new vulnerabilities MDP

4.3.4 Reward

The reward is derived directly from the environmental feedback and motivates the agent learning the successful attack strategy. The reward and transition functions are not predefined for every state-action. They are learned in the acting process. The task of the agent is to learn the sequence of actions guaranteeing the highest cumulative reward.

The reward is defined as a combination of main and additional rewards, R = R + R1 + R2, where the main reward is defined as R indicating successful execution of a shellcode, and the additional rewards are defined as follows:

R1 – Represents shellcode release date. The more recent the shellcode used, the bigger the reward that will be awarded.

R2 – Represents heterogeneity of the platforms where the shellcode can be executed. The more platforms are available for a particular shellcode, the higher the reward is.

4.3.5 Results assessment

The attack agent is able to learn a variety of attack sequences. They are formed by the combination of the shellcode, additional NOP bytes and the return address. The learning ability of the agent will be demonstrated if it will be able to learn the sequence of actions leading to the successful buffer overflow. There are solutions with different optimality that can be identified.

The optimal attacking sequence that the agent can learn is:

- Add Shellcode3
- Add 223 bytes of \x90 (NOP)
- Add return address pointing to the shellcode
- Send an input to the tested program

This solution is the optimal one because it accumulates the highest cumulative reward. It is related to the fact that Shellcode3 covers multiple platforms and is more relevant according to the release date.

The second solution that can be found by the attacking agent is:

- Add Shellcode4
- Add 240 bytes of \x90 (NOP)

- Add return address pointing to the shellcode
- Send an input to the tested program

This solution is less optimal because Shellcode4 is older than Shellcode3.

The performance of the learning algorithm is compared graphically by rewards accumulation during different acting policies. The higher the graphic curve of the cumulative reward rises with episode increase, the better the performance of the learning algorithm. The faster it grows, the faster an acting agent learns. The best hyper-parameters were used, which were identified during the MDP experiment and Field experiment (Gamma = 0.8, Alpha = 0.8).

The hypothesis of the current experiment will be demonstrated if the agent is able to learn a successful attack strategy. This strategy should result in a successful buffer overflow vulnerability exploitation of the target service.

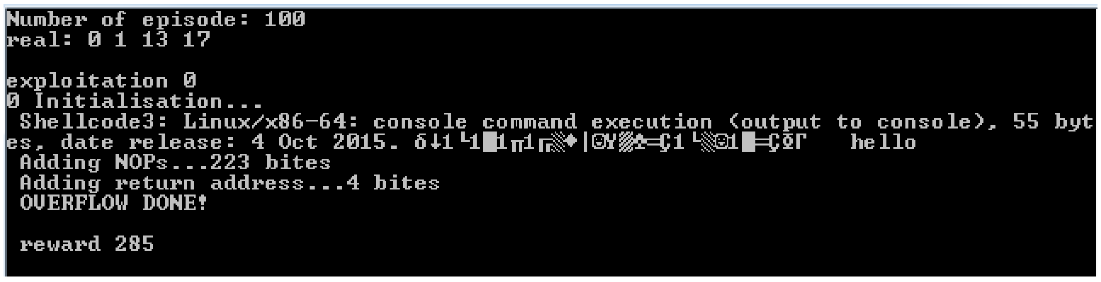
4.3.6 Experiment outcome

This subsection represents the results of the current experiment. As in a previous MDP experiment, the same action policies were applied:

- Random – Actions are chosen randomly.
- ε-greedy – Actions are chosen randomly or according to experience with 1-ε probability.
- Softmax – Actions are selected randomly but each action has a different probability of being chosen. This probability is calculated according to the experience of the learning agent. The higher the potential reward, the more probable it is that the action will be chosen.

4.3.6.1 Q-learning with random policy

Initially, Q-learning with random policy was used. The result is presented in Figure 4.33.



```
Number of episode: 100
real: 0 1 13 17

exploitation 0
0 Initialisation...
 Shellcode3: Linux/x86-64: console command execution (output to console), 55 byt
es, date release: 4 Oct 2015. δ↓1╘1█1╥1╓◈♦|☺Y▒☼═Ç1╘▒☺1█═Ç⁰Γ    hello
 Adding NOPs...223 bites
 Adding return address...4 bites
 OVERFLOW DONE!

 reward 285
```

Figure 4.33 Solution using random policy Q-learning 100 episodes

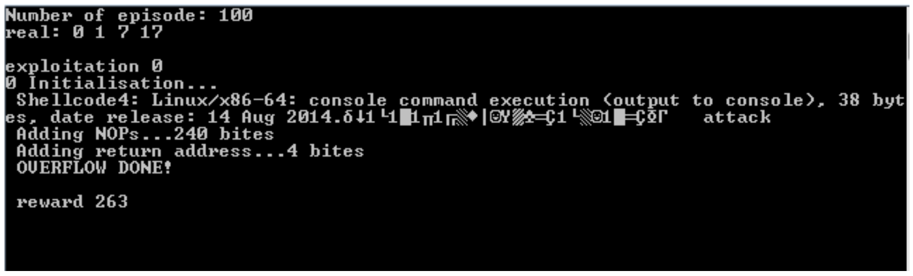
This output shows an MDP solution and cumulative reward. The solution was:

- Initialization
- Add 'Shellcode3: Linux/x86-64: Shell echo command'
- Add 223 bytes of \x90 (NOP)
- Add 4 bytes of return address
- Send combined string: Successful vulnerability exploitation

After only 100 episodes, the optimal solution was found. The later increase of the episodes did not change the solution.

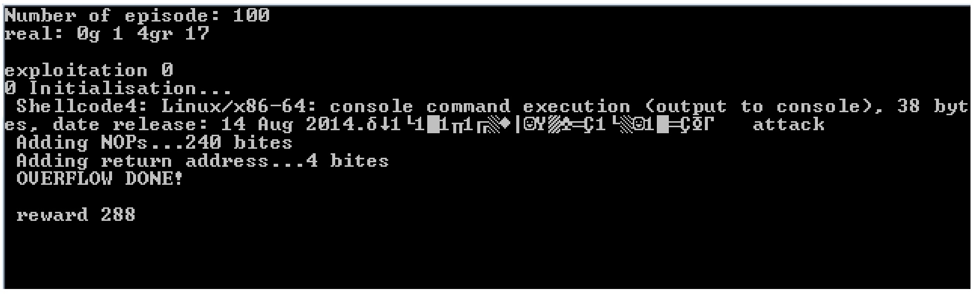
4.3.6.2 Q-learning with ε-greedy policy

At the next stage of the experiment, Q-learning with ε-greedy policy was applied. The MDP solution after 100 episodes using Gamma=0.8, Alpha=0.8, 1-ε=0.15/0.55/0.98 is shown in Figure 4.34, Figure 4.35 and Figure 4.36.



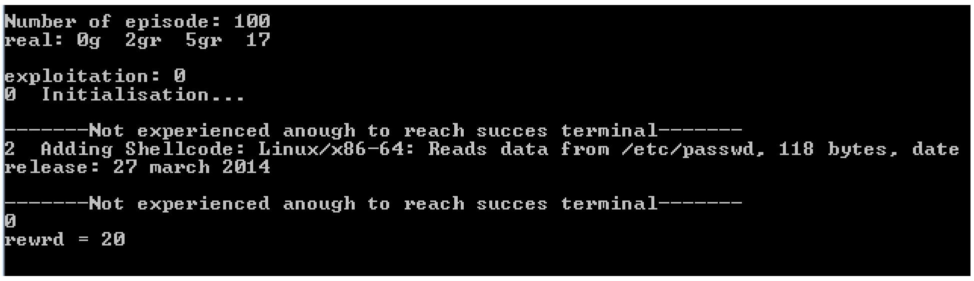
```
Number of episode: 100
real: 0 1 7 17

exploitation 0
0 Initialisation...
 Shellcode4: Linux/x86-64: console command execution (output to console), 38 byt
es, date release: 14 Aug 2014.δ↓1╘1█1╥1╓◈♦|☺Y▒☼═Ç1╘▒☺1█═Ç⁰Γ    attack
 Adding NOPs...240 bites
 Adding return address...4 bites
 OVERFLOW DONE!

 reward 263
```

Figure 4.34 MDP solution using ε-greedy policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.15



```
Number of episode: 100
real: 0g 1 4gr 17

exploitation 0
0 Initialisation...
 Shellcode4: Linux/x86-64: console command execution (output to console), 38 byt
es, date release: 14 Aug 2014.δ↓1└1█1╥1╓▒♦|☺Y▒≐═Ç1└▒☺1█═ÇΩΓ    attack
 Adding NOPs...240 bites
 Adding return address...4 bites
 OVERFLOW DONE!

 reward 288
```

Figure 4.35 Solution using ε-greedy policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.55



```
Number of episode: 100
real: 0g  2gr  5gr  17

exploitation: 0
0  Initialisation...

-------Not experienced anough to reach succes terminal-------
2  Adding Shellcode: Linux/x86-64: Reads data from /etc/passwd, 118 bytes, date
release: 27 march 2014

-------Not experienced anough to reach succes terminal-------
0
rewrd = 20
```

Figure 4.36 Solution using ε-greedy policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8, 1-ε=0.85

Surprisingly, the agent with 1-ε=0.15 did not find the optimal solution since the shellcode was outdated:

- Initialization
- Add Shellcode4
- Add 240 bytes of \x90 (NOP)
- Add return address pointing to the shellcode
- Send an input to the tested program

After 1-ε was increased to 55, the solution was still not the optimal one:

- Initialization
- Add Shellcode4
- Add 240 bytes of \x90 (NOP)
- Add return address pointing to the shellcode
- Send an input to the tested program

Only after the quantity of episodes was increased to 1000 did the solution converge to the optimal one:

- Initialization
- Add ‘Shellcode3: Linux/x86-64: Shell echo command’
- Add 223 bytes of \x90 (NOP)
- Add 4 bytes of return address
- Send combined string: Successful vulnerability exploitation

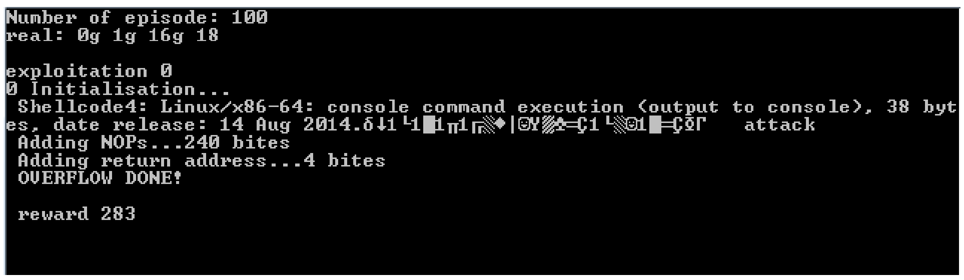
When 1-ε was set up to 0.98, the agent was not able to find the solution at all after 100 episodes. Even after 1000 episodes, the solution was still not optimal. Only a further increase to 5000 guaranteed the convergence to the most effective solution:

- Initialization
- Add ‘Shellcode3: Linux/x86-64: Shell echo command’
- Add 223 bytes of \x90 (NOP)
- Add 4 bytes of return address
- Send combined string: Successful vulnerability exploitation

The next section shows the application of the Q-learning algorithm using softmax acting policy.

4.3.6.3 *Q-learning with softmax policy*

In the current stage of the experiment, Q-learning with softmax policy was applied. The solution after 100 episodes with Gamma=0.8, Alpha=0.8 is shown in Figure 4.37.



```
Number of episode: 100
real: 0g 1g 16g 18

exploitation 0
0 Initialisation...
 Shellcode4: Linux/x86-64: console command execution (output to console), 38 byt
es, date release: 14 Aug 2014.δ↓1└1█1╥1╓▒♦|☺Y▒☼═Ç1└▒☺1█═ÇΩΓ    attack
 Adding NOPs...240 bites
 Adding return address...4 bites
 OVERFLOW DONE!

 reward 283
```

Figure 4.37 Solution using softmax policy Q-learning 100 episodes: Gamma=0.8, Alpha=0.8

The optimal solution was not found straight away after 100 episodes. However, a less optimal sequence of actions was identified:

- Initialization
- Add ‘Shellcode4: Linux/x86-64: Shell echo command’
- Add 240 bytes of \x90 (NOP)
- Add 4 bytes of return address
- Send combined string: Successful vulnerability exploitation

After the further increase of episodes to 1000, the solution was optimal:

- Initialization
- Add ‘Shellcode3: Linux/x86-64: Shell echo command’
- Add 223 bytes of \x90 (NOP)
- Add 4 bytes of return address
- Send combined string: Successful vulnerability exploitation

Next section describes the learning algorithm performance related experiments in regards of different parameters and acting policies.

4.3.7 Learning performance

The performance of the learning algorithm depending on the parameters and policies is presented below.

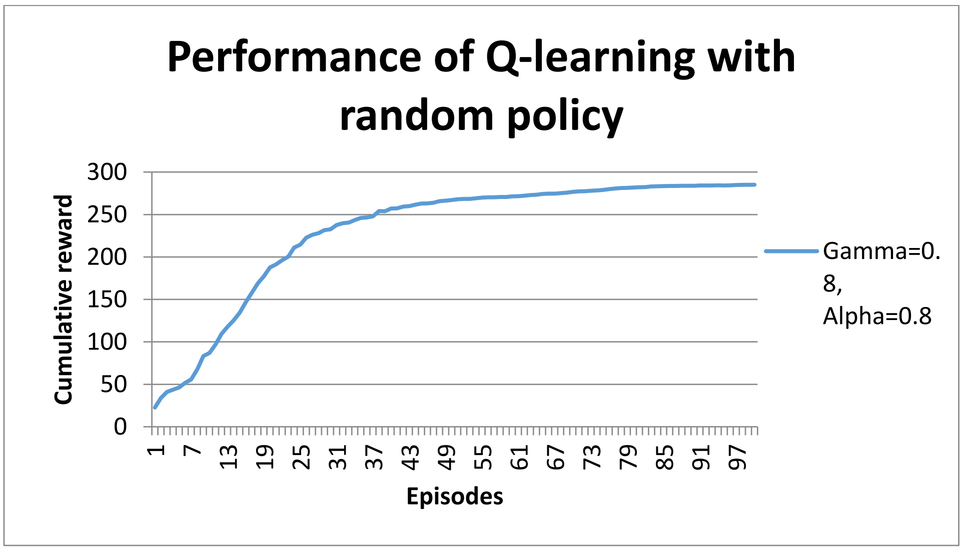


Figure 4.38 Random policy Q-learning performance with Gamma=0.2

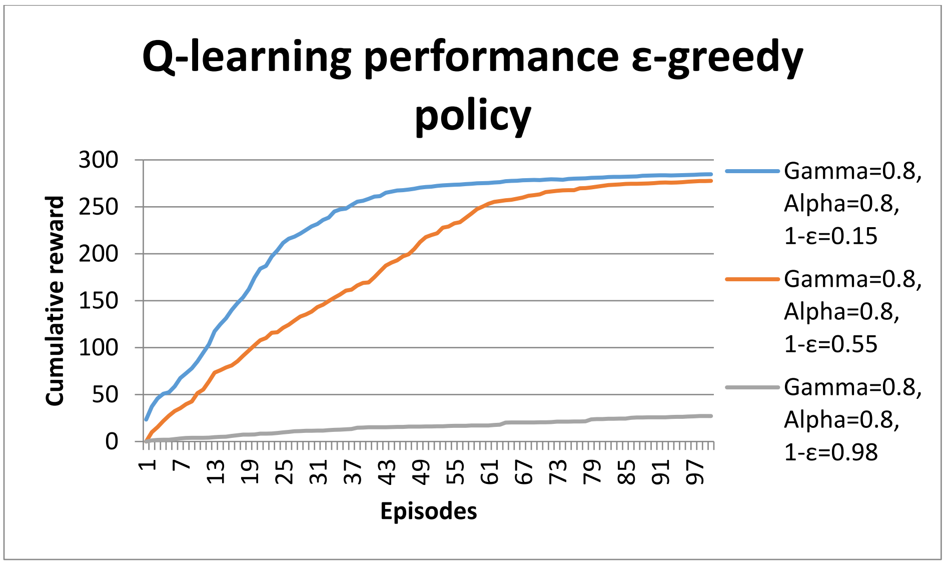


Figure 4.39 ε-greedy Q-learning performance with Gamma=0.8, Alpha=0.8, 1-ε=0.15/0.55/0.98

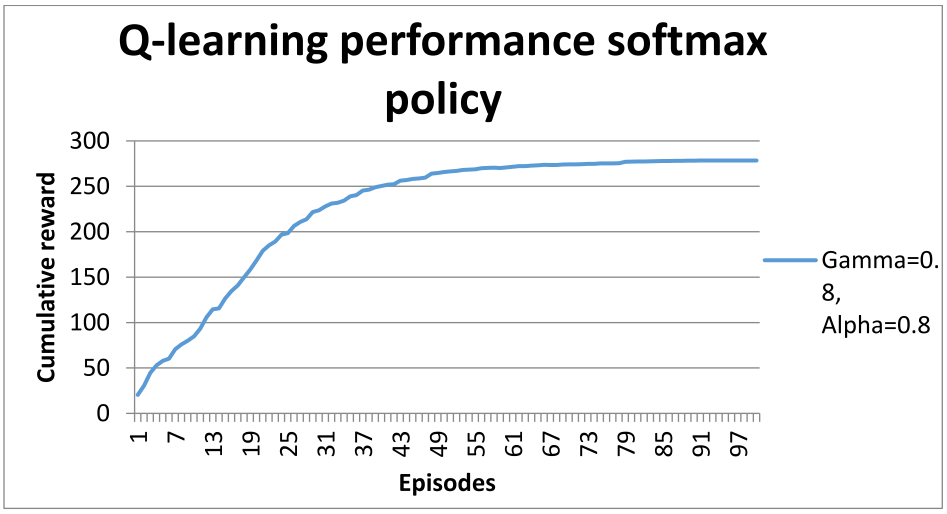


Figure 4.40 Softmax policy Q-learning performance with Gamma=0.8, Alpha=0.8

The performance of the Q-learning algorithm with random policy was tested with the best values of Gamma and Alpha. In Figure 4.38, the optimal values of Gamma=0.8 and Alpha=0.8 are presented. The quantity of episodes is 100 averaged by 100 tests. The curve is smooth and converges to the cumulative reward value of 285. The next experiment was performed with ε-greedy policy.

The performance of the Q-learning algorithm with ε-greedy policy is presented in Figure 4.39, where Gamma=0.8, Alpha=0.8 and 1-ε=0.15/55/98. All the curves smoothly converge to optimality. However, the best performance was shown with 1-ε=0.15. The curve rises faster than the others. It demonstrates a faster learning process and converges to the highest value of cumulative reward. It indicates that the algorithm with such a hyper-parameter for ε-greedy policy guarantees the most effective learning process.

The performance of Q-learning with softmax policy with Gamma=0.8 and Alpha=0.8 is presented in Figure 4.40. The curve smoothly converges to the optimal value of 278. It rises quite fast at almost the same speed as that of the curve representing the most effective ε-greedy policy. The algorithm with those two policies converges faster to the optimal solution. It reaches the highest cumulative reward value as well.

4.3.8 Conclusion

The hypothesis that the penetration testing system is able to learn a vulnerability exploitation which it has never experienced before was demonstrated for the particular example of buffer overflow. The multistep sequence of actions was found. The algorithm was able to learn the optimal attack strategy without any predefined knowledge about transition and reward function being provided, which describes all transitions and rewards from all states. Such knowledge was learned by the agent while acting.

The performance of Q-learning in this experiment showed that the best acting policy was ε-greedy with 1- ε = 0.15. The best convergence to the optimality was demonstrated with learning rate = 0.8, discount factor = 0.8.

The current experiment validated the model-free approach to the new vulnerabilities discovery process, in particular buffer overflow. The experiment successfully proved the initial hypothesis. In order to make the experiment more realistic, it requires further generalization, which is considered to be future work. This experiment is an initial research towards a self-updating autonomous penetration testing system, which can update its functionality with new exploits by itself. It also covers the problem of new vulnerabilities search which has not yet been covered by exploits. The research showed sufficient potential to make the penetration testing tool significantly autonomous and self-updating.

The next experiment represents the generalization of the Field experiment, which is a logical extension of the MDP experiment. It should prove the model-free learning approach in a general case scenario.

4.4 Deep Architecture ANN experiment

4.4.1 Hypothesis

This group of experiments is arranged in order to generalize a model-free learning approach to the general penetration testing case scenario. It is intended to demonstrate the hypothesis that the Q-learning algorithm with an approximator, based on an artificial neural network, will be able to learn attack strategies on the go and successfully attack targets it has not previously experienced. The agent derives the reward directly from the environmental feedback while acting.

4.4.2 Experiment setup

The experiment uses the testing infrastructure described in the Environment architecture section of the Methodology chapter. It is shown in Figure 4.41.

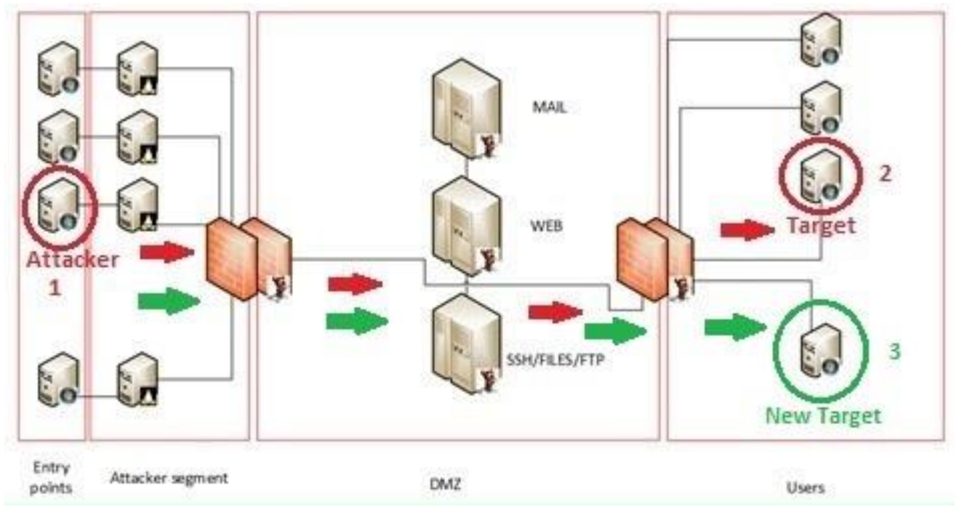


Figure 4.41 The experiment setup

Attacker host 1 represents the penetration testing system with an attacker agent installed. Target host 2 is the first goal to attack. The second goal is the New Target 3 host. Both target hosts have feedback agents installed in order to provide the environmental feedback to the penetration testing system. All agents communicate via sockets. Both target hosts have remote code execution and privilege escalation vulnerabilities. Their configurations are similar to each other but not identical. Both have Windows 7 OS installed and a similar set of processes.

The experiment is intended to generalize the particular case of the Field experiment. It has two stages. During the first stage, the penetration testing system attacks Target host 2 in order to learn the attack strategy. 'Attack strategy' means the sequence of actions leading to the successful penetration for Target 2.

The learning process is different to that of the Field experiment. Instead of acting according to the Q-values table, the learning algorithm derives Q-values using the neural network as an approximator. An artificial neural network approximates the Q-value function. It uses features representing the target host as an input, and returns a Q-value for the current state/action pair as an output. Features representation of the states are described in the Feature extraction section. Features are extracted using an autoencoder. It allows the algorithm to efficiently learn attack strategies in the case of huge state spaces.

The autoencoder configuration is presented in Table 4.5.

Table 4.5 Parameters for autoencoder

Variables	Values	Comments
Layers:	1	Quantity of hidden layers
Neurons per layer:	285	Quantity of neurons in hidden layers
Activation function:	sigmoid	Activation function
Epsilon:	0.12	Random weights initialization parameter
Lambda1:	0.003	Weight decay parameter
p:	0.1	Sparsity parameter
Beta:	3	Weight of sparsity penalty term
Lambda2:	0.02	Learning rate
Iterations:	500	Learning iterations

In order to make the penetration system learn the attack sequence (using the ANN approximator) against the target host configuration, defined by features, a deep learning architecture was developed. It consists of two neural networks combined: the autoencoder and the ANN. When the system experiences a new target host, features are collected. The features description and collection process is presented in the section Autoencoder. The feature set is used as an input for the autoencoder. It decreases the features space by processing input information. As a result, the smaller feature set (which is the set of hidden layer neurons of the autoencoder), joined with the action vector, is fed to the approximator ANN as an input. The output of the approximator is an estimated Q-value. After the environment agent sends the reward to the attacking agent, the real Q-value is calculated. As a result, the error between the estimated and real Q-values is determined and the approximator readjusts its weights.

The optimal parameters for Q-learning, which were revealed during the MDP experiment/Field experiment, were used. The performance of the modified Q-learning algorithm is tested against random, ε-greedy and softmax acting policies.

The second phase of the experiment consists of attacking New Target 3 host, using the experience learned during the attack of the Target host 2. The penetration testing system attacks the host, which it has never experienced before. The only knowledge it has is the feature set representing the configuration of the new target. This feature set was used as an input to the approximator in order to get the sequence of actions leading to the successful penetration for the New Target host.

The experiment demonstrates the general case scenario for the smart penetration testing system. It accumulates the attack experience during the first stage by attacking Target host 2. Afterwards, it uses its experience in order to successfully attack the New Target host 3, which it never experienced before. In other words, the final target and the host providing the knowledge for the system are different. Therefore, the system learns the attack behavior and demonstrates successful penetration testing activity without a human interaction.

4.4.3 MDP process

The representation of the penetration testing process is generalized for huge state space. It means that instead of a fixed number of states, their general representation is defined by the feature set. The feature set is related to the definition of the configuration of the Target host. ‘Configuration’ means the combination of processes and dlls used by the host. More details on features description can be found in the Autoencoder section of the Methodology chapter.

The actions of the MDP process are defined in Table 4.6.

Table 4.6 MDP Actions

0.	OS_Win_Check – Checks if OS is Windows 7
1.	Ex_1_remote – Execute remote Exploit 1 (user privileges, Windows 7)
2.	Ex_2_remote – Execute remote Exploit 2 (admin privileges, Windows XP)

3.	Ex_3_remote – Execute remote Exploit 3 (admin privileges, Windows 7)
4.	Ex_4_local – Execute Local exploit 4 (escalate privileges)

The transition function is not defined for every state and is learning on the go while acting. The reward function is not defined for every state-action as well and is described in the next section.

4.4.4 Reward

A reward is the numerical value received by the agent after attack/probe action was executed. The reward is received directly from the environmental feedback. It is the mechanism motivating the agent to learn the successful sequence of actions leading to the successful penetration to the target host. The task of the agent is to learn the attack strategy guaranteeing the highest cumulative reward.

The reward is defined as follows:

$$R = R + R1 + R2 + R3$$

where the main reward is defined as R. This reward indicates the successful penetration for the agent. In cases where the agent got admin rights on a target host, it receives 100 as a main reward, 0 otherwise:

R1 – Represents exploit effectiveness according to the metasploit rating (great, good and so on). The more effective the exploit, the higher the reward.

R2 – Represents the escalation of privileges bonus. The higher reward is given for remote penetration with admin privileges contrary to the user privileges only.

R3 – Additional reward for testing actions (portscan, OS detection and so on), defined as 20/n (where n is the number of iterations per episode). This reward motivates an agent to explore the environment and reduces the state space before exploitation.

4.4.5 Results assessment

The setup of the experiment provides four exploits available for the attacking agent: Remote exploit 1 (user privileges, Windows 7), Remote exploit 3 (admin privileges, Windows 7), Remote exploit 2 (admin privileges, Windows XP) and Local exploit 4 for privilege escalation for Windows 7. The first exploit is remote and guarantees user privileges on the target host. The second exploit provides admin privileges on the target host with remote access. Windows XP exploit won't work on the Target. Local exploit escalates the privileges locally. Exploits differ according to the reliability rating and potential harm for the target system. Remote Exploit 3 (admin privileges, Windows 7) is more reliable and less harmful. It means that an agent receives more reward when it successfully uses Exploit 3 compared to Exploit 1. The agent also has probe actions available. The combination of actions leading to successful penetration defines the attack strategy. A variety of attack strategies differs by optimality (the quantity of reward accumulated by an agent during its execution).

Attack strategies that can be found are presented below and sorted by optimality:

1. Probe action + execute a remote exploit 3 with admin privileges.
2. Execute a remote exploit 3 with admin privileges.
3. Probe action + execute a remote exploit 2 with user privileges + execute local exploit.
4. Execute a remote exploit 2 with user privileges + execute local exploit.
5. The rest of the actions leading to Fail state.

The optimal attack strategy is the first one. The optimality of the other solutions decreases to the end of the list. The performance of the learning algorithm is defined by cumulative rewards value and measured graphically. The higher it is and the faster it grows, the more effective the learning process is.

The attack agent should not just learn the attack strategy but it should be able to successfully apply the attack strategy to the new, never experienced target.

4.4.6 Experiment outcome

This subchapter represents the actual results of the described experimental setup. The experiment was based on deep learning architecture, consisting of two parts connected sequentially: autoencoder and state space ANN approximator. The autoencoder extracts the feature set defining the state of the target host. The ANN approximator uses this information in order to make the attacking agent learn the attack strategy. Therefore the description of the experiment is divided into two parts. The feature extraction section describes the group of experiments related to the learning process of the autoencoder with different values of hyper parameters. It demonstrates the feature extraction process. The second part of the experiment represents the description of the actual Q-learning process with the ANN approximator. There were different acting policies used during the learning process:

- random – actions are chosen randomly.
- ε-greedy – actions are chosen randomly or according to experience depending on parameter ε.
- softmax – actions are selected randomly but each action has a different probability of being chosen. This probability is calculated according to the experience of the learning agent. The higher the potential reward, the more probable the action will be chosen.

4.4.6.1 Feature extraction

The first part of the current experiment is related to the features extraction process. This is based on autoencoder architecture. The features representation and architecture itself is described in the Autoencoder section. The experiment is to make the autoencoder learn features representation. The autoencoder receives the list of processes and dlls of the target host as an input. It encodes the features into its smaller hidden layer and decodes it back to the output layer. After this process, the error is calculated as a difference between the input and output and the weights of the autoencoder are corrected. After a number of iterations the error becomes minimal and the hidden layer can be used as a new decreased feature set. The feature extraction experiment consists of testing the performance of the learning process of the autoencoder using different hyper parameters. The learning process is defined by the error minimisation procedure and compared graphically.

4.4.6.1.1. Autoencoder learning performance

This section shows the performance of the autoencoder learning process according to different numbers of iterations. Decoding error graphs are presented below.

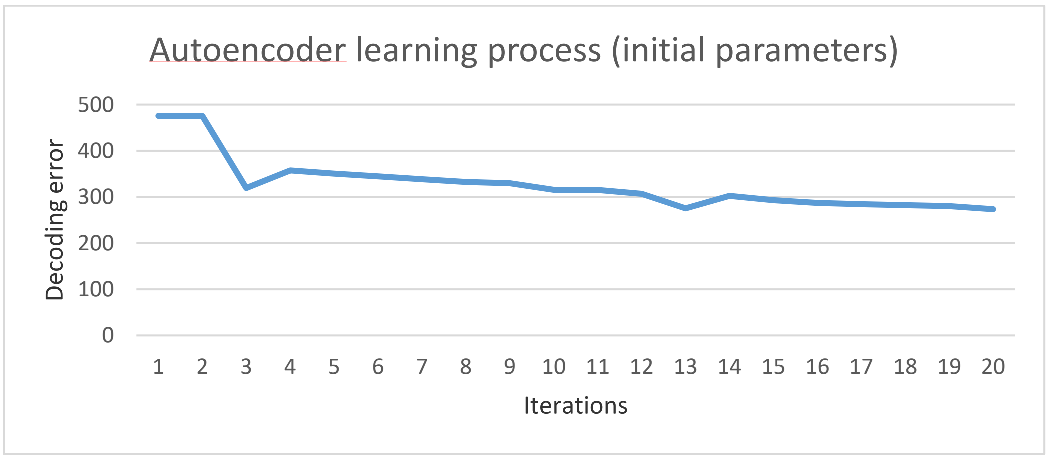


Figure 4.42 Autoencoder learning process using initial parameters

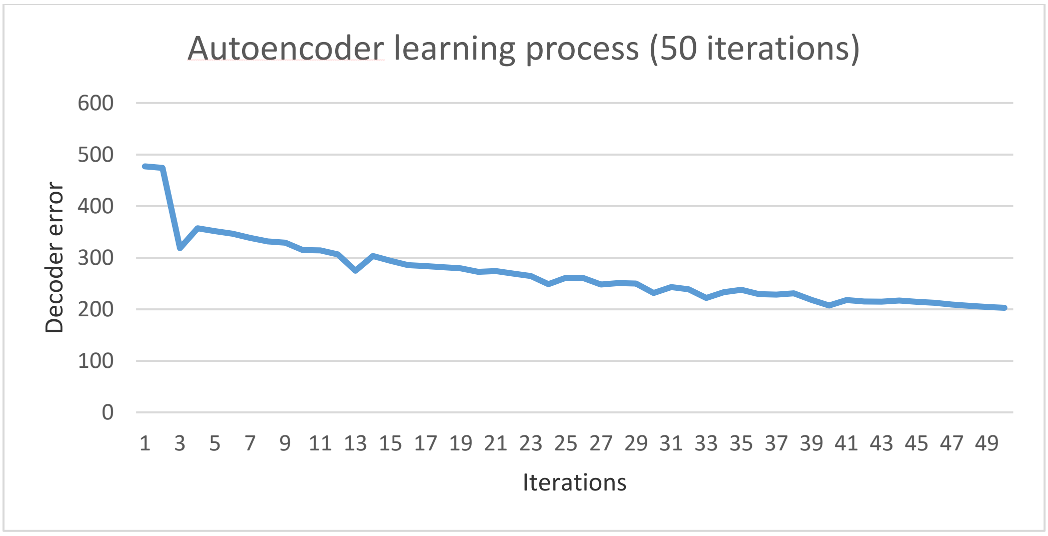


Figure 4.43 Autoencoder learning process after 50 iterations

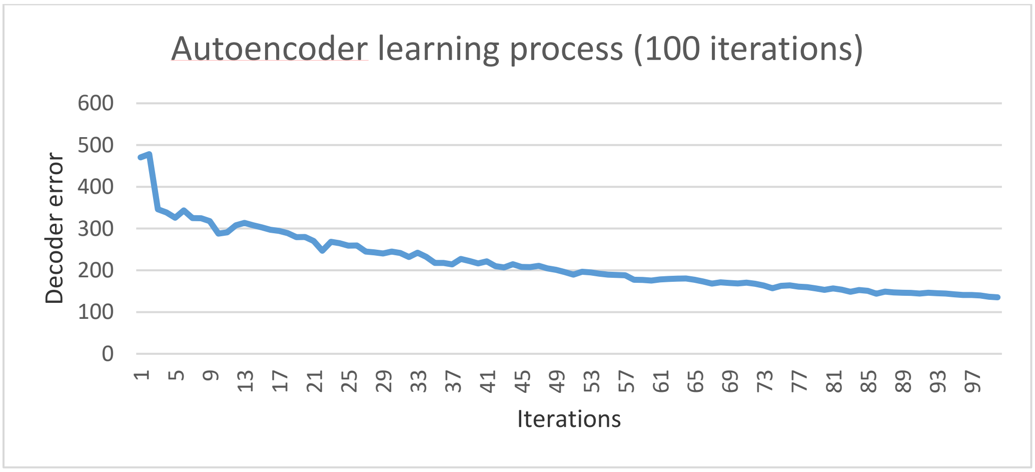


Figure 4.44 Autoencoder learning process after 100 iterations

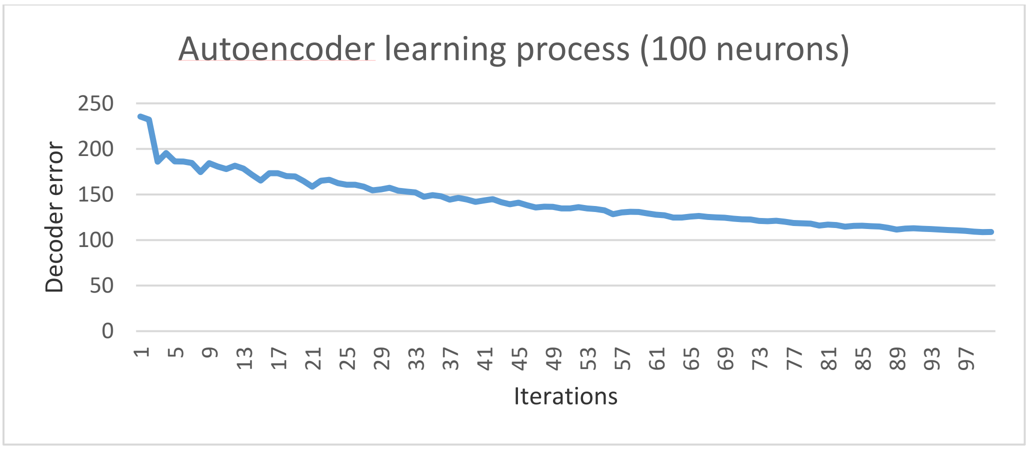


Figure 4.45 Autoencoder learning process (100 neurons per hidden layer and 100 interations)

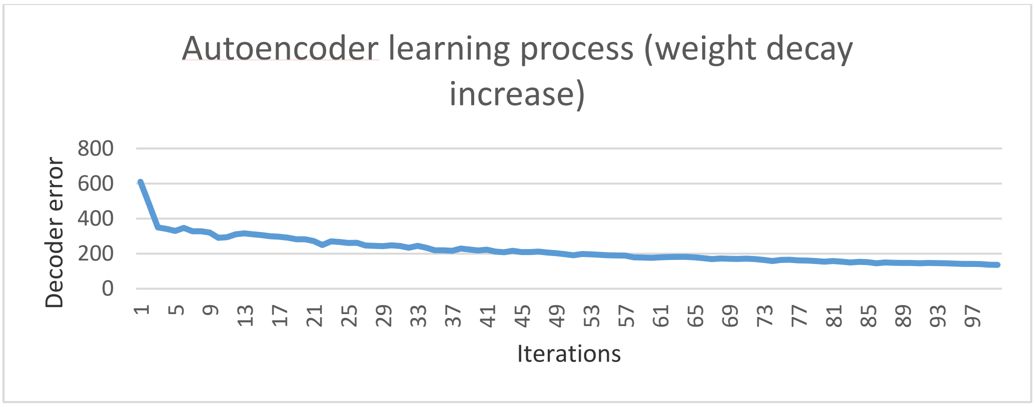


Figure 4.46 Autoencoder learning process (λ1 = 0.1)

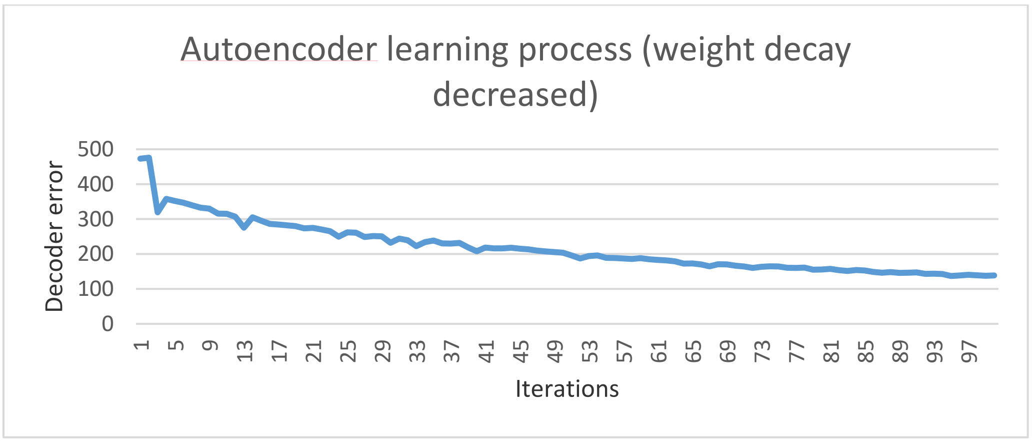


Figure 4.47 Autoencoder learning process (λ1 = 0.00003)

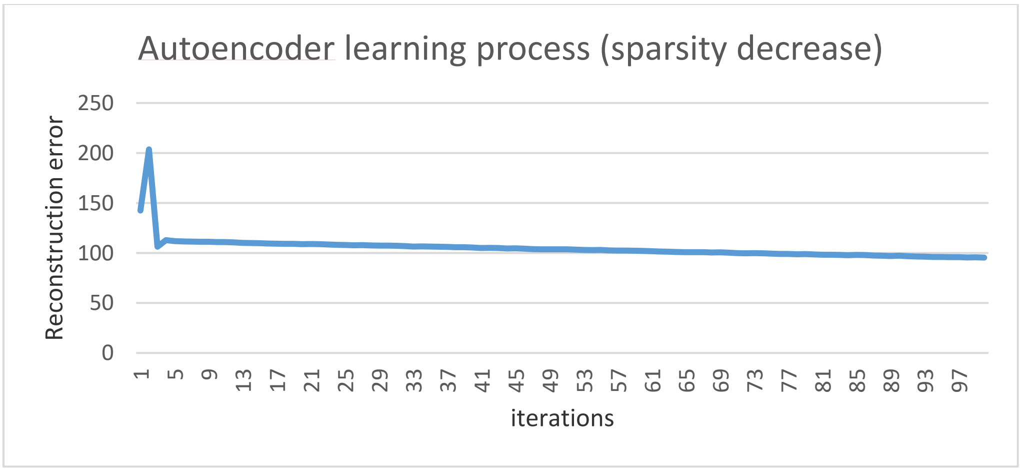


Figure 4.48 Autoencoder learning process (β = 0.1)

The first number of iterations was set up to 20. The result of decoding error minimization is presented in . It can be seen that 20 iterations are not enough. The autoencoder did not minimize the decoding error much. It equals 273, which is too large. The quantity of iterations was increased to 50 (). It is noticeable that the curve becomes smoother. The learning error was decreased from 273 to 200. This trend was expected, since the accuracy of weights in the autoencoder increases with the quantity of iterations. represents the learning error after 100 iterations. The curve has become quite smooth. The learning error has been decreased to 136. This is the optimal value in order to have a balance between the accuracy of the feature set and the speed of the extraction process.

The other hyper-parameter to test is the number of neurons per hidden layer. Previous experiments related to the iteration quantity were performed with 300 neurons per hidden layer. In order to decrease the error even more, the number of neurons was set to 100. The result of features extraction at this number is shown in . Decreasing the quantity of neurons leads to the minimization of learning errors to 108. However, during this process, the quantity of information, which is needed for reliable features representation also decreases. During experiments, it was found that 285 is an optimal quantity of neurons to extract features for the Q-learning ANN approximator.

The next parameter to test is the weight decay λ1 (regularization term). All previous feature extraction experiments were performed using λ1 = 3e-3. For the current case, the weight decay was significantly increased to 0.1. This parameter prevents the weights of the autoencoder from growing too large. The reaction of the decoder error is presented in . As a result, the learning error increased to 136 against the smaller weight decay case of 108. In order to test the small weight decay case, λ1 was significantly decreased to 0.00003 (). The learning error is increased compared to the average weight decay value of 138. Therefore, the optimal value for λ1 (weight decay) was found to be 0.003.

The next hyper-parameter for the autoencoder to test is the sparsity penalty term. This parameter is responsible for switching off some of the neurons in a hidden layer. It is used when the quantity of hidden layer neurons is larger than the quantity of input neurons. For all previous cases, the sparsity penalty term was set up to 3. This parameter was decreased to 0.1 (). As a result, the error was decreased to 100. However, this representation of features caused problems in the Q-learning algorithm learning process. The optimal value for sparsity penalty term was found to be 3.

Therefore, the optimal hyper-parameters for the autoencoder were found: 285 neurons per hidden layer, λ1 (weight decay) = 0.003, β (sparsity penalty term) = 3. The next three subsections show the results of the penetration testing system's learning, using different action policies. The learning process is based on deep architecture consisting of autoencoder and ANN. All hyper-parameters of both neural networks are optimal, according to the previous experiments. These action policies were tested: random policy, ε-greedy policy and softmax policy.

4.4.6.2 *Q-learning with random policy*

During this experiment, Q-learning with random policy was applied. The solution with Gamma=0.8, Alpha=0.8 is shown in Figure 4.49.

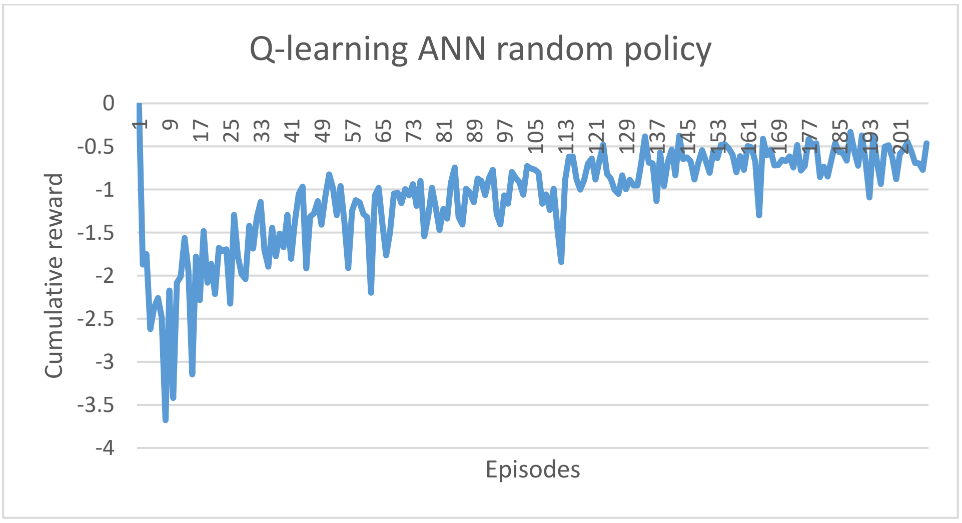


Figure 4.49 Q-learning ANN with random policy

The cumulative reward converges quite slowly to -0.664. The graph has lots of small peaks. This is caused by the neural network approximator. The optimal solution was not found after almost 200 iterations. Eventually, penetration was successful but the action sequence was less optimal:

- Initialization.
- Execute a remote exploit with admin privileges.

The next subsection represents the results of learning using the ε-greedy acting policy.

4.4.6.3 *Q-learning with ε-greedy policy*

In this experiment, an ε-greedy policy was used. The results are shown in Figure 4.50 (Gamma=0.8, Alpha=0.8, ε=0.15).

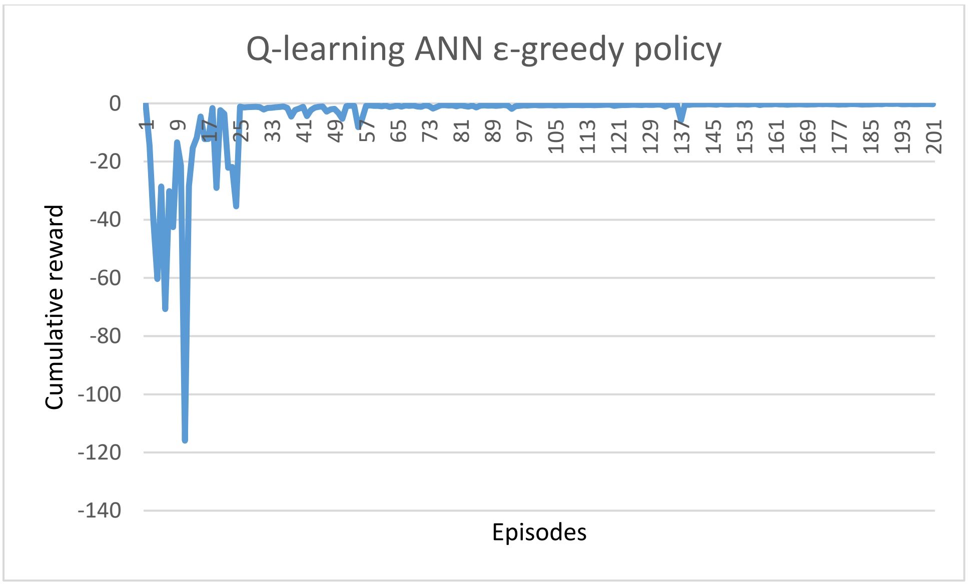


Figure 4.50 Q-learning ANN with ε-greedy policy

As a result, the cumulative reward converges faster than in the random policy case andthe value itself is higher and equals -0.548. Higher cumulative rewards indicate the better solution. This value is the highest among the policies. The attack strategy that was found during this acting policy was optimal:

- Initialization.
- Probe action (check OS).
- Execute the remote exploit with admin privileges.

This sequence of actions is better compared to the random policy case. This is related to the fact that a more effective exploit chain was found.

The next acting policy which was tested was the softmax policy.

4.4.6.4 Q-learning with softmax policy

The next experiment is based on the same optimal values for gamma and alpha, but the acting policy was changed to softmax (Figure 4.51).

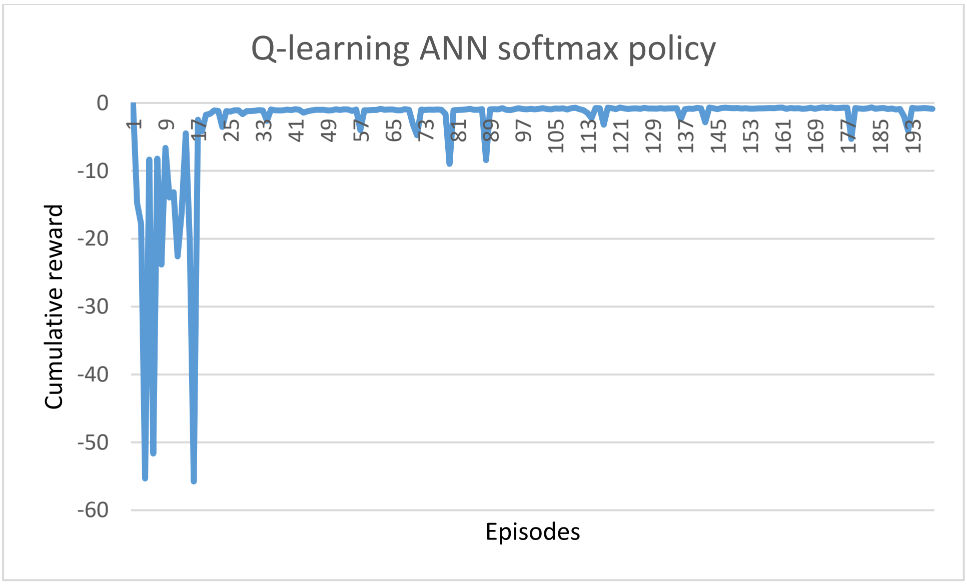


Figure 4.51 Q-learning ANN with softmax policy

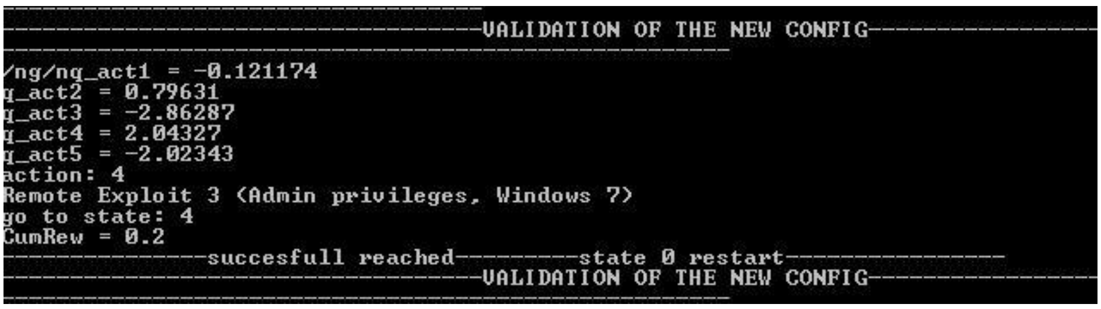
The cumulative reward converges to -0.784. This value is lower than ε-greedy policy. The agent learns slower as well. As in random acting policy case, the optimal solution was not found:

- Initialization.
- Execute the remote exploit with admin privileges.

The next experiment described in the Deep architecture RNN section is related to the usage of a different approximator for the Q-learning algorithm.

4.4.6.5 Previously unexperienced target attack

This experiment represents the attack of previously learned agent against a new host, which has not been experienced before. After the attacking agent has learned the successful attack sequence on the Target 2, it attacks New Target 3. It acts according its own experience gathered during learning stage using ε-greedyacting policy. This policy was used as it provides the best performance for the learning algorithm. It is demonstrated in Q-learning with ε-greedy policy experiment. The agent extracted the feature set representing the configuration of the New target 3. This feature set was used as an input to the approximator in order to get the sequence of actions leading to successful penetration. The solution is presented in Figure *4.52*.



```
----------------------------------------
-------------------------------------VALIDATION OF THE NEW CONFIG------------------
------------------------------------------------------
/ng/nq_act1 = -0.121174
q_act2 = 0.79631
q_act3 = -2.86287
q_act4 = 2.04327
q_act5 = -2.02343
action: 4
Remote Exploit 3 (Admin privileges, Windows 7)
go to state: 4
CumRew = 0.2
---------------succesfull reached---------state 0 restart-----------------
-------------------------------------VALIDATION OF THE NEW CONFIG------------------
------------------------------------------------------
```

Figure 4.52: The solution for previously unexperienced target attack

The cumulative reward has the value 0.2. A solution was found:

- Initialization.
- Execute the remote exploit with admin privileges.
- Successful penetration

The sequence of actions leading to the successful penetration is not the optimal one. However it is effective to get admin privileges on the remote host.

The agent was able to penetrate a new host, with which it had never interacted before. It used an experience accumulated during the previous learning iterations while attacking another host.

4.4.7 Conclusion

The experiment proved the hypothesis that the penetration testing system was able to learn attack strategies on the go by applying the Q-learning algorithm with an artificial neural network as approximator. This group of experiments extended the Field experiment to the general case scenario and demonstrated the ability of the agent to successfully function in much bigger state space.

The agent learned the attack strategy trying to penetrate to the Target host 2. It was able to identify a New Target host 3, never experienced before, as being similar to the Target host 2 and applied its experience to attack it. As a result, it successfully penetrated the New Target host 3. During the learning phase different attack policies were used. The best acting policy was ε-greedy.

The experiment demonstrated the room for improvement. The efficiency of the approximator can be increased by applying a different neural network architecture.

The experiment showed the successful penetration to the new, previously unknown host, after learning similar configuration. The system learned the attack behavior and demonstrated successful penetration testing activity without a human interaction.

However, in real life scenarios it can be the situation that the configuration of the host is updated after an agent was learned. It might be still similar to the configurations the agent experienced earlier, but a number of vulnerabilities could be patched. It might lead to penetration failure, since the learned attack sequence will no longer work.

Similar problem can appear when the agent will attack a new configuration, completely different from the ones it has experienced before. In this case it will fail to successfully penetrate the host because of lack of knowledge.

Thus, there is the need for the agent to be adaptive. It should be able to restart the learning process in case of failure, to find a new successful attack strategy. This process will readjust the approximator and will adapt the agent to the new host configuration, while keeping previously accumulated knowledge.

Deep architecture RNN experiment addresses those issues.

4.5 Deep architecture RNN experiment

4.5.1 Hypothesis

This group of experiments is intended to prove the ability of the penetration testing system to learn attack strategies on the go by applying the Q-learning algorithm with approximator, based on recurrent neural network. It also should be able to successfully attack previously unknown targets and adapt to its configuration changes. ‘Adaptation’ means the relearning process and identification of a new successful attack strategy.

4.5.2 Experiment setup

The experiment uses the testing infrastructure described in the Environment architecture section of the Methodology chapter. It is shown in Figure 4.53.

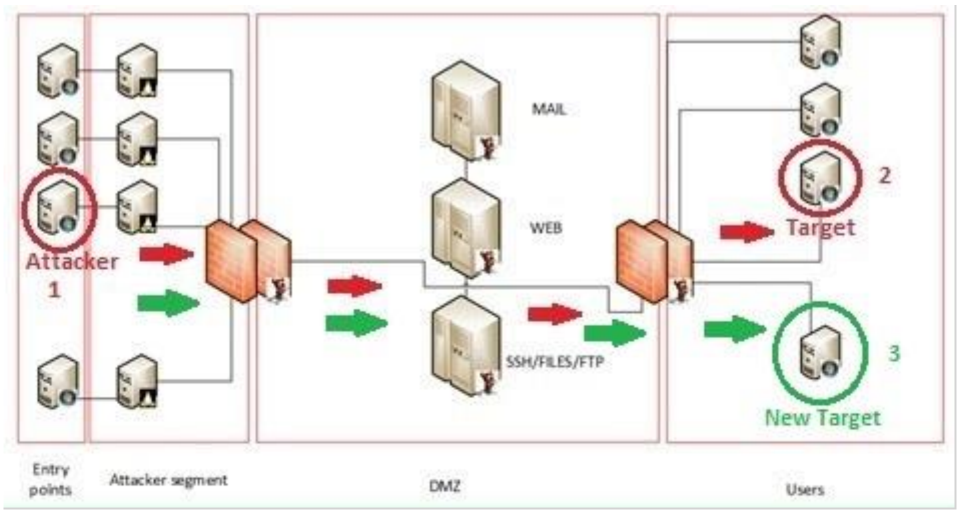


Figure 4.53 The experimental setup

The Attacker computer 1 has an attacker agent installed. It serves as a host for the penetration testing system. The Target host 2 is the computer under attack. The agent uses it to learn the attack strategies. The New Target host 3 is the computer which is attacked as previously unknown target. Both target hosts have the environmental agent installed. It provides the environmental feedback to the penetration testing system. All agents communicate using sockets. The configuration of the targets is similar but not identical. The configuration of the New Target host 3 changes during the experiment. Targets have Windows 7 installed with similar sets of processes. They have the same remote and local vulnerabilities as in previous experiments.

According to the issues highlighted in the Deep Architecture ANN experiment, the current group of experiments is intended to improve the existing generalized case architecture of the penetration testing system. A new approximator is used based on Recurrent Neural Network (RNN). It has additional connections among hidden layer neurons, which feed the previous information to the input again. This architecture provides to the approximator the ability to “memorise” past actions. It should significantly speed up the learning process. More details on RNN can be found in the Elman Recurrent Neural Network section of the Methodology chapter.

Another purpose of these experiments is to demonstrate the adaptivity of the agent. The experiment consists of three stages. During the first phase, the penetration testing agent will attack Target host 2 in order to learn the successful attack strategy. The optimal parameters for Q-learning, which were revealed during the previous experiment, were used. Different acting strategies were used: random, ε-greedy and softmax.

During the second phase, the agent will attack New Target host 3, which it never experienced before. The sequence of actions, leading to successful penetration, will be chosen according to the previous experience. The experience was received during phase 2.

After successful penetration on the New Target host 3, the configuration of the host will be changed. The new configuration will be updated in such a way that the old successful attack strategy (revealed during the second phase) will no longer provide target penetration. As a result the agent should restart its relearning process to find a new successful attacks strategy. Therefore, the system will learn the attack behavior and will not only show successful penetration testing activity without a human interaction, but will automatically adapt to the changing environment.

4.5.3 MDP process

State space definition for the generalized architecture used in the current experiment is identical to the state space in the Deep Architecture ANN experiment. It is represented by the set of features. Those features describe the configuration of attacked hosts from the point of view of residing processes. More details on features description can be found in the Autoencoder section of the Methodology chapter.

The actions of the MDP process are defined in the same way as for the Field experiment (Table 4.7).

Table 4.7 MDP Actions

0.	OS_Win_Check – Checks if OS is Windows 7
1.	Ex_1_remote – Execute remote Exploit 1 (user privileges, Windows 7)
2.	Ex_2_remote – Execute remote Exploit 2 (admin privileges, Windows XP)
3.	Ex_3_remote – Execute remote Exploit 3 (admin privileges, Windows 7)
4.	Ex_4_local – Execute Local exploit 4 (escalate privileges)

The agent does not have a predefined transitions function reward function for every possible state. It is the representation of a free-model approach to the real-world scenario. The model of the environment is not given to the agent. The only knowledge it has is state and action spaces.

4.5.4 Reward

The reward is defined by the numerical value received from the environment as a reaction on an agent's actions. It provides a motive to the agent to learn the sequence of actions leading to the final goal. The target for the current experiment is to find the attack strategy leading to successful penetration to the host under attack.

The reward is defined as follows:

$$R = R + R1 + R2 + R3$$

where the main reward is defined as R. This reward indicates the successful penetration for the agent. In cases where the agent got admin rights on a target host, it receives 100 as a main reward, 0 otherwise:

R1 – Represents exploit effectiveness according to the metasploit rating (great, good and so on). The more effective the exploit, the higher the reward.

R2 – Represents the escalation of privileges bonus. The higher reward is given for remote penetration with admin privileges contrary to the user privileges only.

R3 – Additional reward for testing actions (portscan, OS detection and so on), defined as 20/n (where n is the number of iterations per episode). This reward motivates an agent to explore the environment and reduces the state space before exploitation.

4.5.5 Results assessment

The attacking agent has four exploits in its arsenal: Remote exploit 1 (user privileges, Windows 7), Remote exploit 3 (admin privileges, Windows 7), Remote exploit 2 (admin privileges, Windows XP) and Local exploit 4 for privilege escalation for Windows 7. The first exploit is the remote one, guaranteeing user rights on the attacked machine after its execution. The second exploit provides admin access level to the target remotely. The local exploit escalates the privileges to admin level locally. Windows XP exploit will fail. Exploits have different ratings and different possibilities to harm the target. A remote exploit 3 with admin privileges causes less harm and is more reliable. The agent is able to use probe actions as well. While acting, the agent learns the sequence of actions leading to the successful penetration to the target (attack strategy). There are a number of attack strategies that can be learned by the agent, which are all different according to their optimality.

Possible attack strategies are similar to the deep architecture ANN experiment:

1. Probe action + execute a remote exploit 3 with admin privileges.
2. Execute a remote exploit 3 with admin privileges.
3. Probe action + execute a remote exploit 1 with user privileges + execute local exploit.
4. Execute a remote exploit 1 with user privileges + execute local exploit.
5. The rest of the actions leading to Fail state.

The attack strategies are sorted according to their optimality. The first one is the optimal one. It will guarantee the highest cumulative reward. The performance of the acting policies is compared graphically. The higher the graphic curve rises across the episodes, the better the policy performance. The faster it grows, the faster an attacking agent learns.

4.5.6 Experiment outcome

This subchapter describes the outcome of the experiment defined in the Experiment setup section above. It demonstrates the learning process of the attacking agent based on Q-learning with approximation. The approximator consists of deep machine learning architecture: autoencoder + RNN. The autoencoder is used for feature extraction of the host under attack. The RNN approximates the state space defined by the feature set. Three acting policies were applied to the learning process:

- random – actions are chosen randomly.
- ε-greedy – actions are chosen randomly or according to experience with 1-ε probability.
- softmax – actions are selected randomly but each action has a different probability of being chosen. This probability is calculated according to the experience of the learning agent. The higher the potential reward, the more probable the action will be chosen.

4.5.6.1 Q-learning with random policy

This experiment consists of applying Q-learning based on an RNN approximator with random policy. The results are presented in Figure 4.54.

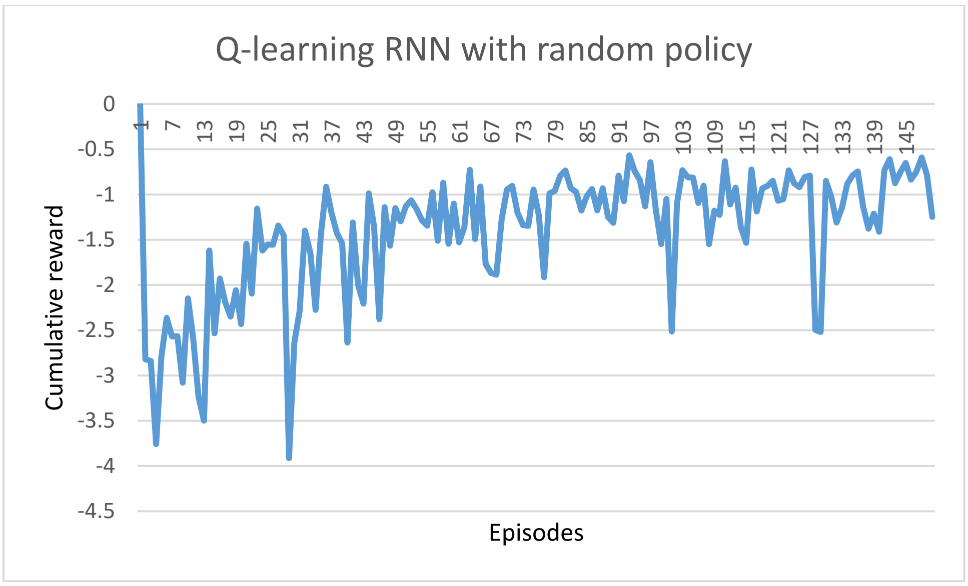


Figure 4.54 Q-learning RNN with random policy

The cumulative reward convergence value is -0.752. The agent learns quite fast. The following solution was found:

- Initialization.
- Execute the remote exploit with user privileges.
- Execute the local exploit.

The next acting policy that was applied was the ε-greedy acting policy.

4.5.6.2 Q-learning with ε-greedy policy

During this experiment, the ε-greedy policy was chosen. Figure 4.55 presents the results (Gamma=0.8, Alpha=0.8, ε=0.15).

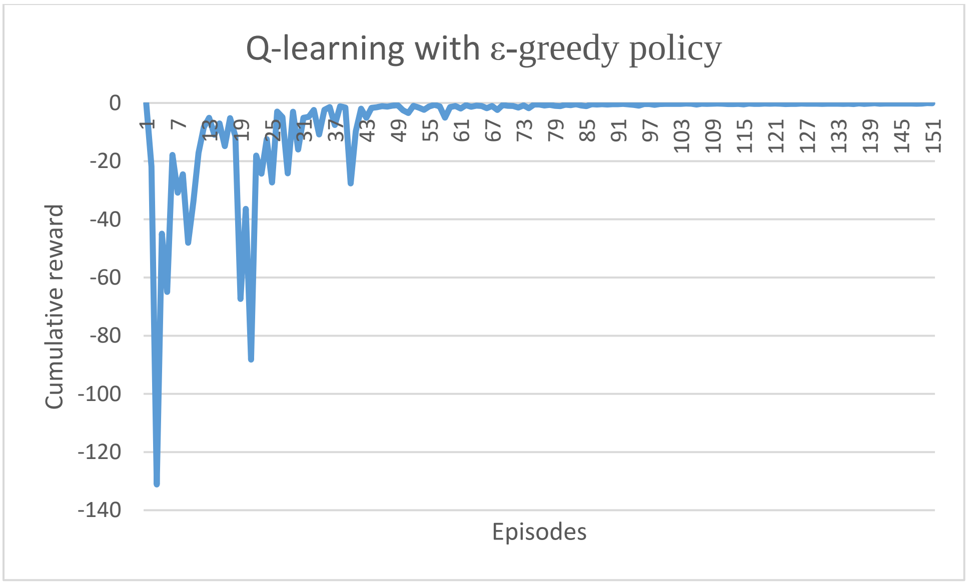


Figure 4.55 Q-learning RNN with ε-greedy policy

The cumulative reward converges to -0.04, which is a higher value than the random policy. The optimal solution was found. The following solution was found:

- Initialization.
- Probe action (check OS).
- Execute the remote exploit with admin privileges.

Furthermore, the softmax acting policy was tested.

4.5.6.3 Q-learning with softmax policy

During this experiment, the softmax policy was used (Figure 4.56).

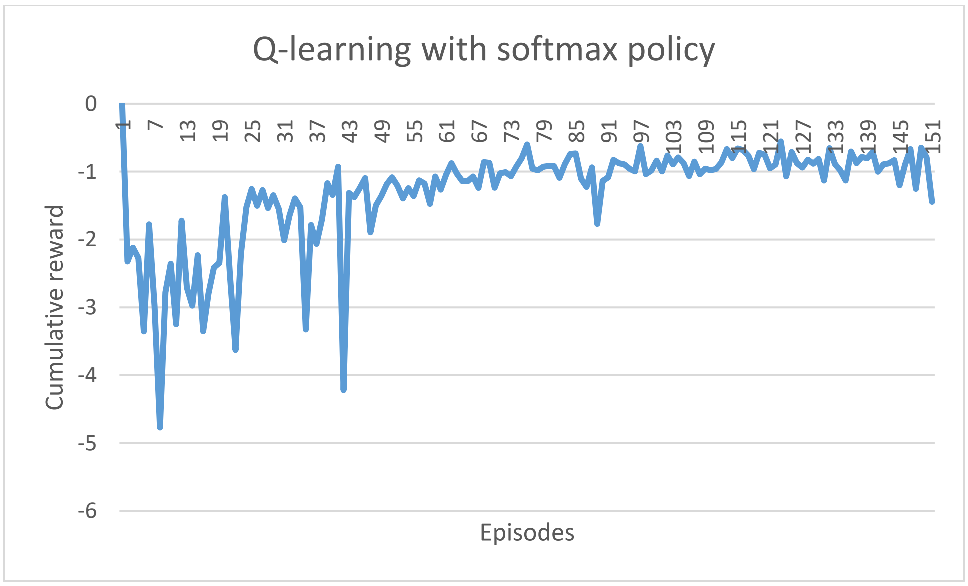


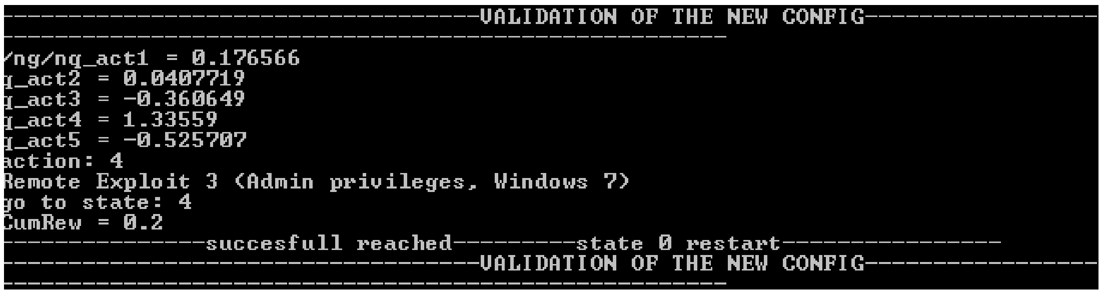
Figure 4.56 Q-learning RNN with softmax policy

The cumulative reward converges to -0.752. The optimal solution was found. However, the agent was able to find the strategy leading to successful penetration:

- Initialization.
- Execute the remote exploit with admin privileges.

4.5.6.4 Adaptivity during attack of a previously unknown target

This experiment represents the attack of previously learned agent against a new host, not experienced before. After the attacking agent has learned the successful attack sequence on Target 2, it attacks New Target 3. It acts according to its own experience gathered during the learning stage using ε-greedy policy. This policy was used as it provides the best performance for the learning algorithm. The agent extracted the feature set representing the configuration of the New target 3. This feature set was used as an input to the approximator in order to get the sequence of actions leading to successful penetration. The solution is presented in Figure *4.57*.



```
----------------------------------VALIDATION OF THE NEW CONFIG-----------------
----------------------------------------------------
/ng/nq_act1 = 0.176566
q_act2 = 0.0407719
q_act3 = -0.360649
q_act4 = 1.33559
q_act5 = -0.525707
action: 4
Remote Exploit 3 (Admin privileges, Windows 7)
go to state: 4
CumRew = 0.2
--------------succesfull reached---------state 0 restart-----------------
----------------------------------VALIDATION OF THE NEW CONFIG-----------------
----------------------------------------------------
```

Figure 4.57: The solution for attack on previously unknown target

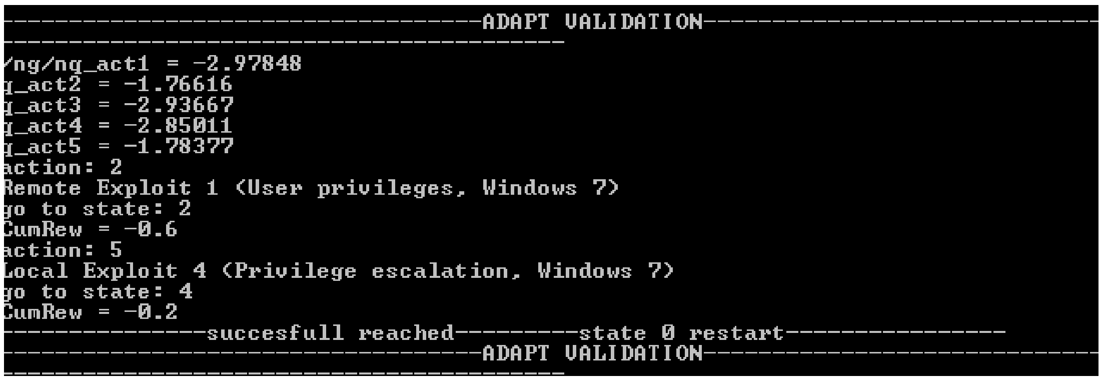
The cumulative reward has the value 0.2. A solution was found:

- Initialization.
- Execute the remote exploit with admin privileges.
- Successful penetration

The sequence of actions leading to successful penetration is not the optimal one. However, it is effective to get admin privileges on the remote host.

The agent was able to penetrate a new host, with which it had never interacted before. It used an experience accumulated during the previous learning iterations while attacking another host.

After successful penetration, the vulnerability exploiting by Remote Exploit 3 was patched. The Remote Exploit 3 was no longer working. However, the agent was able to find an alternative solution (Figure *4.58*).



```
----------------------------------ADAPT VALIDATION------------------------------
----------------------------------------
/ng/nq_act1 = -2.97848
q_act2 = -1.76616
q_act3 = -2.93667
q_act4 = -2.85011
q_act5 = -1.78377
action: 2
Remote Exploit 1 (User privileges, Windows 7)
go to state: 2
CumRew = -0.6
action: 5
Local Exploit 4 (Privilege escalation, Windows 7)
go to state: 4
CumRew = -0.2
--------------succesfull reached---------state 0 restart-----------------
----------------------------------ADAPT VALIDATION------------------------------
----------------------------------------
```

Figure 4.58: Adaptive alternative solution

The cumulative reward has been decreased to -0.2. A solution was found:

- Initialization.
- Execute the remote exploit with user privileges.

- Execute local exploit
- Successful penetration

The sequence of actions will guarantee the successful penetration.

4.5.7 Conclusion

The experiment showed that the penetration testing system was able to learn attack strategies on the go by applying the Q-learning algorithm with an RNN approximator. It is the second case of generalization of the MDP experiment. This experiment demonstrated the adaptivity for the real-life scenario, when the audited host was pathed (fixed) during the penetration testing process. Therefore, the system learned the attack behavior. It demonstrated the successful penetration testing activity without a human interaction. The system automatically adapted to the changing environment as well.

4.5.8 Results conclusion

Table 5.1 describes the performance of the learning algorithm and its parameters across all experiments. Best performance is marked using green color.

Table 5.1 Q-learning performance comparison (over 100 episodes)

Experiment	Learning rate	Discount factor	Policy	1- ε	Higher cumulative reward	Q-learning type
MDP experiment	0.2	0.2	Random	N/A	99	Tabular
MDP experiment	0.5	0.2	Random	N/A	137	Tabular
MDP experiment	0.8	0.2	Random	N/A	151	Tabular
MDP experiment	0.2	0.5	Random	N/A	127	Tabular
MDP experiment	0.5	0.5	Random	N/A	149	Tabular
MDP experiment	0.8	0.5	Random	N/A	150	Tabular
MDP experiment	0.2	0.8	Random	N/A	150	Tabular

MDP experiment	0.5	0.8	Random	N/A	195	Tabular
MDP experiment	0.8	0.8	Random	N/A	250	Tabular
MDP experiment	0.2	0.8	ε-greedy	0.15	149	Tabular
MDP experiment	0.5	0.8	ε-greedy	0.15	176	Tabular
MDP experiment	0.8	0.8	ε-greedy	0.15	263	Tabular
MDP experiment	0.2	0.8	ε-greedy	0.55	100	Tabular
MDP experiment	0.5	0.8	ε-greedy	0.55	175	Tabular
MDP experiment	0.8	0.8	ε-greedy	0.55	210	Tabular
MDP experiment	0.2	0.8	ε-greedy	0.98	30	Tabular
MDP experiment	0.5	0.8	ε-greedy	0.98	100	Tabular
MDP experiment	0.8	0.8	ε-greedy	0.98	140	Tabular
MDP experiment	0.2	0.8	softmax	N/A	115	Tabular
MDP experiment	0.5	0.8	softmax	N/A	175	Tabular
MDP experiment	0.8	0.8	softmax	N/A	205	Tabular
Field experiment	0.2	0.2	Random	N/A	59	Tabular
Field experiment	0.5	0.2	Random	N/A	113	Tabular
Field experiment	0.8	0.2	Random	N/A	123	Tabular
Field experiment	0.2	0.5	Random	N/A	75	Tabular

Field experiment	0.5	0.5	Random	N/A	130	Tabular
MDP experiment	0.8	0.5	Random	N/A	180	Tabular
Field experiment	0.2	0.8	Random	N/A	99	Tabular
Field experiment	0.5	0.8	Random	N/A	178	Tabular
Field experiment	0.8	0.8	Random	N/A	249	Tabular
Field experiment	0.2	0.8	ε-greedy	0.15	98	Tabular
Field experiment	0.5	0.8	ε-greedy	0.15	176	Tabular
Field experiment	0.8	0.8	ε-greedy	0.15	251	Tabular
Field experiment	0.2	0.8	softmax	N/A	53	Tabular
Field experiment	0.5	0.8	softmax	N/A	140	Tabular
Field experiment	0.8	0.8	softmax	N/A	215	Tabular
New vulnerabilities experiment	0.8	0.8	random	N/A	287	Tabular
New vulnerabilities experiment	0.8	0.8	ε-greedy	0.15	290	Tabular
New vulnerabilities experiment	0.8	0.8	ε-greedy	0.55	280	Tabular
New vulnerabilities experiment	0.8	0.8	ε-greedy	0.98	25	Tabular

New vulnerabilities experiment	0.8	0.8	softmax	N/A	275	Tabular
Deep architecture ANN experiment	0.8	0.8	random	N/A	-0.664	Deep architecture (approximation)
Deep architecture ANN experiment	0.8	0.8	ε-greedy	0.15	-0.548	Deep architecture (approximation)
Deep architecture ANN experiment	0.8	0.8	softmax	N/A	-0.784	Deep architecture (approximation)
Deep architecture RNN experiment	0.8	0.8	random	N/A	-0.752	Deep architecture (approximation)
Deep architecture RNN experiment	0.8	0.8	ε-greedy	0.15	-0.04	Deep architecture (approximation)
Deep architecture RNN experiment	0.8	0.8	softmax	N/A	-0.750	Deep architecture (approximation)

The first experiment was related to modelling the typical penetration testing concept. It consisted of a single host attack. The audit process was modelled as an MDP. In order to find the optimal attack sequence, a traditional Q-learning algorithm was applied. The standard modification for tabular cases was used. This experiment approved successful application of a model-free approach to the penetration testing. It showed that the agent can successfully learn the optimal attack strategy without a predefined reward function, described as all possible rewards for all possible states-actions pairs. The reward was derived directly from the environmental feedback. The performance of the learning algorithm demonstrated convergence to optimality after 100 episodes. The best Q-learning hyper-parameters were found: learning rate = 0.8, discount factor = 0.8. During testing, the ε-greedy acting policy (1 - ε = 0.15) was revealed as the best one from the point of view of performance. The MDP experiment was later generalized in Deep learning ANN and Deep learning RNN experiments. In contrast to Reddy & Yalla (2016), where an expert is used to analyse probe data and run mixed test plans manually, the result here demonstrates that the experimental penetration testing system learned the attack strategy automatically, which consisted of probe actions followed by attack actions. In other words, the experiment showed that the audit system was able to demonstrate the typical behavior of the penetration tester. The human factor was successfully excluded and the system learned the attack strategy itself.

The second experiment addressed the exclusion of the transition function from the definition of the environment. The experiment setup was based on the first experiment. However, the attacking agent was updating the reward and transition table for the Q-learning algorithm dynamically according to the environmental feedback. In those conditions, the attacking agent was able to learn the optimal attack strategy and was able to perform a multistep attack as well. The experiment demonstrated the core hypothesis about the possibility of developing automated self-learning penetration testing.

In comparison with Duan et al. (2008), the approach taken here was able to perform successful attacks using exploit combinations. The penetration testing system learned by itself, without any manually created scripts. The human factor was excluded from the decision-making process. In compared paper, scripts should be created manually and updated regularly. The approach described in this thesis is more general. It makes it possible to learn any sequence of attack actions, including scripts construction in perspective. It will be possible if the scripts are divided on atomic actions and the particular rewards are set up. This case was partly approached in the New vulnerabilities experiment, which still needs generalization. This experiment is an important basis for further research on penetration testing, which is able to add new exploits to its database itself or search for zero day vulnerabilities.

The purpose of the third experiment was to test the hypothesis that the penetration testing system is able to learn a new vulnerability exploitation, in particular buffer overflow, which it has never experienced before. The case was demonstrated for the particular MDP. The attacking agent was trying to combine different shellcodes, NOP operations and return addresses in order to learn successful vulnerable sequences. A multistep sequence of actions was found. The algorithm was able to learn the optimal attack strategy without any predefined knowledge about transition and reward function being provided, which describes all transitions and rewards from all states. Such knowledge was learned by the agent while acting. The experiment has not yet been generalized. It is considered to be future work.

The goal of the fourth experiment was to generalize the MDP experiment for huge state spaces. The learning algorithm was modified by Artificial Neural Network approximator implementation instead of Q-table. The modification is able to learn attack strategies against multiple host configurations in a network. It demonstrated the hypothesis for the general case scenario. The experiment showed room for improvement. The efficiency of the approximator can be increased by applying different neural network architectures. The experiment showed the successful penetration to the new, previously unknown host, after learning similar configuration. In comparison to competitors, it makes the approach described here work in new, previously unknown networks. Therefore, the penetration testing system can be relearned in one network and perform the audit in another.

The fifth experiment is a modification of the fourth experiment. The main difference is in using a Recurrent Neural Network approximator instead of an ANN one. It validated the hypothesis as well. Additionally, it proved the adaptivity for the real-life scenario, when the audited host was pathed during the penetration testing process, but the audit system was still able to adapt and perform successful penetration.

For each of the five experiments, additional experiments were accomplished in order to investigate the learning algorithm performance. In every case, the learning algorithm was modified to have a different acting policy. It was changed from random to ε-greedy (0.15), ε-greedy (0.55), ε-greedy (0.98) and softmax for every experiment. The features extraction process was tested as well. The deep architecture hyper-parameters were investigated: iteration quantity, hidden layer neuron quantity, weight decay value and sparsity penalty term value.

5 Conclusion and future work

This thesis describes automating the audit process, penetration testing in particular. Full automation including decision making is somewhat more difficult than automation of separate stages of penetration testing or resolving the planning problem. Modern tools, automating different stages of penetration testing, do not provide the automation of the decision making process. This is due to the huge state space and difficult environmental models. It demands serious resource consumption in order to analyse all state space by model checker/planning algorithm in order to provide a reasonable attack strategy. The computer security expert uses mentioned semi-automatic tools to support manually created attack strategies. Therefore, the problem of human resources load is still unresolved in a modern penetration testing life cycle.

From the other side, the existing planning approach tries to resolve this problem by development of the environmental model for future penetration testing simulations. As a result, the attack strategies can be derived. By attempting to automate the decision making process in such a way, the planning approach generates its own specific problems. The process of building the environmental model has huge time and computational costs. This fact brings additional load related to the audit process and resolves the problem only in the specifically modelled environment. In cases where the model was not developed properly, the attack strategy of the audit agent will have no sense. Leading specialists in the field (Core Impact Lab) approach this problem by doing a huge amount of simulation in their virtual infrastructure to generate the statistics on which the environmental model will be based. Later, the planning algorithm is used to derive the attack strategy. Thus, the approach is not adaptive at all. Every new version of the operating system or new configuration of the host demands new additional simulations in order to rebuild the model of the environment. Additionally, modern penetration testing systems are not able to accumulate experience while acting. As a result, they cannot learn the attack strategies themselves and generate them on the go even after the host configuration has changed.

The involvement of the cyber security expert to the decision making in penetration testing lifecycle brings several drawbacks. The volume of the information in modern corporate networks is enormous. Human expert is not able to analyse full amount of data in the reasonable time. Therefore, he’s forced to skip some facts to be able to operate further. Such information might be crucial for decision making process and, as a result critical attacks can be missed. Another serious drawback is the expert time cost. The more skilled cyber security professional is, the more expensive penetration testing process is. The automation of the penetration testing process will let human expert to control more penetration testing processes simultaneously. It will move him to the higher control role, which can be filled my less skilled specialist, decreasing the cost of penetration testing. In the same time an advanced security expert can be moved to the machine learning related tasks for tuning up the penetration testing automation algorithms. Additionally, those algorithms can work out of hours nonstop. Reduction of the penetration testing cost leads to the availability of such service for the smaller companies. They will be able to have the regular information security checks without attraction of the highly trained professionals. It will make all industry much resistant to cyber security attacks in general.

This thesis has demonstrated the automation of the penetration testing process by applying a model free machine learning approach. Therefore, the human factor has been fully excluded from the decision making process. It made the full automation of the penetration testing process possible contrary to the automation of separate stages. There is a variety of advantages related to this approach. Firstly, the audit system is able to derive the attack strategy itself without cyber security officer involvement. Secondly, the approach lets the penetration testing system learn the attack strategy on the go by accumulating the experience, received during the audit process. Thirdly, in order to build an attack strategy, the system does not need the environmental model. Consequently, the penetration testing system does not depend on the configuration of the attacking system. It will be able to find the attack strategy in any case. Therefore, the system is much more adaptive to the configuration changes. The approach mitigates the problem of huge state space by using the approximators to define all possible host configurations. This solution allows derivation of successful attacks strategies even for hosts that have not been experienced before. Another advantage of our approach is independence of the features describing target hosts. The developed deep learning architecture works universally for any quantity and quality of the chosen feature sets. Therefore, the penetration testing system becomes significantly scalable and adjustable.

Lastly, realizing the significance of the automation of the new vulnerabilities discovery process, the approach was applied to the buffer overflow vulnerability discovery as well. The results of the research showed that the penetration testing system was able to learn and identify the sequence of actions triggering the buffer overflow vulnerability.

5.1 Automation of penetration testing using a model free machine learning approach

The MDP experiments and Field experiments in the Results Chapter verified the model free approach to the automation of penetration testing. The particular example of penetration testing process as an MDP was modeled and the state space and action space defined. Contrary to the planning approach, the reward and transition spaces were not defined. Instead of definition of full environmental model and later penetration testing process simulation, the approach let the audit agent act straight away to accumulate experience and learn the attacker behavior autonomously. In other words, the rewards and transition function are learned on the go. As opposed to the model checking or planning approaches, this approach does not collect any state/action transition statistics via model simulations or using model checker. The advantage of the approach is in its ability to generate attack strategies without going through a huge amount of simulations for every possible host configuration such as: OS version, open ports, programs installed etc. The agent is able to derive the attack strategies based on minimal experience, which is accumulated with every penetration testing action, making the attack strategy more and more optimal. In order to develop the self-learning penetration testing system, a model free Q-learning algorithm was applied. It guaranteed the convergence to the most optimal attack strategy for tabular cases. Using this algorithm, the audit agent is able to predict the optimality of actions in particular states leading to the penetration testing goal. Based on that fact, the agent can derive the optimal attack strategy. Additionally, in the Deep Architecture ANN experiments/Deep architecture RNN experiments in the Results chapter, the approach was generalized by application of an approximator based on custom deep learning architecture consisting of an autoencoder and ANN/RNN neural networks. Approximator incorporation showed the effectiveness of the approach in general case scenarios. It mitigated the huge penetration testing state space problem. The audit agent was able to learn faster and accumulate more generalized information about hosts. Another advantage of the deep learning architecture itself is its universalism. The usage of an autoencoder made the learning process independent of quantity or quality of features. In other words, no matter which features have been chosen to represent hosts, the audit agent will be able to accumulate the knowledge and generate attack strategies.

The developed approach can be easily deployed in real-life networks. Attack agent needs to be installed at the host, which will be the starting point of the penetration test. It will be performing actual attacks. Additionally, feedback agents should be installed on all target hosts of the network. It can be done simultaneously by network administrator using remote tools. After agents are deployed, the system checks the connection between components and ready to start autonomous penetration testing.

The results show valuable insight of applying the model free machine learning approach to penetration testing automation. The approach makes the automation of full penetration testing cycle possible, including the decision making process. Because of the deep architecture approximator, the solution is more adaptive compared to the planning or model checking approaches. The results showed that the audit agent was able to derive successful attack strategies for hosts it had never experienced before. Such a scenario will not work in named alternative solutions. In the case of experiencing new host configurations, the model dependent approaches will not be able to proceed until the model is rebuild. In contrast, this agent was not only able to deal with never experienced host configurations, but to adapt itself when the configuration of the target host changed in the middle of the audit process. The approach taken here made it possible for the audit agent to be relearned and to generate a new successful attack strategy. Therefore, the approach demonstrated the possibility to create a smart penetration testing tool which learns the behavior of penetration tester and apply it for autonomous audit without decision making made by human.

5.2 Conclusion

The computer security audit process has been successfully automated, penetration testing in particular. A model free machine learning approach was applied in order to automate the decision making. The objectives of the project were accomplished successfully. Modern approaches to penetration testing and machine learning application were investigated and are available in the literature review. Using the theoretical ground of this research, the automated self-learning approach to penetration testing was developed based on Q-learning application. It was generalized by using approximators. The results show that the methodology allows us to not model the target system for further validation, but to use a model-free learning algorithm that learns attack strategies on the go. The results are very encouraging and are better than previous planning and model-checking approaches in terms of adaptivity and scalability.

The approach outlined in this research does not depend on the modelling stage, which makes it possible to exclude the human factor, which was demonstrated in Results section. The penetration testing system uses the general feature set to represent the configuration of the hosts. They are represented as a set of processes. Therefore, such features minimize platform dependency. In other words, even if a new version of the OS does appear, the approach would still be relevant. It should also be mentioned that the deep learning architecture, supporting the learning process, is not dependent on particular features. It was developed in such a way that the feature set can be changed without any harm to the knowledge accumulation process. If the features are changed from processes to open/closed ports or any other possible features, the approach will not be affected and will still work. This is a significant and novel step for cyber security audit automation.

5.3 Future work

There are several research directions from the work presented in this thesis. Deep Architecture ANN experiment/Deep architecture RNN experiment in the Results chapter presented the custom build machine learning architecture. Optimal learning algorithm parameters were found and a custom feature set developed. Different ways of feature representation could be developed in the future in order to optimize the proposed approach. Additional information could be added to the feature set, describing a host's network position related to the attack path. It will make the agent learn attack strategies based not only on the information related to the target host, but to its network environment as well. Current deep learning architecture leaves the freedom to experiment with different feature extraction mechanisms. For example, instead of RNN/ANN, different types of neural networks can be used in the future. The back-propagation algorithm can be substituted, as well as activation functions to improve learning performance.

The New vulnerabilities experiment applied the model free machine learning approach to the buffer overflow vulnerability discovery. The agent was able to learn the sequence of actions leading to the successful exploitation of the vulnerability. The effectiveness of the approach was shown, however it was based on a particular example. Future work could generalize this experiment by developing new appropriate features sets and deep learning architecture. As a next stage, the deep architecture can be extended on the additional types of vulnerabilities such as SQL attacks.

Several future extensions to this research on adaptive penetration testing are suggested. At the moment, the exploit is a minimal action unit in developed machine learning architecture. Breaking exploits into smaller parts of code and making them inputs for the learning system could be pursued. It would give the possibility for the audit agent not only to learn the sequence of exploits leading to successful penetration, but to compile them from the code parts automatically depending of the target. This might give the opportunity to the agent to create completely new exploits never existing before based on its experience. Future work on generalization of the New vulnerabilities experiment could also be incorporated in that concept.

In the MDP experiments in the Results chapter, the audit system is modeled using MDP. However, the model could be improved by remodeling it using POMDP, which can represent hidden state space. Such remodeling will not affect the learning process, but will make it possible to represent actions consisting of another action. This will extend the capabilities of current penetration testing systems. It will make it possible for the audit agent to learn complicated multi step attacks consisting of another multistep attack. Such an approach will also give the opportunity to the agent to break the goals into sub goals and process them in the right order.

6 References

Abu-Nimeh, S. *et al.* (2007) 'A comparison of machine learning techniques for phishing detection', *Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit on - eCrime '07*, pp. 60–69. doi: 10.1145/1299015.1299021.

Aguilar, S. and Riquelme, C. (2007) 'Supervised Learning', 11(4), pp. 466–479.

Alhazmi, O. H. and Malaiya, Y. K. (2005) 'Modeling the vulnerability discovery process', *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 2005, pp. 129–138. doi: 10.1109/ISSRE.2005.30.

Anfilofiev, A. E., Hodashinsky, I. A. and Evsutin, O. O. (2015) 'Algorithm for tuning fuzzy network attack classifiers based on invasive weed optimization', *2014 Dynamics of Systems, Mechanisms and Machines, Dynamics 2014 - Proceedings*, pp. 1–4. doi: 10.1109/Dynamics.2014.7005632.

Arkin, B., Stender, S. and McGraw, G. (2005) 'Software penetration testing', *IEEE Security and Privacy*, 3(1), pp. 84–87. doi: 10.1109/MSP.2005.23.

Aslam, J., Bratus, S. and Pavlu, V. (2006) 'Semi-supervised data organization for interactive anomaly analysis', *Proceedings - 5th International Conference on Machine Learning and Applications, ICMLA 2006*, pp. 55–62. doi: 10.1109/ICMLA.2006.47.

Atkins, B. and Huang, W. (2013) 'A Study of Social Engineering in Online Frauds', *Open Journal of Social Sciences*, 1(3), pp. 23–32. doi: 10.4236/jss.2013.13004.

B. Schneier (1999) 'Academic: Attack Trees - Schneier on Security'. Available at: https://www.schneier.com/academic/archives/1999/12/attack_trees.html.
Baldi, P. (2012) 'Autoencoders, Unsupervised Learning, and Deep Architectures', *ICML Unsupervised and Transfer Learning*, pp. 37–50.

Barik, M. S., Sengupta, A. and Mazumdar, C. (2016) 'Attack graph generation and analysis techniques', *Defence Science Journal*, 66(6), pp. 559–567. doi: 10.14429/dsj.66.10795.

Bechtsoudis, A. and Sklavos, N. (2012) 'Aiming at higher network security through extensive penetration tests', *IEEE Latin America Transactions*, 10(3), pp. 1752–1756. doi: 10.1109/TLA.2012.6222581.

Bengio, Y. (1994) 'Learning long-term dependencies with gradient descent is difficult'. doi: 10.1109/72.279181.

Caglar Gulcehre, Marcin Moczulski, Misha Denil, Y. B. (2016) 'Noisy Activation Functions', *Arxiv*. Available at: http://arxiv.org/abs/1603.00391.

Chellapilla, K. and Simard, P. (2004) 'Using Machine Learning to Break Visual Human Interaction Proofs (HIPs)', *Advances in Neural Information Processing Systems 17*, (February), pp. 265–272.

Dasgupta, D. and Brian, H. (2001) 'Mobile security agents for network traffic analysis', *… &amp; Exposition II, 2001. DISCEX'01. …*, (June). Available at:

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=932184.

Denis, M., Zena, C. and Hayajneh, T. (2016) 'Penetration testing: Concepts, attack methods, and defense strategies', *2016 IEEE Long Island Systems, Applications and Technology Conference, LISAT 2016*. doi: 10.1109/LISAT.2016.7494156.

Duan, B., Zhang, Y. and Gu, D. (2008) 'An easy-to-deploy penetration testing platform', *Proceedings of the 9th International Conference for Young Computer Scientists, ICYCS 2008*, pp. 2314–2318. doi: 10.1109/ICYCS.2008.335.

Durkota, K. and Lisy, V. (2014) 'Computing Optimal Policies for Attack Graphs with Action Failures and Costs', *Frontiers in Artificial Intelligence and Applications*, 264, pp. 101–110. doi: 10.3233/978-1-61499-421-3-101.

Erik G. (2014) 'Introduction to Supervised Learning', pp. 1–5. Available at: http://people.cs.umass.edu/~elm/Teaching/Docs/supervised2014a.pdf.

Ford, V. and Siraj, A. (2014) 'Applications of Machine Learning in Cyber Security', *CAINE - International Conference on Computer Applications in Industry and Engineering*, (December). Available at: https://www.researchgate.net/publication/283083699_Applications_of_Machine_Learning_in_Cyber_Security.

Futoransky, A. *et al.* (2003) 'Building Computer Network Attacks', *Security*, (June 2010), pp. 1–18. Available at: http://arxiv.org/abs/1006.1916.

Geist, M. and Scherrer, B. (2014) 'Off-policy Learning With Eligibility Traces: A Survey', *Journal of Machine Learning Research*, 15(Jan), p. 289−333. Available at: http://jmlr.org/papers/v15/geist14a.html.

Gers, F. A., Schmidhuber, J. and Cummins, F. (2000) 'Learning to forget: continual prediction with LSTM.', *Neural computation*, 12(10), pp. 2451–2471. doi: 10.1162/089976600300015015.

Gers, F. a, Schraudolph, N. N. and Schmidhuber, J. (2002) 'Learning Precise Timing with LSTM Recurrent Networks', *Journal of Machine Learning Research*, 3(1), pp. 115–143. doi: 10.1162/153244303768966139.

Gou, S. *et al.* (2009) 'Distributed Transfer Network Learning Based Intrusion Detection', *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 511–515. doi: 10.1109/ISPA.2009.92.

Gupta, N. and Mathur, A. (1998) 'Automated test data generation using an iterative relaxation method', *ACM SIGSOFT Software Engineering*, pp. 231–244. doi: 10.1145/291252.288321.

Haddadi, F. *et al.* (2014) 'Botnet behaviour analysis using IP flows: With http filters using classifiers', *Proceedings - 2014 IEEE 28th International Conference on Advanced Information Networking and Applications Workshops, IEEE WAINA 2014*, pp. 7–12. doi: 10.1109/WAINA.2014.19.

Haldar, V., Chandra, D. and Franz, M. (2005) 'Dynamic taint propagation for Java', *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 2005(Acsac), pp. 303–311. doi: 10.1109/CSAC.2005.21.

Harm van, S. *et al.* (2009) 'A theoretical and empirical analysis of expected sarsa', *2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, ADPRL 2009 - Proceedings*, pp. 177–184. doi: 10.1109/ADPRL.2009.4927542.

Hinton, G. E., Krizhevsky, A. and Wang, S. D. (2011) 'Transforming auto-encoders', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6791 LNCS(PART 1), pp. 44–51. doi: 10.1007/978-3-642-21735-7_6.

Hoffmann, J. (2015) 'Simulated Penetration Testing: From" Dijkstra" to" Turing Test++".', *Icaps*, pp. 364–372.

Holik, F. *et al.* (2014) 'Effective penetration testing with Metasploit framework and methodologies', *CINTI 2014 - 15th IEEE International Symposium on Computational Intelligence and Informatics, Proceedings*, pp. 237–242. doi: 10.1109/CINTI.2014.7028682.
Impact, C. (2009) 'Core impact'.

Hssina, B., et al. (2014). 'A comparative study of decision tree ID3 and C4.5. ' (IJACSA) International Journal of Advanced Computer Science and Applications. Special Issue on Advances in Vehicular Ad Hoc Networking and Applications. 10.14569/SpecialIssue.2014.040203.

Irodova, M. and Sloan, R. (2005) 'Reinforcement Learning and Function Approximation', *FLAIRS Conference*. Available at: http://www.aaai.org/Papers/FLAIRS/2005/Flairs05-075.pdf.

Jajodia, S. and Noel, S. (2010) 'Topological vulnerability analysis', *Advances in Information Security*, 46, pp. 139–154. doi: 10.1007/978-1-4419-0140-8_7.

Jajodia, S., Noel, S. and O'Berry, B. (2005) 'Topological analysis of network attack vulnerability', *Managing Cyber Threats*, pp. 247–266. doi: 10.1145/1229285.1229288.

Jha, S., Wing, J. and Sheyner, O. (2002) 'Minimization and reliability analyses of attack graphs', *Citeseer*. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.1788&rep=rep1&type=pdf%5Cnpapers2://publication/uuid/BFFE4D66-6A9E-4144-B0E8-26AE7B3C0D81.

Khan, M. S., Ferens, K. and Kinsner, W. (2014) 'A chaotic measure for cognitive machine classification of distributed denial of service attacks', *Proceedings of 2014 IEEE 13th International Conference on Cognitive Informatics and Cognitive Computing, ICCI*CC 2014*, pp. 100–108. doi: 10.1109/ICCI-CC.2014.6921448.

Kordy, B. *et al.* (2013) 'ADTool: Security Analysis with Attack- Defense Trees', *Lecture Notes in Computer Science*, 8054(318003), pp. 173–176. Available at: http://arxiv.org/abs/1305.6829.

Kotsiantis, S. B., Zaharakis, I. D. and Pintelas, P. E. (2006) 'Machine learning: A review of classification and combining techniques', *Artificial Intelligence Review*, 26(3), pp. 159–190. doi: 10.1007/s10462-007-9052-3.

L. Swiler, C. Phillips, T. G. (1998) 'A Graph-Based Network-Vulnerability Analysis System', *Sandia report*.
Lane, T. (2006) 'A Decision-Theoretic, Semi-Supervised Model for Intrusion Detection', *Machine learning and data mining for computer security: Methods and applications*.

Li, L., Deng, Z. and Zhang, B. (1997) 'A Fuzzy Elman Neural Network', *CiteSeer-Scientific Literature Digital Library and Search Engine 1997*.

Lin, K. P. and Chen, M. S. (2008) 'Releasing the SVM classifier with privacy-preservation', *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 899–904. doi: 10.1109/ICDM.2008.19.

Lippmann, R. *et al.* (2000) 'The 1999 DARPA o -line intrusion detection evaluation', *Computer Networks*, 34(4), pp. 579–595. doi: 10.1016/S1389-1286(00)00139-0.

Lipton, Z. C. *et al.* (2015) 'Learning to Diagnose with LSTM Recurrent Neural Networks', *Iclr*, pp. 1–18. Available at: http://arxiv.org/abs/1511.03677.

Liu, S. (no date) 'Conquering vanishing gradient : Tensor Tree LSTM on aspect-sentiment classification', pp. 1–7.

Lore, K. G., Akintayo, A. and Sarkar, S. (2015) 'LLNet: A Deep Autoencoder Approach to Natural Low-light Image Enhancement'. Available at: http://arxiv.org/abs/1511.03995.

Manadhata, P. K. and Wing, J. M. (2011) 'An attack surface metric', *IEEE Transactions on Software Engineering*, 37(3), pp. 371–386. doi: 10.1109/TSE.2010.60.

Masri, A. El *et al.* (2014) 'Identifying Users with Application-Specific Command Streams', pp. 232–238.

Di Mauro, M. and Longo, M. (2014) 'Skype traffic detection: A decision theory based tool', *Proceedings - International Carnahan Conference on Security Technology*, 2014–Octob(October). doi: 10.1109/CCST.2014.6986975.

Mauw, S. and Oostdijk, M. (2006) 'Foundations of attack trees', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3935 LNCS(im), pp. 186–198. doi: 10.1007/11734727_17.

Melo, F. S., Meyn, S. P. and Ribeiro, M. I. (2008) 'An analysis of reinforcement learning with function approximation', *Proceedings of the 25th international conference on Machine learning*, pp. 664–671. doi: 10.1145/1390156.1390240.

Mhaskar, H. N. and Hahm, N. (1997) 'Neural networks for functional approximation and system identification.', *Neural computation*, 9(1), pp. 143–159. doi: 10.1162/neco.1997.9.1.143.

Moore, A. P., Ellison, R. J. and Linger, R. C. (2001) 'Attack modeling for information security and survivability', *Technical Note CMUSEI2001TN001*, 17(March), pp. 15–33. Available at:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.9283&amp;rep=rep1&amp;type=pdf.

Moore, A. W. *et al.* (2005) 'Internet traffic classification using bayesian analysis techniques', *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS '05*, 33(1), p. 50. doi: 10.1145/1064212.1064220.

Most, T. (2005) 'Approximation of complex nonlinear functions by means of neural networks', *Proc. Weimarer Optimierungs-und Stochastiktage*, 2, pp. 1–2.

Mukhopadhyay, I. (2014) 'Web Penetration Testing using Nessus and Metasploit Tool', *IOSR Journal of Computer Engineering (IOSR-JCE)*, 16(3), pp. 126–129. doi: 10.9790/0661-1634126129.

Nexpose (no date) 'Nexpose'.

Nissen, S. (2007) 'Large Scale Reinforcement Learning using Q -SARSA(λ) and Cascading

Neural Networks', *Department of Computer Science University of Copenhagen*, pp. 43–77.

Obes, J. L., Sarraute, C. and Richarte, G. (2003) 'Attack Planning in the Real World', *Security*.

Ou, X., Boyer, W. F. and McQueen, M. a. (2006) 'A scalable approach to attack graph generation', *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*, (March 2016), p. 336. doi: 10.1145/1180405.1180446.

Pacanu, R., Mikolov, T. and Bengio, Y. (2013) 'On the Difficulties of Training Recurrent Neural Networks', *Icml*, (2). doi: 10.1109/72.279181.

Pan, W. and Li, W. (2009) 'A penetration testing method for e-Commerce authentication system security', *2009 International Conference on Management of e-Commerce and e-Government, ICMeCG 2009*, pp. 449–453. doi: 10.1109/ICMeCG.2009.111.

Pascal, V. and Hugo, L. (2010) 'Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion Pierre-Antoine Manzagol', *Journal of Machine Learning Research*, 11, pp. 3371–3408. doi: 10.1111/1467-8535.00290.

Qiu, X. *et al.* (2014) 'Automatic generation algorithm of penetration graph in penetration testing', *Proceedings - 2014 9th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2014*, pp. 531–537. doi: 10.1109/3PGCIC.2014.104.

Reddy, M. R. and Yalla, P. (2016) 'Mathematical analysis of penetration testing and vulnerability countermeasures', *Proceedings of 2nd IEEE International Conference on Engineering and Technology, ICETECH 2016*, (March), pp. 26–30. doi: 10.1109/ICETECH.2016.7569185.

Report, F. T. (2010) 'Advanced Cyber Attack Modeling , Analysis , and Visualization', *Advance Cyber Attack Modelling, Analysis, and visualization*, (March).

Revett, K. *et al.* (2007) ‘A machine learning approach to keystroke dynamics based user authentication’, *International Journal of Electronic Security and Digital Forensics*, 1(1), p. 55. doi: 10.1504/IJESDF.2007.013592.

Ritchey, R. W. *et al.* (2000) ‘Using Model Checking to Analyze Network Vulnerabilities’.
Rushing, D., Guidry, J. and Alkadi, I. (2015) ‘Collaborative penetration-testing and analysis toolkit (CPAT)’, *IEEE Aerospace Conference Proceedings*, 2015–June. doi: 10.1109/AERO.2015.7119262.

Sahu, S. K., Sarangi, S. and Jena, S. K. (2014) ‘A detail analysis on intrusion detection datasets’, *Souvenir of the 2014 IEEE International Advance Computing Conference, IACC 2014*, pp. 1348–1353. doi: 10.1109/IAdCC.2014.6779523.

Samek, D. (1999) ‘Elman neural networks in model predictive control’, 6(Cd).
Sarraute, C., Buffet, O. and Hoffmann, J. (2013) ‘Penetration Testing == POMDP Solving?’, pp. 66–73. Available at: http://arxiv.org/abs/1306.4714.

Sarraute, C., Richarte, G. and Obes, J. L. (2013) ‘An Algorithm to Find Optimal Attack Paths in Nondeterministic Scenarios’. Available at: http://arxiv.org/abs/1306.4040.

Schneier, B. (2000) ‘Attack Trees’, *Dr. Dobb’s Journal of Spftware Tools*, 24(12), p. 60. Available at: http://www.schneier.com/paper-attacktrees-ddj-ft.html.

Secunia (2015) ‘Secunia Vulnerability Review 2015’, p. 24.

Shabtai, A., Fledel, Y. and Elovici, Y. (2010) ‘Automated static code analysis for classifying android applications using machine learning’, *Proceedings - 2010 International Conference on Computational Intelligence and Security, CIS 2010*, pp. 329–333. doi: 10.1109/CIS.2010.77.

Shah, S. and Mehtre, B. M. (2015a) ‘An automated approach to vulnerability assessment and penetration testing using net-nirikshak 1.0’, *Proceedings of 2014 IEEE International Conference on Advanced Communication, Control and Computing Technologies, ICACCCT 2014*, (978), pp. 707–712. doi: 10.1109/ICACCCT.2014.7019182.

Shah, S. and Mehtre, B. M. (2015b) ‘An overview of vulnerability assessment and penetration testing techniques’, *Journal of Computer Virology and Hacking Techniques*, 11(1), pp. 27–49. doi: 10.1007/s11416-014-0231-x.

Sheyner, O. *et al.* (2002) ‘Automated generation and analysis of attack graphs’, *Proceedings - IEEE Symposium on Security and Privacy*, 2002–Janua, pp. 273–284. doi: 10.1109/SECPRI.2002.1004377.

Sheyner, O. and Wing, J. (2004) ‘Tools for Generating and Analyzing Attack Graphs’, pp. 344–371.

Shinde, P. and Parvat, T. (2014) ‘Analysis on : Intrusions Detection Based On Support Vector Machine Optimized with Swarm Intelligence’, 3(12), pp. 559–566.

Stefinko, Y., Piskozub, A. and Banakh, R. (2016) ‘Manual and automated penetration testing. Benefits and drawbacks. Modern tendency’, *Modern Problems of Radio*

Engineering, Telecommunications and Computer Science, Proceedings of the 13th International Conference on TCSET 2016, 1, pp. 488–491. doi: 10.1109/TCSET.2016.7452095.

Sutton, R. S. and Barto, A. G. (2012) 'Reinforcement learning', *Learning*, 3(9), p. 322. doi: 10.1109/MED.2013.6608833.

Templeton, S. J. and Levitt, K. (2000) 'A requires/provides model for computer attacks', *Proceedings of the 2000 workshop on New security paradigms - NSPW '00*, pp. 31–38. doi: 10.1145/366173.366187.

Tenable Network Security Inc. (2015) 'Nessus Professional Vulnerability Scanner', pp. 1–2.
Tidwell, T. *et al.* (2001) 'Modeling Internet Attacks', *Network*, 1(January 2001), pp. 5–6. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.9040&amp;rep=rep1&amp;type=pdf.

Tokarchuk, L., Bigham, J. and Cuthbert, L. (no date) 'Fuzzy Sarsa : An approach to fuzzifying Sarsa Learning'.

Tomandl, A., Fuchs, K. P. and Federrath, H. (2014) 'REST-Net: A dynamic rule-based IDS for VANETs', *2014 7th IFIP Wireless and Mobile Networking Conference, WMNC 2014*. doi: 10.1109/WMNC.2014.6878854.

Tsang, E. C. C. (no date) 'A Feature Space Analysis for Anomaly Detection', *2005 IEEE International Conference on Systems, Man and Cybernetics*, pp. 3599–3603. doi: 10.1109/ICSMC.2005.1571706.

W. t. Wong and C. y. Lai (2006) 'Identifying Important Features for Intrusion Detection using Discriminant Analysis and Support Vector Machine', (707), pp. 3563–3567. doi: 10.1109/ICMLC.2006.258571.

Wack, J., Tracy, M. and Souppaya, M. (2003) 'C o m p u t e r s e c u r i t y', *Nist Special Publication*, (October).

Wagh, S. K. (2014) 'Effective Intrusion Detection System Using Semi- Supervised Learning', *2014 International Conference on Data Mining and Intelligent Computing*, pp. 1–5.

Wang, G. (2008) 'A survey on training algorithms for support vector machine classifiers', *Proceedings - 4th International Conference on Networked Computing and Advanced Information Management, NCM 2008*, 1, pp. 123–128. doi: 10.1109/NCM.2008.103.

Watkins, C., Dayan, P. (1992) 'Q-learning', Machine learning 8, pp. 279-292

Wing, J. M. (2008) 'Scenario Graphs Applied to Network Security', *Information Assurance*, pp. 247–277. doi: 10.1016/B978-012373566-9.50011-2.

Xu, B., Huang, R. and Li, M. (2016) 'REVISE SATURATED ACTIVATION FUNCTIONS', pp. 1–4.

Xu, X., Zuo, L. and Huang, Z. (2014) 'Reinforcement learning algorithms with function approximation: Recent advances and applications', *Information Sciences*. Elsevier Inc., 261, pp. 1–31. doi: 10.1016/j.ins.2013.08.037.

Xu, Z., Sun, J. and Xiaojun, U. (2003) ‘An immune genetic model in rule-based state action ids’, (November), pp. 2–5.

Xue Qiu *et al.* (2014) ‘An automated method of penetration testing’, *2014 IEEE Computers, Communications and IT Applications Conference*, pp. 211–216. doi: 10.1109/ComComAp.2014.7017198.

Yanchao, Z. *et al.* (2001) ‘An immunity-based model for network intrusion detection’, *Info-tech and Info-net, 2001. Proceedings. ICII 2001 - Beijing. 2001 International Conferences on*, 5, pp. 24–29 vol.5. doi: 10.1109/ICII.2001.983489.

Yang, Y. *et al.* (2013) ‘Rule-based intrusion detection system for SCADA networks’, *Renewable Power Generation Conference (RPG 2013), 2nd IET*, pp. 1–4. doi: 10.1049/cp.2013.1729.

Younis, A. A. and Malaiya, Y. K. (2014) ‘Using software structure to predict vulnerability exploitation potential’, *Proceedings - 8th International Conference on Software Security and Reliability - Companion, SERE-C 2014*, pp. 13–18. doi: 10.1109/SERE-C.2014.17.

Yu, W. and Cao, J. (2006) ‘Cryptography based on delayed chaotic neural networks’, *Physics Letters, Section A: General, Atomic and Solid State Physics*, 356(4–5), pp. 333–338. doi: 10.1016/j.physleta.2006.03.069.

Zhang, J. *et al.* (2015) ‘Robust Network Traffic Classification’, *IEEE/ACM Transactions on Networking*, 23(4), pp. 1257–1270. doi: 10.1109/TNET.2014.2320577.

Zhang, Z., Tang, Z. and Vairappan, C. (2007) ‘LETTER A Novel Learning Method for Elman Neural Network Using Local Search’, *A Nover learning method for elman neural network*, 11(8), pp. 181–188.

Zhao, J. *et al.* (2015) ‘Penetration testing automation assessment method based on rule tree’, *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, pp. 1829–1833. doi: 10.1109/CYBER.2015.7288225.

Zhu, N. *et al.* (2008) ‘Design and application of penetration attack tree model oriented to attack resistance test’, *Proceedings - International Conference on Computer Science and Software Engineering, CSSE 2008*, 3, pp. 622–626. doi: 10.1109/CSSE.2008.1137.

Zhuang, W. *et al.* (2012) ‘Ensemble clustering for internet security applications’, *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, 42(6), pp. 1784–1796. doi: 10.1109/TSMCC.2012.2222025.