Exploring a Test-Driven Approach to Network Security Policy

R. David Sigsby

Davenport University

Fall 2017

In partial fulfillment of the requirements for the

Degree of Master of Science in Information Assurance
Abstract

Agile software development has evolved over the years to keep pace with the growing demands of digital business. For the same reason, the growing DevOps movement has introduced agile principles into IT infrastructure and operations. This paper asserts that agile techniques can be applied to a range of information security domains in order to effectively achieve similar benefits and to better integrate security with software development and DevOps practices. This paper explores how the agile practice of Test-Driven Development (TDD) can be applied to the development of network security policy. First, existing research on the benefits of TDD as it applies to software development is reviewed along with a brief discussion on DevOps as it relates to infrastructure-as-code, software-defined infrastructure, and the emerging practice of Test-Driven Infrastructure (TDI). Next, a number of choices in platforms, tools, and strategies is considered for the development of a proof-of-concept implementation to show how TDI can be leveraged to apply a test-driven approach to developing network security policy. In conclusion, the obstacles and limitations encountered during this work are discussed and future research is proposed.
Contents

Abstract .......................... 2

Introduction ....................... 5

Literature Review .................. 7

Benefits of Test-Driven Development .................................................. 7
Principles of Test-Driven Development ................................................. 10
Behavior-Driven Development ............................................................. 11
DevOps and Test-Driven Infrastructure ............................................... 12

Methodology ......................... 14

Testing Platform .................... 14
  Workstation Operating System ......................................................... 14
  Virtualization Tools ................................................................. 15
  Platform Summary and Other Tools ............................................... 16
  Simulated Network Architecture .................................................. 17

Findings ......................... 18

Conclusions ......................... 22

References ......................... 24

Appendix A: Firewall Basebox Makefile ........................................... 28
Appendix B: Vagrant Basebox BASH Script ........................................ 30
Appendix C: Vagrantfile for the Simulated Environment ....................... 33
Appendix D: Ansible Playbook for the Simulation .................................. 36
Appendix E: Ansible Playbook for the Firewall

Appendix F: Ansible Tasks for a Firewall API Call

Appendix G: Ansible Tasks for a Firewall External Zone

Appendix H: Ansible Tasks for a Firewall Internal Zone

Appendix I: BASH/BATS Tests for the External Host

Appendix J: BASH/BATS Tests for the Internal Host
Introduction

According to Lopez (2014) of Gartner Inc., the internet will connect at least 30 billion devices by the year 2020, continuing the trend towards all business becoming digital business. This has placed increasing demands on Information Technology (IT) teams to become faster and more efficient, often with fewer resources. Older development processes such as the waterfall method focus on heavy design and documentation before any development begins, resulting in an inability to react to change. This means that mistakes made early in the development process are very costly to correct later. Agile software development practices have evolved over time to keep pace with the needs of constantly changing digital business by using techniques such as iterative development, test-driven development, automation, and continuous delivery in order to allow for quick reaction to changing requirements (Abbas, Gravell, & Wills, 2008; Larman & Basili, 2003). However, software development is only one of several services within IT necessary to produce and deliver technology to support digital business. Other services include infrastructure, operations, information security, and more. All too often these different services within IT function independently from each other in silos.

The Oxford English Dictionaries offer several definitions of “silo” (n.d.), including "a system, process, department, etc. that operates in isolation from others." The presence of silos within a business is widely considered detrimental to efficiency, morale, productivity, and the overall health of an organization (Gleeson & Rozo, 2013; Gulati, 2007; Thompson, 2013). While silos can be a problem in every part of an organization (Gleeson & Rozo, 2013; Gulati, 2007), we would like to focus on the idea that silos have a negative impact on IT departments (Gleeson & Rozo, 2013). Thompson (2013) also asserts that it is essential to review current processes in order to adapt or change them to meet the demands of digital
EXPLORING TEST-DRIVEN NETWORK SECURITY

business in the context of more cross-functional teams.

A 2013 Gartner report (as cited in [Thompson, 2013]) claims a major concern is that IT infrastructure and operations are unable to meet the growing needs of digital business fast enough. One could make the case that this is in part due to development teams and operational teams existing in silos. In response to this concern many IT departments are reintegrating development and operations ([Erich, Amrit, & Daneva, 2014]), a movement that has been termed DevOps, in order to combat the inefficiencies of siloed teams. In reviewing the limited research that exists regarding DevOps, [Erich et al. 2014] makes note that collaboration and automation are important aspects of DevOps culture and that agile development plays an important role. This has also resulted in infrastructure and operations becoming more agile, facilitated by the growing ubiquity of server virtualization, software defined infrastructure, and infrastructure as code. This has led to a growing interest in applying test-driven principles to infrastructure in what is now being called test-driven infrastructure.

Unfortunately, while DevOps seems to be improving the ability of development and operations to deliver the needs of business ([Erich et al., 2014]), information security is not currently a part of DevOps and remains siloed from other teams within IT ([Chesla, 2016; Reeves, 2016]). This is made worse by the extreme shortage of necessary skilled information security workers. There are currently one million unfulfilled jobs ([Culbertson, Humphries, Ivy, Kolko, & Rodden, 2017]), a number that is predicted to grow to 1.5 to 2 million by 2019 ([Culbertson et al., 2017; Kauffman, 2017]). According to Reeves (2016), 'while it’s true that the Agile and DevOps movements have made security more difficult because of the volume of change occurring on a near-daily basis, it is totally not true that accelerated release cycles and secure applications are mutually exclusive.' Chesla (2016) identifies several methods of dealing with security issues related to silos, including consolidating groups within IT, implementing orchestration and automation tools, and making sure security policies can achieve their intended outcome.
In summary, IT professionals face serious challenges in trying to meet the growing needs of digital business. Agile practices have shown benefits for development, and DevOps has helped align operations and infrastructure with development in order to breakdown silos and foster more efficient processes through collaboration and automation. As information security now faces a shortage in skilled labor and severe challenges in meeting the needs of business, it is possible that information security can also benefit from adopting agile principles in order to break down their own silos and enable security to integrate with development and operations. This paper explores one agile practice, namely Test-Driven Development (TDD), and how it can be applied to network security policy. The case for applying test-driven principles to security is made by by 1) reviewing relevant literature on the benefits of TDD as it applies to software development; 2) reviewing DevOps practices and tools used to enable infrastructure-as-code, software-defined infrastructure, and the emerging practice of Test-Driven Infrastructure (TDI); and 3) developing a proof-of-concept implementation showing how TDI can be leveraged to take a test-driven approach to developing network security policy. The conclusion discusses the limitations of this work and proposes future research.

**Literature Review**

**Benefits of Test-Driven Development**

The benefits of TDD have been greatly debated. Early studies of teams at IBM show strong evidence that switching from previous methods of development to TDD reduced the number of defects in the software produced by as much as 50%, as well as show strong evidence of improved code and system design, better integration of code and components throughout a project, and no impact on productivity (Maximilien & Williams, 2003; Williams, Maximilien, & Vouk, 2003). Other studies show quantitative benefits of TDD such as improved code quality, but offer conflicting conclusions on productivity, such as Canfora, Cimitile, Garcia, Piattini, and Visaggio (2006), George and Williams (2004), and
Nagappan, Maximilien, Bhat, and Williams (2008) who found productivity decreased versus Erdogmus, Morisio, and Torchiano (2005) who found an increase in productivity.

In a longer term study covering five years of development at IBM, Sanchez, Williams, and Maximilien (2007) found TDD maintained an increase in code quality with a decrease in productivity, but also found that TDD reduced the code complexity of a given software product, yielding more reliable code over time and likely compensating for any initial decrease in productivity. Marchenko, Abrahamsson, and Ihme (2009) found similar long term benefits in a study of a Nokia Siemens Networks team that used TDD for 3 years, showing an increase in code quality as well as simplified software maintenance over time, but showing no effect on productivity.

Given the inconsistent findings, several studies were conducted that compiled and analyzed the results of previous literature on the benefits of TDD. Rafique and Mišić (2013) analyzed 27 previous studies to determine the impact of TDD on code quality and productivity and found that the majority of previous work agreed on an improvement to code quality using TDD with little to no impact to productivity. It is worth noting that Rafique and Mišić (2013) also found a significant positive correlation between the size and complexity of the programming tasks and the amount of improvement in code quality. A more detailed qualitative analysis by Mäkinen and Münch (2014) of previous empirical studies on the effects of TDD shows support for reduced defects, more maintainable code, a smaller and less complex code base, but an increase in initial effort in development. The most detailed review found of previous work was conducted by Bissi, Neto, and Emer (2016) and found that in previous studies on the effects of TDD on code quality between 1999 and 2014, 57% used experiments, 32% used case studies, with between 76% and 88% of studies showing an improvement in code quality using TDD and 4% of the studies showing a decrease in productivity.

Much of the work examined had a wide range of parameters, so when determining an application of TDD to a specific use case one should review the literature that pertains to
that case. For the purpose of this paper it is enough to show that TDD can have benefits, and what those potential benefits are. The literature reviewed shows that TDD can have definite benefits, particularly improved quality of the code written, fewer defects, and lower maintenance effort over time. This previous work also shows that the scale of the benefits of TDD over traditional methods increases with the size and complexity of the programing tasks, or more simply stated that larger more complex projects show a higher level of a given benefit. According to Fucci, Erdogmus, Turhan, Oivo, and Juristo (2016), who also found that using TDD showed improvement in code quality, "the claimed benefits of TDD may not be due to its distinctive test-first dynamic, but rather due to the fact that TDD-like processes encourage fine-grained, steady steps that improve focus and flow." At this point that statement is mere supposition, and even if evidence is later shown to support the idea, it does not negate the previous works which show clear potential benefits to using TDD. The possibility that other methods may be as effective as TDD does not diminish the effectiveness and benefits of using a TDD approach.

Erdogmus et al. (2005) elaborate on the benefits of TDD over more traditional approaches to software development when they discuss what they call “points of view” in relation to TDD:

- feedback: tests provide feedback as to whether or not the system is functioning as intended, and that new code does not break existing functionality
- task-orientation: tests drive activity, encourage the decomposition of problems into small manageable tasks, and provide focus on measurable progress
- quality-assurance: running of tests ensures the intended function
- design: tests provide context for design decisions

Erdogmus et al. (2005) also discuss previous studies that suggest other benefits may include enhanced understanding by the developers of the software being written;
specifically that test-first programming encourages better decomposition of problems, improves understanding of requirements, reduces the scope of the tasks to be performed, and reduces the amount of rework on a project. This, in conjunction with the claim by Fucci et al. (2016) that the benefits of TDD may be "due to the fact that TDD-like processes encourage fine-grained, steady steps that improve focus and flow", supports the assertion that the processes introduced by using TDD can yield similar results in information security domains.

**Principles of Test-Driven Development**

The study by Erdogmus et al. (2005) used the typical approach to test-driven development, called test-first, where tests are written before the code to be tested. This is a key aspect of test-driven development and allows for the workflow in which programmers write a test that fails, then write the code that makes the test pass. This is sometimes called the red-green-refactor approach: red for tests failing, green for tests passing, and refactor for optimizing the code. Of particular note, programmers can refactor with more confidence and there is less chance of changes in the code negatively altering the intended functionality because these tests exist. This process is more specifically detailed by Maximilien and Williams (2003):

With TDD, before writing implementation code, the developer writes automated unit test cases for the new functionality they are about to implement. After writing test cases that generally will not even compile, the developers write implementation code to pass these test cases. The developer writes a few test cases, implements the code, writes a few test cases, implements the code, and so on. The work is kept within the developer’s intellectual control because he or she is continuously making small design and implementation decisions and increasing functionality at a relatively consistent rate.
Behavior-Driven Development

To expand on the idea of the test-first dynamic of TDD, Erdogmus et al. (2005) outline a workflow based on what are called user-stories, or descriptions of a specific piece of intended functionality

1. Pick a story.
2. Write a test that expresses a small task within the story and have the test fail.
3. Write production code that implements the task to pass the test.
4. Run all tests.
5. Rework production code and test code until all tests pass.
6. Repeat 2 to 5 until the story is fully implemented.

This workflow outlines what became a slight variation of TDD, called Behavior-Driven Development (BDD), that is heavily utilized in agile development processes. Traditional TDD in software development will often focus on what are called unit tests. These are tests of the smallest units of functionality that together compose the overall functions of a system. In order to test the entire system, other types of testing are employed such as regression, integration, and acceptance. This approach can be viewed as an inside-out approach, first testing the discrete units of functionality and moving outward to more generalized tests of the system. According to Solís and Wang (2011), "BDD is focused on defining fine-grained specifications of the behaviour of the targeting system, in a way that they can be automated." The key here is the focus on specifying the "behavior" of the system, not the system's discrete functions. Solís and Wang (2011) define 6 characteristics of BDD based on previous research, but the one we will pay attention to is the idea that BDD is an "Iterative Decomposition Process", in which
the analysis starts with identification of the expected behaviours of a system, ... derived from the business outcomes it intends to produce. Business outcomes are then drilled down to feature sets. A feature set splits a business outcome into a set of abstract features, which indicate what should be done to achieve the business outcome.

The idea of decomposition represents an outside-in approach, starting with generalized behaviors and working to more discrete units of functionality. This approach has the benefit of more clearly focusing on and defining the business requirements of a system in the beginning and ensuring the system meets the correct business need.

The implementation being developed for this work will not be sticking strictly to traditional TDD nor specifically to BDD, but the principles of BDD heavily influenced the direction. Strict BDD often relies on specific tools which may not function in the test environment developed, but they may be explored as appropriate. The key aspect of BDD for the purpose of this work is a process of decomposition to define system behaviors that meet a desired business outcome.

**DevOps and Test-Driven Infrastructure**

DevOps has become a buzzword in the IT world and is frequently viewed as a solution to the problem of IT infrastructure operations being unable to keep pace with agile software development. [Duvall][2011] discusses some of the issues that have led to the widespread adoption of DevOps principles, specifically that most organizations reduce the reliability and repeatability of production environments through several practices including: manual installation and configuration of resources, which takes far too long and can not easily be recreated; a process sometimes referred to as document-driven provisioning in which a team attempts to document every step of installation and configuration, which quickly becomes outdated, incorrect, or not understandable; some teams attempt to automate the installation and configuration of systems by a collection of
ad-hoc scripts which generally becomes difficult to maintain, leading to the belief that the system is too difficult to automate.

Duvall (2011) provides a 5-step process to successfully implement a continuous process for infrastructure that closely resembles that in use by agile development using TDD/BDD:

1. document, or outline the system requirements and describe the installation, configuration, and management of each component
2. test, or more specifically produce automated tests
3. script, which is the actual automation of the installation and configuration process, often using configuration management tools such as Chef or Puppet
4. version, generally using a version control system such as git, and including versioning of the tests as well as the installation and configuration scripts
5. continuous, which means the process must be automated to run with little or no human interaction

Duvall (2011), Kulesza (2014), and Nelson-Smith (2013) all provide specific examples for implementing a Test-Driven Infrastructure (TDI) that can potentially provide similar benefits as TDD/BDD does for software development. Nelson-Smith (2013) goes further to suggest that "infrastructure can and should be treated as code" and that "infrastructure developers should adhere to the same principles of professionalism as other software developers." Whether you agree or not, the prevalence of virtualization, infrastructure-as-a-service, software-defined infrastructure, and infrastructure-as-code provides an environment where TDI is possible, feasible, and even desirable. While the tooling available for TDI is not as advanced as that available for TDD/BDD, the foundation exists and can be leveraged for other domains as well, specifically pertaining to this work for information security and the development of network security policy.
Methodology

In order to do more quantitative studies on the benefits of a test-driven approach within information security domains, there needs to be clearly defined tools, methods, and best practices. While these items are always changing, they are well established within the domain of software development, and TDD is a fairly well researched field. However, for information security to become more agile, there is a danger in relying on the tools, methods, and best practices of the DevOps movement and the state of the tools and practices within TDI. As of the time of publication for and according to Erich et al. (2014), there was little quality research on DevOps. In addition, the state of tools and practices used to implement TDI are complex and still evolving, in part due to the fact that testing infrastructure presents unique challenges not addressed by traditional software TDD/BDD (Nelson-Smith 2013). In determining the methodology, environment, and tools for this works proof of concept, possible options are mentioned and briefly discussed as appropriate before listing the criteria and reasons for the selections made. In general it is the goal to keep the tools simple and portable while outlining alternatives that could be considered. Automation and treating infrastructure as code in a version control system are also key goals. This is a complex and quickly changing field of practice and requires some creativity and experimentation.

Testing Platform

Workstation Operating System. Ideally the choice of a workstation operating system should not matter, and the process used should be usable across platforms. In reality this is not always the case. Some tools are only available on specific platforms, and many that are available on different platforms may not support the same features or may be more difficult to install and configure. For example, when considering a virtualization platform to use, Hyper-V will only be available on Windows and KVM is only available on Linux. VMware workstation is available on all platforms but is more complicated to setup.
on Linux, while VirtualBox is easy to set up on all platforms. The choice of hypervisor is discussed more more later on. Given the complexity of choices, Linux will be used for the workstation operating system for the testing environment. While it does require more knowledge and experience, it opens up considerably more options for testing tools and methodologies. In theory, the same processes should generalize across platforms with enough work and careful tool selection.

Virtualization Tools. The next important choice is that of a virtualization platform, or hypervisor. In this area there are several categories of choices. The virtualization can take place on a single workstation, on a datacenter or cluster virtualization environment, or using Infrastructure as a Service (IaaS) in the cloud. Cloud options include those offered by Amazon AWS, RackSpace (which uses OpenStack under the hood), and possibly others. Platform as a Service (PaaS) in general would not be suitable for this work, since the ability to simulate real networks and servers as close to production as possible is required, and the nature of PaaS abstracts out the underlying infrastructure by its design and purpose. For this work it is necessary to control and tap into that underlying infrastructure. That may be use-cases for using PaaS to implement TDD for security policy, but is unsuitable for this work. Additionally, cloud-based infrastructure such as that offered by Amazon AWS and RackSpace can offer very unique tools and workflows in particular as a part of a continuous delivery system, but these cost money and are not necessarily accessible to everyone. The immediate goal is to use the solution with the easiest path to entry and the most versatility while being as simple as possible. While sometimes requiring more technical knowledge to implement, keeping the development on a local workstation allows the widest available options of tools and workflows while retaining a level of simplicity, a low barrier to entry, and the most control.

The same assessment applies to in-house datacenter-based virtualization solutions such as VMware ESX/vCenter, OpenStack, and Hyper-V based clusters. These can and should be considered but are beyond the scope of this work. There are also other tools that
muddy the water and can be considered as a part of a TDI security workflow, including other virtualization platforms and tools such as Xen, Qemu, and even container systems such as Docker. However, those complicate the workflow and therefore will not be considered for this work. This leaves us with several choices of hypervisor that work on our selection of Linux as a workstation platform.

Linux supports Xen, KVM, VMware Workstation, and VirtualBox as fairly common virtualization solutions. In addition, many of these tools can be managed with their own tools from a Graphical User Interface (GUI) or Command Line Interface (CLI), as well as from generalized tools such as libvirt and virt-manager that can manage multiple hypervisors. Libvirt and virt-manager in particular can manage all the supported hypervisors mentioned, namely Xen, KVM, VMware workstation, and VirtualBox, though they are best suited for managing Xen and KVM. Libvirt is a CLI tool while virt-manager is GUI front-end to libvirt. Most of the native tools for each hypervisor can easily be scripted from a Linux environment, as can libvirt/virt-manager, which could be very important in our attempt to automate as much as possible.

Platform Summary and Other Tools. There is also one more tool in particular to examine, and that is Vagrant. Vagrant is a cross platform tool written in the Ruby programming language that allows for the specificatin of sets of virtual machines in a configuration file that can be added to version control. Those machines can then be created and destroyed on demand. Vagrant has some drawbacks however. It natively supports VirtualBox, has third-party support for KVM and libvirt, and while it does have VMware support is a paid commercial product. Vagrant is also limited in how it can configure virtual machines and networking since it makes specific assumptions and attempts to abstract functionality across multiple hypervisors. Many of these shortcomings can be remedied by scripting in addition to the native Vagrant functionality to modify or add features that are supported by a native hypervisor but not by Vagrant. Vagrant was also originally designed to be simple and for single or very simple sets of virtual machines
and makes some assumptions. One such assumption is that every virtual machine has and is required to have a NAT internet connection, which may not be desirable if we are trying to force all traffic through another interface and through our firewall. These issues are discussed in greater detail in the results section.

Given this brief overview of platform tools, the preference is to utilize Vagrant with VirtualBox as much as possible in order to maintain simplicity and the potential for cross-platform compatibility. Custom scripting is used to remedy any issues presented by the limitations of these tools. Other miscellaneous tools are used as necessary to setup and configure the virtual test environment. Specifically this work makes heavy use of the configuration management tool Ansible for setting up the host and virtual machines. This has the added benefit being usable to configure production environments if used correctly. Bash command line scripts and GNU Make are also used for generating the test environment, with the idea that these portions of functionality are not part of the actual development process and can be ported to the same or similar native tools on operating systems other than Linux.

**Simulated Network Architecture**

The goal for the proof of concept is to keep things as simple as possible in order to focus on the process. The implementation consists of 2 separate virtual networks with a single virtual host per network. These 2 networks are referred to as the internal and external zones and are separated by a firewall. The initial security policy to test is to allow the host within the internal zone to initiate connections with the host in the external zone, but disallow the host within the external zone to initiate connections with the host in the internal zone (see Figure 1).
**Findings**

This section reviews the actual development of the proof of concept implementation. The discussion will be a chronological timeline of the tasks undertaken, the original intent of each task, issues that were discovered, and the final resolutions for each task. The development workstation consisted of a laptop with an Intel(R) Core(TM) i7-6600U CPU and 16GB of RAM running the Arch Linux operating system. Specific versions of software used include GNU Make 4.2.1, Vagrant 2.0.0, VirtualBox 5.2.0, Ansible 2.4.1, and BATS 0.4.0. The full source code for the implementation is maintained in a git repository (Sigsby, 2017).

The first step is to generate the virtualized test environment. Vagrant is designed to create virtual hosts using a preconfigured template called a basebox. There are a number of existing baseboxes available for Vagrant, and the official CentOS 7 baseboxes are used for both our internal and external hosts. Using Linux for these hosts simplifies the implementation by allowing the use of the BATS testing framework with direct access to common command line utilities such as ping, traceroute, dig, and many more. In theory these hosts could just as easily be other operating systems such as Windows, but this would complicate the selection of testing tools and the point is to test firewall policy, not specific server or host functionality. In expanding this work later it would make sense to explore other host operating systems in order to more closely model real production environments.
While the components for implementing the hosts within each zone are readily available, this is not the case when it comes to the firewall. There are a number of choices for implementing the firewall for this test environment, including another Linux virtual host with multiple network cards using the native iptables or firewalld, the open source version of pfSense, or virtual versions of commercial firewalls. This project uses a virtual version of the Palo Alto firewall running PAN-OS 7.1 as that is the system with the most direct applications to the author’s current work. The tests are independent of the specific firewall and should work for any firewall selected. In order to use the chosen firewall, it is necessary to generate a Vagrant basebox. This is done by importing the virtual image of the Palo Alto firewall into VirtualBox, configuring it for Vagrant, and then exporting the VirtualBox machine as a basebox using native Vagrant tools. See Appendix A and B for the Makefile and helper scripts used to automate this process.

The next step is to create the virtual components within the test environment. This is done using Vagrant to specify the virtual machines and networks to achieve the desired architecture as shown in Figure 1, with the resulting network shown in more detail by Figure 2. At first the environment was partially configured using Vagrant and its integration with Ansible while other parts were configured manually on each virtual machine and verified manually as well. Manual steps were automated of several iterations. The end result is an automated process using Vagrant and Ansible scripts (see Appendices C through H) to create and destroy the test environment.
Of particular note in regards to the simulated environment was the feature of Vagrant that configures every machine's first network interface as a NAT interface with internet access. This interface is necessary in order to install some initial software on the internal and external hosts, but for policy testing the machines need to be completely isolated from outside networks in order to ensure the correct routing through the firewall as well as to eliminate any risk of testing malicious behavior from reaching outside of the test environment. This was accomplished by setting the default routing to the firewall interfaces, making it impossible for traffic to be routed to external networks over the NAT interface. The Ansible scripts used to setup the base configuration of the internal and external hosts then temporarily set the default routing to the NAT interface in order to install software from internet repositories, and then immediately change the default routing back to the simulation interface. This was verified by writing tests to verify the simulated environment, but this presented a potential issue.
In writing tests to verify the test environment, it is sometimes confusing as to what was being tested. Many of these tests had no use in the production environment and applied only to verifying our test environment. These tests were necessary but added significant effort that had little or no direct application on a final production deployment. It is unclear how much this detracted from the focus of the project, namely to demonstrate TDD for the development of security policy. However, this extra effort is consistent with findings covered in the literature review, particularly the study by Mäkinen and Münch (2014) which reviewed previous empirical studies and showed that TDD generally created an increase in initial effort but yielded long term benefits such as reduced defects, easier maintenance over time, and less complexity.

As the items that needed to be checked in order to verify the environment became more clear, several points came into focus. It quickly became obvious that the ability to automatically destroy and recreate the environment from scratch was useful in order to build up the configuration from a known base state. One issue encountered however is that while recreating the environment is automated, it still took a considerable amount of time. The firewall in particular took between 5-10 minutes to boot, and bringing up the environment completely from scratch (not including the generation of the firewall Vagrant basebox) took anywhere between 15-20 minutes on the test hardware. This inconvenience was improved by the ability to snapshot the virtual machines in a booted but unconfigured state.

Over repeated iterations of manual steps and then automating them with Vagrant and Ansible, a process for managing the test environment emerged.

1. Bring up the test environment and provision to the latest state
2. Run all tests to verify the current state is valid (all tests should pass)
3. Determine the goal for a new security policy
4. Write a test for the new policy
5. Run the new test (the test should fail)

6. Write the code to provision the new policy

7. Provision the new policy and run the test

8. Repeat the previous 2 steps until the test passes

9. Run all tests to verify the new state is valid

This demonstrates a clear test driven approach similar to that as outlined by Erdogmus et al. (2005) in the literature review. The process outlined above could also be applied to multiple iterations of development and turned into a continuous process as defined in the literature review by Duvall (2011).

Conclusions

In conclusion, this work demonstrates that developing network security policy can be achieved with the principles of TDD using the tools of TDI. However, there are a number of issues that still need to be addressed before future work can focus on empirically measuring the possible benefits. First, the automated policy provisioning and tests as produced by this implementation are not directly deployable production systems. There needs to be further refinement in order to keep the policy provisioning and test components separate from those that apply only to the test environment.

Due to the time constraints for finishing this work, a greater portion of time was spent discovering and resolving issues with test environment itself, rather than focusing on the actual TDD process as it applies to security policy. Unfortunately, there is significantly more work to be done to improve the usability of the test environment itself. Ideally a process for generating a test environment could be generalized better so it could be applied to alternative tools including other virtualization environments, a much broader range of testing tools, and other network components including other firewalls, more types of network gear such as router, switches, etc., and more complex network topologies.
It is currently planned to continue this work by refining and simplifying the test environment, exploring more types of tests along with more testing tools, developing a much more comprehensive set of test cases and scenarios, and adapting what has been learned here into a continuous process that can applied to a production network. Once that is achieved it will be possible to pursue more empirical research into the long term effects of applying the principles of TDD to the development of network security policy.


31(3), 226-237. doi: 10.1109/TSE.2005.37


Appendix A

Firewall Basebox Makefile

# Makefile variables
#

tmp := tmp
key := $(tmp)/id_rsa
version := PA-VM-7.1.0
img := $(tmp)/$(version).ova
ssh := 2222
https := 8443
user := admin
pass := admin
box := $(tmp)/$(version).box
name := msia/$(version)

.PHONY: default
default:
  @echo 'default task not implemented'
$(tmp):
  mkdir $(tmp)
$(tmp)/id_rsa: | $(tmp)
  @echo 'Generating authentication keys...'
  ssh-keygen -f $(key) -t rsa -N '
$(key).pub: | $(key)

# Vagrant Tasks
$(box): | vbox-start
  @echo 'Packaging vagrant basebox...'
  vagrant package --base $(version) --output $(box)

.PHONY: vagrant-package
vagrant-package: $(box)
  vagrant box add --name $(name) $(box)

.PHONY: vagrant-clean
vagrant-clean:
  if vagrant box list | grep $(name) > /dev/null 2>&1 ; then \
    echo 'Removing vagrant box...'
    vagrant box remove $(name) ; \
  fi
  if [[ -f $box ]] ; then rm $(box) ; fi

# VirtualBox Tasks
.PHONY: vbox-import
vbox-import:
EXPLORING TEST-DRIVEN NETWORK SECURITY

@if ! vboxmanage list vms | grep $(version) > /dev/null 2>&1 ; then \n    echo '∗ Importing machine image...'; \n    vboxmanage import $(img); \nfi

.PHONY: vbox-start
vbox-start: vbox-import $(key).pub
@if ! vboxmanage list runningvms | grep $(version) > /dev/null 2>&1 ; then \n    echo '∗ Configuring machine settings...'; \n    if ! vboxmanage list hostonlyifs | grep 'vboxnet0'; then \n        vboxmanage hostonlyif create; \n    fi; \n    if ! vboxmanage showvminfo $(version) | grep 'NIC 1.Attachment: NAT'; then \n        vboxmanage modifyvm $(version) --nic1 nat; \n    fi; \n    if ! vboxmanage showvminfo $(version) | grep 'NIC: Rule: ssh'; then \n        vboxmanage modifyvm $(version) --natpf1 'ssh, tcp, $(ssh), 22'; \n    fi; \n    if ! vboxmanage showvminfo $(version) | grep 'NIC: Rule: https'; then \n        vboxmanage modifyvm $(version) --natpf1 'https, tcp, $(https), 443'; \n    fi; \n    if ! vboxmanage showvminfo $(version) | grep 'NIC/uni24232.Attachment:/uni2423Host-only/uni2423Interface/uni2423/vboxnet0'; then \n        vboxmanage modifyvm $(version) --nic2 hostonly --hostonlyadapter2 vboxnet0; \n    fi; \n    echo '∗ Starting machine...'; \n    vboxmanage startvm $(version); \nfi

@while ! curl -k --connect-timeout 10 https://localhost:$(https) > /dev/null 2>&1 ; do \n    echo -n '. '; sleep 5 ; \ndone

bin/vbox-config $(version) $(user) $(pass) $(key).pub

.PHONY: vbox-poweroff
vbox-poweroff:
@if vboxmanage list runningvms | grep $(version) > /dev/null 2>&1 ; then \n    echo '∗ Forcing machine off...'; \n    vboxmanage controlvm $(version) poweroff; \nfi

.PHONY: vbox-clean
vbox-clean: vbox-poweroff
@if vboxmanage list vms | grep $(version) > /dev/null 2>&1 ; then \n    echo '∗ Removing machine...'; \n    vboxmanage unregistervm $(version) --delete; \nfi
Appendix B

Vagrant Basebox BASH Script

#!/bin/bash

# get cli arguments
declare -rx vmm="$1"
declare -rx username="$2"
declare -rx password="$3"
declare -rx keyfile="$4"

# helper functions
## send individual characters/keys
function send_key() {
  # http://www.comptechdoc.org/os/linux/howlinuxworks/linu_keycodes.html
  case $1 in
    RETURN|ENTER) t="1c, 9c" ;;
    !) t="2a, 02, 82, aa" ;;
    1) t="03, 83" ;;
    0) t="2a, 03, 83, aa" ;;
    3) t="04, 84" ;;
    #) t="2a, 04, 84, aa" ;;
    4) t="05, 85" ;;
    $) t="2a, 05, 85, aa" ;;
    5) t="06, 86" ;;
    %) t="2a, 06, 86, aa" ;;
    6) t="07, 87" ;;
    \) t="2a, 07, 87, aa" ;;
    7) t="08, 88" ;;
    \d) t="2a, 08, 88, aa" ;;
    8) t="09, 89" ;;
    *) t="2a, 09, 89, aa" ;;
    9) t="0a, 8a" ;;
    () t="0b, 8b" ;;
    \b) t="2a, 0b, 8b, aa" ;;
    __) t="0c, 8c" ;;
    ___) t="2a, 0c, 8c, aa" ;;
    =) t="0d, 8d" ;;
    \+) t="2a, 0d, 8d, aa" ;;
    q) t="10, 90" ;;
    Q) t="2a, 10, 90, aa" ;;
    \w) t="11, 91" ;;
    W) t="2a, 11, 91, aa" ;;
    \v) t="12, 92" ;;
    E) t="2a, 12, 92, aa" ;;
    \r) t="13, 93" ;;
    R) t="2a, 13, 93, aa" ;;
    \t) t="14, 94" ;;
    T) t="2a, 14, 94, aa" ;;
    \y) t="15, 95" ;;
    Y) t="2a, 15, 95, aa" ;;
    \u) t="16, 96" ;;
    U) t="2a, 16, 96, aa" ;;
### send_string_one_character_at_a_time

```bash
function send_string() {
    echo "invalid character \${c}"; exit 1;
}
esac
vboxmanage controlvm \"$\{vm}\" keyboardputscancode \$\{t\} }
```

### send_string_as_a_line(ending_with\return/enter)

```bash
function send_line() {
    send_string \"${1}\"
    send_key \"RETURN\"
}
```

### send_lines

```bash
declare -rx key="\$(base64 \$\{keyfile\} | tr -d \"\n\")"
declare -ax lines=(\"\${username}\" \"\${password}\" \"set ssh-authentication public-key \$\{key\} \" configure \" commit\")
for i in \$\{lines \[@\]\}; do
    send_line \"\$\{i\}\"
done
sleep 1
sleep 30
send_line \"exit\"
sleep 1
send_line \"exit\"
sleep 1
```

---

**EXPLORING TEST-DRIVEN NETWORK SECURITY**

---

```
t="39 b9";
for e in invalid character \${c}; do exit 1; esac
vboxmanage controlvm \"$\{vm}\" keyboardputscancode \$\{t\} }
```

### send_string()

```bash
function send_string() {
    echo "invalid character \${c}"; exit 1;
}
esac
vboxmanage controlvm \"$\{vm\}\" keyboardputscancode \$\{t\} }
```

### send_string_as_a_line(ending_with\return/enter)

```bash
function send_line() {
    send_string \"${1}\"
    send_key \"RETURN\"
}
```

### send_lines

```bash
declare -rx key="\$(base64 \$\{keyfile\} | tr -d \"\n\")"
declare -ax lines=(\"\${username}\" \"\${password}\" \"set ssh-authentication public-key \$\{key\} \" configure \" commit\")
for i in \$\{lines \[@\]\}; do
    send_line \"\$\{i\}\"
done
sleep 1
sleep 30
send_line \"exit\"
sleep 1
send_line \"exit\"
sleep 1
```

---

**EXPLORING TEST-DRIVEN NETWORK SECURITY**

---

```bash
for ((i=0; i<\#$\{1\}; i++)); do
    send_key \"\${1: $i :1}\"
done
```

### send_line()

```bash
function send_line() {
    send_string \"${1}\"
    send_key \"RETURN\"
}
```

### send_lines

```bash
declare -rx key="\$(base64 \$\{keyfile\} | tr -d \"\n\")"
declare -ax lines=(\"\${username}\" \"\${password}\" \"set ssh-authentication public-key \$\{key\} \" configure \" commit\")
for i in \$\{lines \[@\]\}; do
    send_line \"\$\{i\}\"
done
sleep 1
sleep 30
send_line \"exit\"
sleep 1
send_line \"exit\"
sleep 1
```

---

**EXPLORING TEST-DRIVEN NETWORK SECURITY**

---

```bash
vm=\\echo \" invalid character \${c}\" ; exit 1 ;
}
esac
vboxmanage controlvm \"$\{vm\}\" keyboardputscancode \$\{t\} }
```

### send_string()

```bash
function send_string() {
    echo "invalid character \${c}"; exit 1;
}
esac
vboxmanage controlvm \"$\{vm\}\" keyboardputscancode \$\{t\} }
```

### send_string_as_a_line(ending_with\return/enter)

```bash
function send_line() {
    send_string \"${1}\"
    send_key \"RETURN\"
}
```

### send_lines

```bash
declare -rx key="\$(base64 \$\{keyfile\} | tr -d \"\n\")"
declare -ax lines=(\"\${username}\" \"\${password}\" \"set ssh-authentication public-key \$\{key\} \" configure \" commit\")
for i in \$\{lines \[@\]\}; do
    send_line \"\$\{i\}\"
done
sleep 1
sleep 30
send_line \"exit\"
sleep 1
send_line \"exit\"
sleep 1
```

---

**EXPLORING TEST-DRIVEN NETWORK SECURITY**

---

```bash
for ((i=0; i<\#$\{1\}; i++)); do
    send_key \"\${1: $i :1}\"
done
```

### send_line()

```bash
function send_line() {
    send_string \"${1}\"
    send_key \"RETURN\"
}
```

### send_lines

```bash
declare -rx key="\$(base64 \$\{keyfile\} | tr -d \"\n\")"
declare -ax lines=(\"\${username}\" \"\${password}\" \"set ssh-authentication public-key \$\{key\} \" configure \" commit\")
for i in \$\{lines \[@\]\}; do
    send_line \"\$\{i\}\"
done
sleep 1
sleep 30
send_line \"exit\"
sleep 1
send_line \"exit\"
sleep 1
```
Appendix C
Vagrantfile for the Simulated Environment

```ruby
cfg = {
  'firewall' => {
    'https' => '8443',
  },
}

[vagrant-vbguest', # https://github.com/dotless-de/vagrant-vbguest
  'vagrant-triggers', # https://github.com/emyl/vagrant-triggers/wiki/Trigger-recipes
  'vagrant-vbox-snapshot',
  'vagrant-hostmanager',
].each do |plugin|
  if !Vagrant.has_plugin?("#{plugin}")
    system("vagrant/plugin/uninstall/#{plugin}")
  end
end

Vagrant.configure('2') do |c|
  # vagrant-vbguest plugin
  # disable fetching/installing virtualbox guest additions by default
  c.vbguest.auto_update = false
  c.vbguest.no_remote = true
  c.hostmanager.enabled = false
  c.hostmanager.include_offline = true
  c.hostmanager.manage_host = false

  # global virtualbox settings
  c.vm.provider 'virtualbox' do |t|
    #t.gui = true
    t.customize [
      'modifyvm', :id, '-groupsv','/msia'
    ]
  end

  # vagrant shares the project folder with each vm, disable this by default
  c.vm.synced_folder '.', '/vagrant', disabled: true
  # c.vm.synced_folder '<host-path>', '<vm-mount-point>'

  # use vagrant-triggers and vagrant-vbox-snapshot plugins
  # to create a base snapshot of each vm before further provisioning
  c.vm.provision 'snapshot-basebox', type: 'trigger', run: 'always' do |t|
    t.fire do
      if ! 'vagrant snapshot list #(@machine.name)').match('base')
        run "vagrant/snapshot/take/#{@machine.name}/base"
      end
    end
  end

  # ansible provisioner
  # https://www.vagrantup.com/docs/provisioning/ansible_common.html
```
c.vm.provision 'provision-ansible', type: 'ansible' do |
  t |
  t.become = true
  t.playbook = 'ansible/vagrant.yml'
  t.tags = (ENV['TAGS'] ? ENV['TAGS'] : 'all')
  t.extra_vars = {}
  t.host_vars = {
    'firewall' => {
      'https' => cfg['firewall']['https']
    }
  }
end

# Palo Alto firewall host
# see Makefile for how this was created
# note that this vagrant box is unique
# it has no virtualbox guest additions (impossible for that machine?)
# the first interface is NAT and should have internet access
# this first interface should also be the management interface
h.vm.define 'firewall' do |
  h |
  h.vm.box = 'msia/PA-VM-7.1.0'
  h.vm.box_url = 'tmp/PA-VM-7.1.0.box'
  h.vm.hostname = 'firewall'
  h.vm.boot_timeout = 900
  h.vm.network 'forwarded_port', guest:22, host:2222, id: 'MGMT-SSH'
  h.vm.network 'forwarded_port', guest:443, host:cfg['firewall']['https']
  h.vm.network 'private_network', ip: '192.168.201.2', auto_config: false
end

# internal network interfaces
h.vm.network 'private_network', ip: '192.168.201.2', auto_config: false
# example routing when using dhcp w/ route added (192.168.56.0/24):
# Destination   Next-Hop   Interface
# 0.0.0.0/0   192.168.201.1 ethernet1/1
# 192.168.201.0/24   192.168.201.2 ethernet1/1
# 192.168.202.0/24   0.0.0.0

# external network interface

# external interface (bridge)
# doesn't work on all host NIC types (e.g., fails on USB phone tether?)
# h.vm.network 'public_network', auto_config: false

h.vm.provider 'virtualbox' do |
  t |
  t.gui = true
  t.name = 'msia.firewall'
end
h.ssh.username = 'admin'
h.ssh.private_key_path = 'tmp/id_rsa'
end
c.vm.define 'external' do |h|
  h.vm.box = 'centos/7'
  h.vm.hostname = 'external'
  h.vm.network 'private_network', ip: '192.168.202.3'
  h.vm.provider 'virtualbox' do |t|
    #t.gui = true
    t.name = 'msia.external'
  end
  c.vm.provision :hostmanager
end

c.vm.define 'internal' do |h|
  h.vm.box = 'centos/7'
  h.vm.hostname = 'internal'
  h.vm.network 'private_network', ip: '192.168.201.3'
  h.vm.provider 'virtualbox' do |t|
    #t.gui = true
    t.name = 'msia.internal'
    t.customize ["modifyvm", :id, "--groups', '/msia' ]
  end
  c.vm.provision :hostmanager
end
end
Appendix D

Ansible Playbook for the Simulation

---

- hosts: internal external

  tasks:

  - name: disable vagrant NAT as default route
    lineinfile:
      path: /etc/sysconfig/network-scripts/ifcfg-eth0
      create: no
      regexp: '^DEFROUTE=\n
      line: DEFROUTE=no
      register: eth0

  - name: restart network
    service:
      name: network
      state: restarted
      when: eth0.changed

  - name: add vagrant NAT as default route
    shell: |
      if ! ip route | grep -F 'default via 10.0.2.2'; then
        ip route replace default via 10.0.2.2
      fi
    changed_when: false

  - name: misc vagrant packages
    package:
      name={{ item }}
      state=installed
    with_items: [ traceroute, nmap, mc, tcpdump ]

  - name: delete vagrant NAT as default route
    shell: |
      if ip route | grep -F 'default'; then
        ip route del default
      fi
    changed_when: false

  - name: mc config
    copy:
      src=mc
      dest=/{{ item }}/config/
    with_items: [ 'root', 'home/vagrant' ]
    tags: dev

# add current users public key for ssh

  - name: current user ssh authorized keys
    authorized_key:
      user={{ item }}
      key={{ lookup('file', '~/ssh/id_rsa.pub') }}
    with_items: [ root, vagrant ]

  - name: should not be able to ping the internet
    command: ping -c1 google.com
    register: result
    changed_when: false
failed_when: not (result|failed)

- hosts: internal
tasks:

  - name: set default route / gateway
    lineinfile:
      path: /etc/sysconfig/network
      create: no
      regexp: ^GATEWAY=.*$
      line: GATEWAY=192.168.201.2
      register: network

  - name: restart network
    service:
      name: network
      state: restarted
      when: network.changed

  # - name: can ping switch
  #   command: ping -c 1 192.168.201.1
  #   changed_when: false
  #
  # - name: can ping firewall
  #   command: ping -c 1 192.168.201.2
  #   changed_when: false

- hosts: external
tasks:

  - name: set default route / gateway
    lineinfile:
      path: /etc/sysconfig/network
      create: no
      regexp: ^GATEWAY=.*$
      line: GATEWAY=192.168.202.2
      register: network

  - name: restart network
    service:
      name: network
      state: restarted
      when: network.changed

  # - name: can ping switch
  #   command: ping -c 1 192.168.202.1
  #   changed_when: false
  #
  # - name: can ping firewall
  #   command: ping -c 1 192.168.202.2
  #   changed_when: false

- import_playbook: firewall.yml
Appendix E

Ansible Playbook for the Firewall

---

```yaml
# http://panwansible.readthedocs.io/en/latest/
# https://github.com/PaloAltoNetworks/ansible
  - hosts: firewall
    connection: local
    gather_facts: false

  vars:
  - username: 'admin'
  - password: 'admin'
  - address: '{{ hostvars.firewall.ansible_host }}:{{ hostvars.firewall.https }}'
  - validate_certs: false

  tasks:

  - name: python dependencies
    pip:
      name: [ 'pan-python', 'pandevice', 'xmltodict' ]
      state: latest
      become: true
      with_items: [ ]

  - name: request api key
    uri:
      url: https://{{ address }}/api?type=keygen&user={{ username }}&password={{ password }}
      validate_certs: 'true'
      return_content: true
      register: request_api_key
    failed_when: 'not (request_api_key.content | search("response.*status.*=.*success"))'
    # - debug: var=request_api_key

  - name: get api key
    xml:
      xmlstring: '{ request_api_key.content }'
      xpath: '/response/result/key'
      content: text
      register: get_api_key
    # - debug: var=get_api_key

  - name: set api key
    set_fact:
      api_key: { get_api_key.matches.0.key }
    # - debug: var=api_key

  # <request cmd="set" obj="/config/devices/entry[@name='localhost.localdomain']/deviceconfig/setting" cookie="6592603819717516">
  - name: 'internal interface (ethernet 1/1)'
    include_tasks: tasks/firewall/request.yml
    vars:
      body: 'type=config
      &action=set
      &xpath=/config/devices/entry[@name='localhost.localdomain']/deviceconfig/setting
      &element=<auto-mac-detect>yes</auto-mac-detect>,'

  - name: configure internal zone
```
EXPLORING TEST-DRIVEN NETWORK SECURITY

include_tasks: tasks/firewall/internal.yml

- name: configure external zone
  include_tasks: tasks/firewall/external.yml

# <request cmd="edit" obj="/config/devices/entry[@name='localhost.localdomain']/network/virtual-router/entry[@name='default ']
  - name: 'default router '
  include_tasks: tasks/firewall/request.yml

  vars:
    body: "type=config
    &action=edit
    &xpath=/config/devices/entry[@name='localhost.localdomain']/network/virtual-router/entry[@name='default ']
    &element=<entry name='default '>
    <protocol>
    <bgp>
    <enable>no</enable>
    <dampening-profile>
    <entry name='default '>
    <cutoff>1.25</cutoff>
    <reuse>0.5</reuse>
    <max-hold-time>900</max-hold-time>
    <decay-half-life-reachable>300</decay-half-life-reachable>
    <decay-half-life-unreachable>900</decay-half-life-unreachable>
    <enable>yes</enable>
    </entry>
    </dampening-profile>
    <routing-options>
    <graceful-restart>
    <enable>yes</enable>
    </graceful-restart>
    </routing-options>
    </bgp>
    </protocol>
    </bgp>
    </ecmp>
    <algorithm>
    <ip-modulo/>
    </algorithm>
    </ecmp>
    </interface>
    <member>ethernet1/1</member>
    <member>ethernet1/2</member>
  </interface>
</entry>"

# demo rule deploy
# <request cmd="set" obj="/config/devices/entry[@name='localhost.localdomain ']/vsys/entry[@name='vsys1 ']/rulebase/security/rules/entry[@name='Allow internal to external ']
  - name: 'security policy: allow internal to external '
  include_tasks: tasks/firewall/request.yml

  vars:
    body: "type=config
    &action=set
    &xpath=/config/devices/entry[@name='localhost.localdomain ']/vsys/entry[@name='vsys1 ']/rulebase/security/rules/entry[@name='Allow internal to external ']
    &element=<to>
    <member>external </member>
  </to>"
<from>
  <member>internal</member>
</from>
<source>
  <member>any</member>
</source>
<destination>
  <member>any</member>
</destination>
<source-user>
  <member>any</member>
</source-user>
<category>
  <member>any</member>
</category>
<application>
  <member>any</member>
</application>
<service>
  <member>application-default</member>
</service>
<hip-profiles>
  <member>any</member>
</hip-profiles>
<action>allow</action>

- name: 'commit'
  include_tasks: tasks/firewall/request.yml
  vars:
    body: "type=commit&cmd=<commit></commit>"
Appendix F
Ansible Tasks for a Firewall API Call

- name: request
  uri:
    url: https://{{ address }}/api?key={{ api_key }}
    validate_certs: '{{ validate_certs }}'
    return_content: true
    method: POST
    headers:
      'Content-type': 'application/x-www-form-urlencoded'
  body: '{{ body }}'
  register: request
  failed_when: 'not (request.content | search("response.*status.*success"))'
  # - debug: var=request
Appendix G

Ansible Tasks for a Firewall External Zone

- name: 'interface management profile - ping from external'
  include_tasks: request.yml
  vars:
    body: "type=config
          &action=set
          &xpath=/config/devices/entry[@name='localhost.localdomain']/network/profiles/interface-management-profile/entry[@name='Ping from external']
          &element=<ping>yes</ping>
          <permitted-ip>
            <entry name='192.168.202.0/24'/>
          </permitted-ip>"

- name: 'external interface (ethernet 1/2)
  include_tasks: request.yml
  vars:
    body: "type=config
          &action=set
          &xpath=/config/devices/entry[@name='localhost.localdomain']/interface/ethernet/entry[@name='ethernet1/2']
          &element=<layer3>
            <ipv6>
              <neighbor-discovery>
                <enable>no</enable>
              </neighbor-discovery>
            </ipv6>
            <ndp-proxy>
              <enabled>no</enabled>
            </ndp-proxy>
            <ip>
              <entry name='192.168.202.2/24'/>
            </ip>
            <interface-management-profile>Ping from external</interface-management-profile>
            <lldp>
              <enable>no</enable>
            </lldp>
          </layer3>"

- name: 'external zone'
  include_tasks: request.yml
  vars:
    body: "type=config
          &action=set
          &xpath=/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name='vsys1']/zone/entry[@name='external']
          &element=<network>
            <layer3>
              <member>ethernet1/2</member>
            </layer3>
          </network>"
Appendix H

Ansible Tasks for a Firewall Internal Zone

```yaml
# <request cmd="set" obj="/config/devices/entry[@name='localhost.localdomain']/network/profiles/interface-management-profile/entry[@name='Ping from internal']" cookie="6592603819717516" newonly="yes">
    name: 'interface management profile ping from internal'
    include_tasks: request.yml
    vars:
        body: "type=config &action=set &xpath=/config/devices/entry[@name='localhost.localdomain']/network/profiles/interface-management-profile/entry[@name='Ping from internal'] &element=<ping>yes</ping> &permntd-ip><entry name='192.168.201.0/24'/>
        </permntd-ip>"
# <request cmd="set" obj="/config/devices/entry[@name='localhost.localdomain']/network/interface/ethernet/entry[@name='ethernet1/1']" cookie="6592603819717516" newonly="yes" opaque_param="vsys1">
    name: 'internal interface (ethernet 1/1)'
    include_tasks: request.yml
    vars:
        body: "type=config &action=set &xpath=/config/devices/entry[@name='localhost.localdomain']/network/interface/ethernet/entry[@name='ethernet1/1'] &element=<layer3><ipv6><neighbor-discovery><enable>no</enable></neighbor-discovery></ipv6>
        <ndp-proxy><enabled>no</enabled></ndp-proxy><ip><entry name='192.168.201.2/24'/></ip>
        <interface-management-profile>Ping form internal</interface-management-profile></layer3>"
# <request cmd="set" obj="/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name='vsys1']/zone/entry[@name='internal']" cookie="6592603819717516" newonly="yes">
    name: 'internal zone'
    include_tasks: request.yml
    vars:
        body: "type=config &action=set &xpath=/config/devices/entry[@name='localhost.localdomain']/vsys/entry[@name='vsys1']/zone/entry[@name='internal'] &element=<network><layer3><member>ethernet1/1</member></layer3></network>"
```
Appendix I

BASH/BATS Tests for the External Host

#!/usr/bin/env bats

@test "external has correct default route" {
    vagrant ssh -c "ip route | grep default | default via 192.168.202.2" external
}

@test "external can ping external switch" {
    vagrant ssh -c "ping -c 1 192.168.202.1" external
}

@test "external can ping external firewall" {
    vagrant ssh -c "ping -c 1 192.168.202.2" external
}

@test "external can ping external host (self)" {
    vagrant ssh -c "ping -c 1 192.168.202.3" external
}

@test "external can't ping internal switch" {
    run vagrant ssh -c "ping -c 1 192.168.201.1" external
    [ $status != 0 ]
}

@test "external can't ping internal firewall" {
    run vagrant ssh -c "ping -c 1 192.168.201.2" external
    [ $status != 0 ]
}

@test "external can't ping internal host" {
    run vagrant ssh -c "ping -c 1 192.168.201.3" external
    [ $status != 0 ]
}
Appendix J

BASH/BATS Tests for the Internal Host

#!/usr/bin/env bats

@test "internal/has_correct_default_route" {
  vagrant ssh -c "ip route | grep -F 'default via 192.168.201.2'" internal
}

test "internal/can_ping_internal_switch" { 
  vagrant ssh -c "ping -c1 192.168.201.1" internal
}

test "internal/can_ping_internal_firewall" { 
  vagrant ssh -c "ping -c1 192.168.201.2" internal
}

test "internal/can_ping_internal_host/(self)" { 
  vagrant ssh -c "ping -c1 192.168.201.3" internal
}

test "internal/can't_ping_external_switch" { 
  run vagrant ssh -c "ping -c1 192.168.202.1" internal
  [ $status != 0 ]
}

test "internal/can't_ping_external_firewall" { 
  run vagrant ssh -c "ping -c1 192.168.202.2" internal
  [ $status != 0 ]
}

# demo rule test

test "internal/can_ping_external_host" { 
  vagrant ssh -c "ping -c1 192.168.202.3" internal
}