TESTBED DESIGN FOR EVALUATION OF ACTIVE CYBER DEFENSE SYSTEMS

A Thesis in
Computer Science and Engineering
by
Srikumar Sridhar

© 2018 Srikumar Sridhar

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2018
The thesis of Srikumar Sridhar was reviewed and approved* by the following:

Sencun Zhu
Associate Professor of Computer Science and Engineering
Thesis Advisor

Gang Tan
Associate Professor of Computer Science and Engineering

Chitaranjan Das
Distinguished Professor of Computer Science and Engineering
Department Head of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

As with any system, often times, an attacker only needs to know a single vulnerability to compromise the entire system. To ensure a system is free of vulnerability is extremely difficult, if not impossible, especially for large systems with over millions of lines of code. Hence, we focus on a cyber security methodology called Moving Target Defense (MTD). The philosophy of MTD is that instead of attempting to build flawless systems to prevent attacks, one may continually change the attack surface (certain system dimensions) over time in order to increase complexity and cost for attackers to probe the system and launch the attack.

The approach taken for implementing MTD in this thesis involves a naive checkpoint and restore methodology. If either an application has come under attack or the application hasn’t been switched for a while (a user-defined period), the container that it was running in will be killed and a new instance would be spawned (application switching). This would help prevent a single software vulnerability from compromising the whole system. The new instance of the application will begin its execution from the latest checkpoint available to it. We demonstrate this approach using checkpoint and restore in user space with docker containers.

Additionally, we present the design of a complete, open-source, testbed for systems that utilize docker containers. Tools such as OSSEC (Open Source Host Based Intrusion Detection System Security), Snort (Network intrusion prevention and network intrusion detection system), Bro (Network intrusion detection system), Sysdig Falco (Runtime container monitoring tool) etc., are utilized to detect intrusions or anomalous behavior in a containerized environment. Multiple intrusion detection tools are enforced in the system while various exploits are carried out. We perform application switching along with IP mutation once an intrusion has been detected and evaluate the downtime associated with this process. We find that the process of checkpointing and restoring docker containers on the same host takes roughly 2.5 seconds.
Table of Contents

List of Figures vi
List of Tables viii
Acknowledgments ix

Chapter 1
  Introduction 1

Chapter 2
  Background and Related Work 4
    2.1 System Based Defense 5
    2.1.1 Address Space Layout Randomization 5
    2.1.2 Instruction Set Randomization 6
    2.1.3 Data Randomization 6
    2.2 Network Based Defense 7
    2.3 IP Mutation Based Defense 7
    2.4 Live Migration 8
    2.5 Evaluation Framework 9

Chapter 3
  Container Technology 10
    3.1 Virtual Machines vs Containers 10
    3.2 Docker Platform 11
      3.2.1 Build and Deploy Containers 13
    3.3 Security Concerns 13

Chapter 4
  Intrusion Detection Systems and Runtime Monitoring Tools 16
    4.1 Snort 17
Chapter 5
Cyber Attacks and System Exploits

5.1 Understanding Cyber Attacks
  5.1.1 Malware Attack
  5.1.2 Denial-of-Service (DoS)
  5.1.3 Man-in-the-Middle Attacks

5.2 System Exploits

5.3 Exploit Examples
  5.3.1 Denial of Service
  5.3.2 Data Exfiltration

Chapter 6
Moving Target Defense

6.1 Testbed Design

6.2 Checkpoint/Restore In Userspace

6.3 Implementation Overview

6.4 System Configuration

6.5 CRIU Implementation

6.6 Intrusion Detection and Container Migration

6.7 IP Mutation

6.8 Application Switching

6.9 Evaluation

Chapter 7
Conclusion and Future Work

Bibliography
List of Figures

3.1 Virtual Machines vs Containers ........................................... 11
3.2 Docker Engine Component Flow ........................................... 11
3.3 Docker Architecture ......................................................... 12

4.1 Software Stack to use Snort ............................................. 18
4.2 OSSEC Server-Agent Mode ................................................. 19
4.3 Sysdig Falco Information Flow ............................................. 21

5.1 Example of a Malware Attack ............................................. 23
5.2 Code to create payload of DOS attack ................................. 26
5.3 Code to create payload of Data Exfiltration Attack ............... 27

6.1 Dockerfile for building the container image ......................... 33
6.2 Demonstration of container migration ................................. 34
6.3 Custom Falco rule that flags ping as suspicious activity ........ 34
6.4 Falco rule that flags reading sensitive files ....................... 35
6.5 Script that manages the container lifecycle based on Falco’s alerts 36
6.6 Container migration based on intrusion detection ............... 37
6.7 Log file contents after intrusion detection .................. 37
6.8 Docker-compose file for Nginx web server with static IP address configuration .............................................. 39
6.9 Starting and testing Nginx web server ....................... 39
6.10 Graph displaying various times for different container image sizes . 42
6.11 Line graph representation of checkpoint/restore times vs container image size .................................................. 42
List of Tables

5.1 Detection of various exploits ........................................ 25

6.1 Docker Configuration .................................................. 32
Acknowledgments

First of all, I would like to express my sincere gratitude to my adviser Dr. Sencun Zhu, for the patient encouragement and advice he has provided throughout my time at Pennsylvania State University. I have benefited a lot from his expertise in the area of computer security. I could not have completed my Master’s degree without his guidance and support. I want to thank my other committee member - Dr. Gang Tan as well. Without their insights and help, this thesis would not have been a successful endeavor.

I would also like to thank my parents, brother and friends who have been a constant support throughout my study and inspired me to pursue my interests.
Chapter 1
Introduction

Protecting computer systems from theft or damage of its hardware, software or electronic data is classified as system security. An increasing number of systems are at risk due to the increasing reliance on them by industries, individuals and governments as well as improper training of and usage by end users. Security breaches cause massive financial damages to the parties involved and have adverse effects across a myriad of industries worldwide.

Building secure systems imply that the software and hardware have been designed from the ground up to be secure. Multiple security techniques such as defense in depth, audit trails, principle of least privilege etc., should be implemented. A single flaw or vulnerability in an application can lead to the compromise of an entire infrastructure. Building flawless systems are complex, confusing, challenging and next to impossible. A new paradigm for approaching security has been recently proposed called Moving Target Defense (MTD). Instead of defending unchanging, spread out, distributed and untrusted systems by detecting, preventing or monitoring intrusions, moving target defense focuses on making the attack surface dynamic. A dynamic moving target attack surface ultimately reverses the asymmetric advantage of attackers. This levels the playing field between defenders and attackers. In this thesis, we will concentrate on security of applications and servers running in containers and how moving target defense can be implemented in a containerized environment.

Application container technologies, also known as containers, are a form of operating-system-level virtualization in which the kernel allows the existence of
multiple isolated user space instances. Containers have gained immense popularity since the launch of docker platform even though the technology itself dates back a long time. Containers allow users to easily package an application, and all its dependencies, into a single image. A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries and settings. Containerized software will always run the same, regardless of the environment.

Docker is an open platform that allows users to develop and run applications in an isolated environment. Docker will be used as the container runtime engine throughout this research. An instance where docker streamlines a deployment process is as follows - if someone wants to deploy a web application, they have to set up the entire stack which might include front-ends, back-end application servers, databases, in-memory caches etc., and configure how each instance connects with each other. This process is error-prone and complex. Docker simplifies this process by allowing users to specify all the services in a single file and launch all these services with a single command. This is done through docker-compose, which lets users list all the services they need along with parameters such as environment variables, volumes, dependencies etc., and streamlines the process of web app deployment.

As containers continue to evolve, the concern for security grows larger. Docker is designed to have all containers running on a host share the same kernel. If one container should cause a kernel panic, it will take down the whole host. Another security concern is that of resource manipulation. If one container manages to attain access to certain resources and manages to starve other containers, it would result in a denial of service. Understanding container security is extremely important and hence, we will build a system with different intrusion detection tools and test the effects of various exploits on it.
Moving target defense is implemented through application switching and IP mutation in this research. IP mutation is a novel technique that hides network assets from external and internal scanners by frequently changing applications’ IP addresses. Application switching is the process of checkpointing and restoring an application either on the same host or on different hosts (migration). Multiple checkpoints are created at periodic intervals while the application is running and during restore time, the latest checkpoint is used for minimum loss of work.

The systems that are built include various intrusion detection tools which alert the system administrator of suspicious or unauthorized behavior. This alert acts as a trigger to the application switching process. We evaluate the downtime associated with application switching by performing it multiple times and then report the average value. Applications are run as containers and are switched on the same host. The application memory is mapped to the host machine using volumes and hence, we can focus on just the state of the application while ignoring the application memory (for checkpoint and restore). We can infer from our evaluation that checkpoint and restore time in user space for applications takes roughly 2.5 seconds. This includes time to save the state, freeze the running application, creating a new application and restoring the state in the new application (the new application will get the application memory through old volume mappings onto the host).

In the next chapter, we will discuss previous work on moving target defense systems and container migration. Chapter 3 outlines containers and virtual machines along with a few prominent security concerns related to containers. Chapter 4 will give an introduction to various security tools used. Chapter 5 focuses on a few exploits we will be testing on the system. Chapter 6 illustrates the checkpoint and restore methodology and how we can implement moving target defense using it. Finally, we conclude the thesis with future work in chapter 7.
Chapter 2 | Background and Related Work

Traditionally, a system does not change its software stack (or solution stack), addresses, names, network and host configurations etc., dynamically. This means that the configuration parameters are relatively static and remain more or less the same for long periods. This provides adversaries time to plan and attack the system. Achieving perfect security or trying to build flawless systems are next to impossible. Hence, Moving Target Defense (MTD) relies on dynamically shifting the attack surface and make it more difficult for attackers to exploit the system. The US Department of Homeland Security (DHS) defines MTD as "Moving Target Defense (MTD) is the concept of controlling change across multiple system dimensions in order to increase uncertainty and apparent complexity for attackers, reduce their window of opportunity and increase the costs of their probing and attack efforts."[1]

Moving target defense can be approached in a variety of ways. Based on the parameter(s) that is being changed/shifted we can outline the approaches as follows -

1. Network

   Approaches that change the network configurations such as internet protocols, addresses etc.

2. Data and Platforms

   Approaches that focus on changing the syntax, format, encoding of application data or the operating system/system architecture.
3. Environment

Approaches that modify the runtime environment provided by the operating system to the application.

4. Software

Approaches that change the application’s code dynamically.

These approaches can be generalized into three categories - system based, network based and IP mutation based.

2.1 System Based Defense

2.1.1 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a defense against memory corruption attacks and is a building block for many other countermeasures. In [2], ASLR’s importance is outlined as "The goal of ASLR is to introduce randomness into addresses used by a given task. This will make a class of exploit techniques fail with a quantifiable probability and also allow their detection since failed attempts will most likely crash the attacked task". Randomizing the address-space layout of a software program prevents attackers from using the same exploit code effectively against all instances of the program containing the same flaw or vulnerability. The attacker must hence, resort to brute force attacks to guess the address-space layout or create specific exploits for each program.

ASLR albeit an effective defense mechanism has its limitations. Hovav Shacham et al. [3], discuss how brute force attacks can break the ASLR defense within minutes based on the system architecture. Just-In-Time Code Reuse [4], exploits the ability to repeatedly abuse a memory disclosure. This allows them to map an application’s memory layout on-the-fly, dynamically discover API functions and gadgets, and JIT-compile a target program using those gadgets. In [5], the authors demonstrate how insecure ASLR is in modern cache-based architectures. They describe a new EVICT+TIME cache attack on the virtual address translation performed by the memory management unit (MMU) of modern processors.
ASLR, when implemented with a detection system, will allow system administrators to effectively detect attacks and deploy countermeasures. In MTD sense, system administrators can migrate the application or change other configuration parameters based on the vulnerability that would help mitigate the attack.

2.1.2 Instruction Set Randomization

Instruction-set randomization (ISR) obfuscates the underlying system’s instructions to protect against code-injection attacks by presenting an ever-changing target. In [11], the authors propose a solution across the software stack that randomizes all binaries with different secret keys so that malicious code introduced by the attackers would fail to execute correctly. This prevents the execution of unauthorized binaries and scripts regardless of their origin.

Various implementations of the ISR technique have been published to combat code-injection by attackers. ASIST [12], is a hardware-assisted architecture for ISR support. ASIST offers better performance and improved security by protecting the operating system and resisting key guessing attempts. Gaurav et al. [13], propose an implementation that protects against any type of code injection attack. This is achieved by creating an execution environment that is unique to the running process, hence, invalidating the injected code for that execution environment. In [14] and [15], the authors describe approaches that have minimal performance overheads and utilize strong encryption to thwart state-of-art return oriented programming attacks.

2.1.3 Data Randomization

Data randomization is another randomized-based approach. By applying different random masks to data in the memory [16], it disrupts any attempts to write outside objects on the memory. Attackers cannot determine memory regions that are associated with particular objects. In [17], Cowan et al. presented an approach that randomizes stored pointer values, as opposed to the locations where objects are stored. The encryption is achieved by XORing pointer values with a random integer mask. These methods rely on attackers’ inability to guess a secret key.
2.2 Network Based Defense

At the network level, MTD is provided by using different applications, operating systems, and communication protocols [18], [19] within a networked system. In [20], Mont et al. introduced an approach to ensuring diversity for common, widespread software applications in which diversity is enforced at the installation time by a random selection and deployment of critical software components. In [21], Hiltunen et al. proposed the use of fine-grained customization and dynamic adaptation as the key enabling technologies to achieve the goal of survivable system design. In [22], ODonnell and Sethu presented several distributed algorithms for assigning different versions of software to individual systems. In [23], Yang et al. highlighted the same diversity idea and applied it in the sensor networks with limited choices of software versions. In [24], the authors proposed an algorithm to intelligently assign software to a network with the objective of minimizing the connected component that consists of hosts running the same vulnerable program. In a follow-up work [25], Huang et al. further consider the fact that the vulnerabilities have different severity levels and explore the betweenness and centrality characteristics of network topology to increase system survivability. In [26], Huang et al. proposed a multi-objective software assignment algorithm based on ant-colony optimization (ACO). The algorithm can output pareto optimal solutions for multiple objectives (e.g., maximizing survivability, usability and feasibility).

2.3 IP Mutation Based Defense

IP mutation is a novel proactive moving target defense (MTD) that hides network assets from external/internal scanners by frequently changing hosts’ IP addresses. RHM [27] is an approach that turns end-hosts into moving targets by transparently mutating their IP addresses and it employs multi-level optimized mutation techniques that maximize uncertainty in adversary scanning by using the whole available address range, while at the same time minimizing the size of routing tables and reconfiguration updates. Moreover, it has been proposed to use OpenFlow to develop an MTD architecture that transparently mutates IP addresses [28].
2.4 Live Migration

Live migration is the process of moving a running virtual machine or application between different physical machines without disconnecting the client or application. Memory, storage, and network connectivity of the virtual machine are transferred from the original guest machine to the destination [29]. Proactive maintenance is a preventive maintenance strategy for maintaining the reliability of machines or equipment. The purpose of proactive maintenance is to view machine failure and similar problems as something that can be anticipated and dealt with before they occur. This is one of the most important applications of live migration. Live migration can also be used for load balancing, in which work is shared among computers in order to optimize the utilization of available CPU resources. In [30], Clark et al. present the design, implementation and evaluation of high performance OS migration built on top of Xen VMM. Live migration is a type of MTD wherein network configurations are changed. As applications are migrated between physical machines, information such as IP address and ports that were previously exposed to an attacker become stale and useless. Live migration can be achieved in two ways - Pre-Copy and Post-Copy.

1. Pre-Copy Migration

Memory is copied to the destination before the state. The application is launched after the state is copied over to the destination.

(a) Warm-Up Phase: All the memory pages are copied from the source to the destination while the application is still running on the source. This is done by the hypervisor. If some memory pages change after they have been copied to the destination (that is, they become dirty), they will be re-copied until the rate of re-copied pages is more than the page dirtying rate.

(b) Stop and Copy Phase: Once the warm-up phase is completed, the application will be stopped on the source machine. The memory pages will be evaluated for dirty pages and all such pages will be copied to the destination. The state data will be transferred and the application will be resumed on the destination host. The time between stopping the application on the original host and resuming it on the destination is called downtime. This usually ranges from a few milliseconds to seconds and depends on the size of memory and application state.
If the destination fails during the migration process, pre-copy can recover the application. This is because an up-to-date state is maintained at the source.

2. Post-Copy

State is copied to the destination before the memory. The application is launched as soon as the state and non-pageable memory is copied to the destination.

(a) Source Application Suspend: Post-copy migration is initiated by suspending the application at the source. A minimal subset of the execution state of the application is transferred to the target. Then, the application is resumed at the destination.

(b) Pre-paging and Copy: Concurrently, the source pushes the remaining memory pages of the application to the destination.

The application’s state is distributed over both source and destination. At the destination, if the application tries to access a page that has not yet been transferred, it generates a page fault. This type of page fault is called a network fault as they are trapped at the destination and sent to the source. Too many network faults can degrade the performance of the application heavily. Post-Copy cannot recover from a failed destination during the migration process.

2.5 Evaluation Framework

Huang et al., discuss an evaluation framework based on analytic hierarchy process [49]. They select five general evaluation metrics for the moving target defense (MTD) evaluation and comparison. A generic MTD framework is proposed based on analytic hierarchy process (AHP) and is used to aggregate the evaluation metrics.
Chapter 3  Container Technology

3.1 Virtual Machines vs Containers

Containers and Virtual Machines (VM) share similar goals in terms of isolating an application and its dependencies into a self contained unit that can run anywhere. Figure 3.1 describes the architectures of virtual machines and containers. Each VM runs a unique guest operating system and multiple VMs with different operating systems can run on the same physical server. Every VM has its own binaries, libraries, and applications that it services. A hypervisor (virtual machine monitor or VMM) which is a software, firmware, or hardware creates and runs these virtual machines. Therefore, each guest OS has its own kernel and provides complete isolation from other guest OSes. This separation makes VMs heavyweight in nature - their size can be upwards of 1 GB whereas a container’s size ranges in hundreds of MB. VMs usually take longer to process and execute commands compared to containers due to the additional layer of a hypervisor. Containers, on the other hand, share the host kernel but have their own binaries and libraries.
3.2 Docker Platform

Docker is an open platform that allows users to develop and run applications in an isolated environment. Docker provides tools to manage and deploy containers on a host. Figure 3.2 shows the component flow for the docker engine. Docker uses a client-server architecture wherein the docker client sends requests to the docker daemon which in turn communicates with the server and executes the request. The docker daemon is responsible for running and distributing the containers.
Figure 3.3 shows the architecture of Docker and how clients interact with the docker daemon and public registries. Docker utilizes the docker0 bridge which is a virtual interface created by docker. Docker randomly chooses an address and subnet from the private address range that are not in use on the host machine, and assigns it to docker0. All the docker containers will be connected to the docker0 bridge by default and the docker containers connected to the docker0 bridge use the iptables NAT rules created by docker to communicate with the outside world. Users can run containers from public repositories like docker hub by pulling the image onto the local machine and then running them. The "docker pull" command pulls the specified image onto the local machine. Conversely, the "docker push" command pushes a local docker image onto the public registry (by default it is docker hub). In this thesis, we use various open-source docker images available on docker hub. A docker image is just a set of instructions that specify how to create a container. A container is hence, the runnable instance of the docker image. We can launch an interactive ubuntu container by issuing the following command:

```
docker run -it --name=ubuntu ubuntu /bin/bash
```
3.2.1 Build and Deploy Containers

Docker allows users to build images and run them in two ways - using a dockerfile or using docker-compose. A dockerfile is useful for a single application, for instance, a web server. The user needs to list what the base image should be followed by all the packages they want to be installed in the container. They can further specify what should be run by default when the container is started. By using a dockerfile, the user can pass environment variables and volume mappings through the "docker run" command. When the system needs the use of multiple applications as separate containers connected to each other for communication, docker-compose is used. This is a YAML file that allows users to specify all the services they need, how they are supposed to connect with each other, volume mappings onto the host, the base image for each container or a dockerfile etc. The user can then issue a single command that would build, create, start and link containers -

    docker-compose up

3.3 Security Concerns

1. Malicious and Vulnerable Images

   There are thousands of images available on docker hub that are being used by developers every day. Developers need to make sure that the image they are downloading are from trusted sources and run vulnerability scans on the images before running them in the host environment. An attacker might trick a user into downloading and running a malicious image that would give them control over the host. Without proper verification of a public image, the developer runs the risk of running uncontrolled code that might have vulnerabilities and produce unexpected behavior.

2. Denial of Service Attacks

   A container may open sockets repeatedly on the host and by doing so freeze the entire system. This will result in other containers being unable to execute their tasks. Another way in which denial of service of service can be achieved is
by starvation. All containers share the same resources on the host. If a container attains access to certain resources and starves other containers from these specific resources, it effectively results in a denial of service.

3. Kernel Level Threats

Since all containers share the same kernel, any vulnerability in the kernel will have a huge effect on the system. If one container causes a kernel panic, it will bring down the host and all other containers along with it. Arbitrary code execution and access to privileged ports should be restricted.

4. Container Breakout or Privilege Escalation

Containers might run as a root user and use privilege escalation to break the isolation. This, in turn, gives access to sensitive information from the host.

5. Unrestricted Access to Files/Processes

An attacker who has taken control of a container can break into the system file directory on the host through remounting. If the attacker gains root access in a container, they might gain root access on the host as well. The principle of least privilege isn’t enabled by default.

6. Data Leaks

When a container accesses a database or a service, it might need a secret, like a password or a key. If an attacker gets access to this secret, they can get access to the service as well. This is a big issue for containers as they are frequently stopped and restarted. In a few versions of docker, containers have unrestricted access on docker0 interface. This could lead to information of one container being sent to another container. Inter-container communication should be possible only if links have been established.

7. Network Attacks

A compromised container can be used to launch attacks on the docker0 interface and jeopardize all the containers running on the host. Access restrictions should be placed in order to avoid this kind of attack. In general, containers have a large attack surface due to a large number of containers running on multiple hosts interacting with each other.
8. Lack of Transparency

Containers are run by engines such as docker that interact with the Linux kernel. This results in a layer that masks the activities of specific containers and what users are executing inside containers.
Chapter 4  Intrusion Detection Systems and Runtime Monitoring Tools

Intrusion detection system (IDS) is a security technology built for detecting vulnerabilities and exploits against a computer system. They monitor the network and host system for malicious activities or policy violations. They are broadly classified into two categories - Network Intrusion Detection System (NIDS) which analyzes incoming network traffic and Host-based Intrusion Detection System (HIDS) which monitors host system files. Another common method of classification is signature-based detection system and anomaly-based detection system. When it comes to network monitoring, IDS systems are placed out-of-band. This is done to prevent any impact on network performance by the IDS. The other monitoring method is inline, that is, all packets must traverse through the system before it reaches the server. Usually, Intrusion Prevention Systems (IPS) are placed inline in a network as it must detect and stop the attack/exploit from taking place. In this thesis, we focus on using only detection tools (passive) and take steps based on it. IDS can be configured to report by raising alerts or storing the events in a log file.

Signature-based detection systems need to have every exploit, vulnerability or attack vector in some way for accurate detection. This way of defense has to play catch-up and hence is not useful against new threats or zero-day attacks. Thus, a behavioral-based security monitoring system allows users to detect new threats based on rules or policies.
4.1 Snort

Snort is an open-source network intrusion prevention system and network intrusion detection system. It performs real-time packet logging and traffic analysis through protocol analysis, content searching and matching. The software can be used in three modes - sniffer, packet logger and intrusion detection. In sniffer mode, the packets are analyzed in real-time and displayed on the machine’s terminal while in packet logger mode, they are registered as events in logs and stored in the disk. The intrusion detection mode works like a behavioral system as it matches the traffic against its rule set and performs user-defined actions.

The snort rule set is constantly updated and developers/users can use oinkcode to perform routine updates of their local rule set. To automate this process, pulled pork is used in conjunction with snort. Pulled pork automatically manages and updates the snort rule set and helps in tuning the local user defined rules.

Logs or packets collected and stored by snort cannot be accessed directly by users. This is because of the format these logs are stored in. Figure 4.1 describes the software stack and information flow to implement snort. Snort stores data in a unified binary output file format. To parse this data, we need to use another tool called Barnyard2. Barnyard2 is an open-source interpreter for snort output files. After parsing the logs, barnyard2 can store the data in a database for later retrieval or visualization. Snorby, a front-end web application is used to represent and visualize the data stored by barnyard2. The softwares (Snort, Pulled Pork, Barnyard2, Snorby) mentioned up till now form a security detection system stack that is completely open-source and commonly used for testing purposes.
There are many more open-source network intrusion detection systems available that are either as effective as or better than snort in some ways. Two popular choices in the open-source community are Bro IDS and Suricata. Bro, while being an intrusion detection system primarily, doubles down as a more general network traffic analyzer. It is used for traffic baselining, collecting network measurements etc., and is often called an all-in-one package. Suricata is another NIDS that uses rules and signature-based detection methods. It supports output formats like YAML and JSON which make it easy for data representation using splunk or elastic stack. Elastic stack includes elasticsearch, logstash and kibana, often referred to as ELK.
4.3 Open Source HIDS Security

OSSEC is a scalable, multi-platform open source host-based intrusion detection system that is capable of performing log analysis, integrity checking, rootkit detection, time-based alerting and centralized policy enforcement. It can be used to monitor a single server or multiple servers in a server/agent mode. OSSEC can be installed in different modes - server/agent or local. In local mode, OSSEC is installed where monitoring must take place and is limited to that specific system. In the server/agent mode, multiple agents are run on different systems that report to the server. This way, OSSEC can monitor multiple systems simultaneously. The OSSEC agents are connected to the server via an encrypted and secure connection. The server and agents are encrypted and authenticated using a symmetric key that is defined on the server and then exported and copied to the agent. When started, the agents connect and register to the server and send back alerts and log data in an encrypted format. Users can define rules and actions that must be performed when an action is matched against a specific rule.

![OSSEC Server-Agent Mode Diagram](image.png)

**Figure 4.2.** OSSEC Server-Agent Mode
4.4 Elastic Stack

Elastic stack is a set of softwares that accept data in any format and search, analyze and visualize data in real time. The most commonly used softwares are logstash, elasticsearch and kibana. Logstash accepts data from various sources and processes all this data for storing in a stash. Then, elasticsearch analyzes the stash for useful information and provides users with search capabilities. Kibana is the visualization software that represents the data analyzed by elasticsearch.

4.5 Wazuh

Wazuh is a security detection, visibility and compliance open-source project. It combines OSSEC with elastic stack and OpenSCAP. Security Content Automation Protocol (SCAP) provides security baselines to users. OpenSCAP is a collection of tools that enforce this standard.

Wazuh follows the modes provided by OSSEC and extends its usability by providing a means of data visualization and representation through kibana.

4.6 Sysdig Falco

Sysdig Falco is an open-source, behavioral-based monitoring tool that is used to detect suspicious activity. Sysdig Falco works as an intrusion detection system and is useful when using docker as it supports container-specific context like container.id, container.image and namespaces for its rules.

Falco allows users to define highly granular rules to check for activities involving file and network activity, process execution, inter-process communication etc. Falco alerts can be triggered through specific system calls, their arguments and by properties of the calling process. Falco runs in user space using a kernel module to intercept system calls.
Falco’s rules use Sysdig filtering expressions to identify suspicious activity and send notifications to either files, syslog or programs. In a containerized environment, falco can be launched as a container which monitors the host and all containers running on it.

Figure 4.3. Sysdig Falco Information Flow
5.1 Understanding Cyber Attacks

Cyber attack is a deliberate exploitation of computer systems and networks. Attackers use malicious code to alter computer code, logic or data, resulting in disruptive consequences that can compromise data and lead to cyber crimes, such as information and identity theft. Attackers use a variety of tools to launch attacks, including malware, ransomware, exploit kits, and other methods. A few prominent attacks are listed below.

5.1.1 Malware Attack

Malware attacks come in various forms. They use different attack vectors and can be broadly classified into viruses, worms, and trojans. The effects of these kinds of attacks can range from unauthorized information retrieval to disruption of services. Viruses are programs that can attach themselves to other software and then replicate when that software is executed. Worms are self-replicating malware that scans for vulnerable systems and exploit them. Trojans are malicious code hidden inside legitimate software to avoid detection. They provide backdoors for attackers who gain remote access to the systems that are infected. They primarily spread through web downloads and spam messages. Figure 5.1 outlines the way by which malware compromises a system. Traffic is redirected to the hacker’s exploit rootkit instead of the legitimate web server.
5.1.2 Denial-of-Service (DoS)

Denial of service (DOS) attacks are used to indefinitely disrupt services of a host connected to the internet. Typically, DOS attacks are carried out by consuming the whole network bandwidth of the target or exhausting the target system’s resources. These attacks are usually carried out through botnets. Botnets are networks of compromised machines that have been infected with malware and placed under control of the attacker. Botnets stay dormant until they are given some tasks from the attacker. With a huge botnet, the attacker can cause distributed denial of service which makes it harder to locate the source of the attack.

5.1.3 Man-in-the-Middle Attacks

Man-in-the-middle attacks are a common type of cyber security attack that allows attackers to eavesdrop on the communication between two targets. The attack takes place in between two legitimately communicating hosts, allowing the attacker to “listen” to a conversation they should normally not be able to listen to. This is usually achieved through session hijacking, SSL stripping, packet injection and sniffing.
5.2 System Exploits

An exploit is a set of instructions that takes advantage of a system flaw or vulnerability to gain unauthorized access or cause unexpected behavior. Usually, scripts or frameworks are used to attack target systems. We will use the following exploits on our system to test the efficacy of the intrusion detection tools. For most of the exploits, privilege escalation is utilized before carrying out the task. Payloads and scripts for these attacks are available on Google Hacking Database [50] and on GitHub [51].

We build a couple of testbeds to test the effectiveness of intrusion detection tools against these exploits. The first testbed includes Wazuh as the intrusion detection tool and the second testbed contains sysdig falco as the detection tool. Both intrusion detection systems utilize the default configurations and/or rules. We summarize our results in Table 5.1. A few exploits that were tested on the system are as follows -

1. Network Scanning

A worm is a standalone malware that replicates itself and infects other hosts on the network. For propagation, worms usually scan the compromised machine’s ports for any open connections and infect all such systems. We will simulate this network scanning using NMAP (Network Mapper) and check for flags in our intrusion detection systems’ logs. We can launch an nginx server using docker container and run nmap over its ports. Nginx is a web server which can be used as a proxy, load balancer or a HTTP cache.

2. Denial of Service Attacks

If a web server receives a lot of requests from multiple applications simultaneously, there is a good chance that it will freeze or crash. This type of attack falls under the category of DOS. We can simulate this attack by using NMAP to scan all 65K ports in the web server repeatedly by multiple processes.

3. Data Exfiltration

The /etc/shadow file in Linux stores the actual passwords in encrypted format. In this exploit, we will read this sensitive file and send the data via UDP to a
specific IP Address and port.

4. File Integrity Exploit

The /etc folder in Linux doesn’t provide non-root users write access. In this exploit, we will create an empty file and save it in the /etc directory.

5. Unauthorized Command or Anomalous Behavior

Once a web server has been deployed (for instance an nginx container), we should restrict access to the shell of the web server by other processes. If another process gains shell access, it can read sensitive information and break the isolation. For this exploit, we will utilize docker exec command to launch a bash for the nginx container and check the response of detection tools.

6. Privilege Escalation

This is one of the most basic exploits that can be used to leverage the whole system’s integrity. For this exploit, we will make use of setuid (set user ID) to allow users to run an executable with permissions of the executable’s owner and modify behavior in directories.

7. Malicious Code

In this exploit, we will gain access to a database server and execute another program. In this instance, we utilize the respawn service to launch an "ls" command.

<table>
<thead>
<tr>
<th>Exploit</th>
<th>Detection</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Wazuh</td>
<td>Sysdig Falco</td>
<td></td>
</tr>
<tr>
<td>Network Scanning</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Denial of Service Attack</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Data Exfiltration</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>File Integrity Exploit</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Unauthorized Command or Anomalous Behavior</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Privilege Escalation</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Malicious Code</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1. Detection of various exploits
5.3 Exploit Examples

5.3.1 Denial of Service

We utilize a payload that attempts to connect to a port on the target system (in this case 172.17.0.7) and close it immediately. In the case that the socket could not be connected, the attack fails. Although the attempt to connect to the socket fails, it did register a network imprint on the target. This process of trying to connect to the socket is done repeatedly by multiple containers concurrently. Effectively, this results in network congestion for the target and ultimately result in DOS attack. Figure 5.2 is a code snippet that is utilized for generating a payload that will cause a denial of service against the target system.

```c
void network_activity() {
    printf("Connecting a udp socket to 172.17.0.7:8197...\n");
    int rc;
    int sock = socket(PF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in localhost;
    localhost.sin_family = AF_INET;
    localhost.sin_port = htons(8197);
    inet_aton("172.17.0.7", &localhost.sin_addr);

    if((rc = connect(sock, (struct sockaddr *)&localhost, sizeof(localhost))) != 0)
    {
        fprintf(stderr, "Could not bind listening socket to localhost: %s\n", strerror(errno));
        return;
    }

    close(sock);
}
```

**Figure 5.2.** Code to create payload of DOS attack
5.3.2 Data Exfiltration

We utilize a payload that reads the contents of `/etc/shadow` file and sends the data over to the attacker (172.17.0.7). To achieve this, privilege escalation is done before this payload is executed. The attack is executed from inside the docker0 interface. Figure 5.3 is a code snippet that is utilized to generate a payload that will read sensitive files and send data over a socket.

```c
void exfiltration()
{
    ifstream shadow;
    if (shadow.open("/etc/shadow")
        printf("Reading /etc/shadow and sending to 172.17.0.7:8197...
        if (!shadow.is_open())
            printf(stderr, "Could not open /etc/shadow for reading: \n", strerror(errno));
            return;
        string line;
        string shadow_contents;
        while (getline(shadow, line))
            shadow_contents += line;
            shadow_contents += "\n";
        int rc;
        int sent;
        struct sockaddr_in dest;
        if (connect(sockfd, (struct sockaddr *)&dest, sizeof(dest)) == 0)
            printf(stderr, "Could not bind listening socket to dest: \n", strerror(errno));
            return;
        if ((sent = send(sockfd, shadow_contents.c_str(), shadow_contents.size(), 0)) != shadow_contents.size())
            printf(stderr, "Could not send shadow contents via udp datagram: \n", strerror(errno));
            return;
        close(sockfd);
    }
}
```

**Figure 5.3.** Code to create payload of Data Exfiltration Attack
Chapter 6
Moving Target Defense

6.1 Testbed Design

A testbed is a platform for conducting rigorous, transparent, and replicable testing of scientific theories, computational tools, and new technologies. In this thesis, we define our environment design (software stack) as part of the testbed. We typically have multiple containers running applications with specific roles. An example of such an environment would be as follows -

- **Host** Ubuntu 18.04 operating system with Docker Community Edition
- **Web Server** A container running Nginx web server
- **Runtime Monitoring** A container running Falco that logs all activities.
- **HIDS** A container running Wazuh/OSSEC that detects host based attacks.
- **NIDS** A container running Snort/Bro/Suricata to detect network intrusions.

We define moving target defense as follows: Multiple software (e.g., different operating systems, different types of browsers, different versions of the same software) installed in computers in a networked environment reshuffle over time so that a single software vulnerability will not result in all computers being compromised. To achieve such a system, we take a naive approach of killing the compromised container and launching a new instance of the application.
6.2 Checkpoint/Restore In Userspace

Checkpoint/Restore In Userspace (CRIU) is a tool for Linux operating system that allows users to freeze a running application and checkpoint it as a set of files on disk. These files can later be used to restore the application to the exact same state as when the checkpoint was created.

First, we need to enable experimental mode in docker. Docker wants to manage the full lifecycle of processes running inside its containers, so CRIU should be run by Docker (rather than separately) and can be done only in experimental mode. To checkpoint a running container, we have a top level checkpoint sub-command. Docker lets its users create, delete and manage checkpoints through "docker checkpoint <options> <container-id>". By default, docker manages these checkpoints and stores them in its temporary directory. A custom storage path can be specified so that these checkpoints are persistent. To utilize this feature, docker should be built with seccomp and the kernel must be configured with CONFIG_SECCOMP enabled. To create a checkpoint for an nginx container we can issue the following command -

```
docker checkpoint create nginx checkpoint1
```

By default, once a checkpoint is created, the container is stopped. To avoid this, we must specify the --leave-running flag for the checkpoint command. To restore a container from a checkpoint we need to pass the checkpoint as a flag to the docker start command. This can be done as follows -

```
docker start --checkpoint checkpoint1 nginx
```

Since the above commands restarted the same container from a previous checkpoint, we didn't need to mention the checkpoint ID. In our case, we spawn a new instance of the application (a new container) and hence will have to specify the fully qualified path of the checkpoint instead of just the checkpoint name.
CRIU checkpoints only the running state and metadata of a container. If we have persistent data associated with our container (or application) that needs to be transferred to the new container, we need to make use of volumes. When we create containers we should map data locations onto the local host. This way of mapping volumes onto the host allows us to create multiple containers that access the same data. Volumes are easy to back up and migrate, and can be managed through docker CLI commands. New volumes can have their content pre-populated by a container as well.

6.3 Implementation Overview

To implement moving target defense in a containerized environment, we would need the following prerequisites -

1. An application running in a container that will be compromised through an exploit.
2. Some sort of intrusion detection system or container monitoring tool that stores events in a log file that is mapped onto the host machine.
3. A bash script that continuously probes log files for any events and creates checkpoints for the application.
4. An exploit that can be used on the application to trigger an intrusion alert.

One such instance for implementation purposes is as follows -

1. Launch an nginx web server as a docker container.
2. Launch sysdig falco as a container and map its log files onto the host. Add user defined rules that will definitely flag the exploit as suspicious behavior and raise an alert. In the rule, mention how the event should be stored. Ideally, we should pass the container ID to the log file.
3. Execute the script that continuously probes the log file for any events. This script will not stop probing until the user specifically kills it. If there are no events in the log file, create a checkpoint of the application and store it in a specific
path (the container is not stopped but left running). If the log file has an event listed, extract the respective container ID. Kill the container using docker stop <container_ID> and check for any available checkpoints. If there is no checkpoint, create a new instance and let it begin its execution from scratch. Conversely, create a new instance of the application and attach the latest checkpoint to it. If there were volume mappings in the original container, then add those volume mappings to the new container as well.

4. The probe time and checkpoint time can be different. An application can be checkpointed every 3 hours while the probe time can be 1 second. The probe time should be low so that attacks don’t go unnoticed for a long period of time.

5. Use Google Hacking Database and deploy an exploit that modifies sensitive files in the container.

6. Check if the web server is still running (albeit it would be a new instance).

We can repeat the above process for various intrusion detection systems and exploits. If the IDS successfully flags an event/process, the bash script will relaunch the container with a slightly older state. We can use docker stats and docker inspect for gathering useful information on our container’s resource usage. Docker stats prints out all the running containers’ CPU usage, memory usage, network I/O etc. Docker inspect provides users with low-level information on docker objects. It is used to get detailed constructs controlled by docker.
6.4 System Configuration

1. Operating System: Ubuntu 16.04 LTS
2. Memory: 8 GB
3. Processor: Intel Core i7 Haswell 1.70 GHz x 4
4. CRIU version: 2.6.1
5. Docker: Refer to Table 6.1

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>17.06.2-ce</td>
<td>17.06.2-ce</td>
</tr>
<tr>
<td>API Version</td>
<td>1.30</td>
<td>1.30 (minimum version 1.12)</td>
</tr>
<tr>
<td>Go Version</td>
<td>go1.8.3</td>
<td>go1.8.3</td>
</tr>
<tr>
<td>Git Commit</td>
<td>cec0b72</td>
<td>cec0b72</td>
</tr>
<tr>
<td>Built</td>
<td>Tue Sep 5 19:58:50 2017</td>
<td>Tue Sep 5 19:57:44 2017</td>
</tr>
<tr>
<td>OS/Arch</td>
<td>linux/amd64</td>
<td>linux/amd64</td>
</tr>
</tbody>
</table>

|                |                           |                           |
| Experimental   | True                      |                           |

Table 6.1. Docker Configuration
6.5 CRIU Implementation

For showcasing the working of CRIU, we build a container using Ubuntu as the base image and start a counter when the container is started. This counter is increased by 1 every second. We use a Dockerfile for building our container (Figure 6.1). The Dockerfile mentions the base image to be used (FROM), the maintainer (MAINTAINER), packages to be installed (RUN) and what the container should do when launched (ENTRYPOINT). Figure 6.1 shows contents of a Dockerfile that uses Ubuntu as the base image and installs a few packages. It instructs the container to start a counter when the container is started.

![Figure 6.1. Dockerfile for building the container image](image)

A container counter is created and started. The counter begins at 0 and starts incrementing. We checkpoint the container which stops the container as well (we can leave it running as well but will use that feature later in this chapter). A new container with the same image is launched with the checkpoint. The counter begins updating from previously cut-off value instead of 0 (counter’s state was saved). In Figure 6.2, the whole application switching process is shown. The counter starts incrementing from 5 after switching.
6.6 Intrusion Detection and Container Migration

Now, we will discuss how to implement a system with an intrusion detection tool that will flag an attack and lead to container migration based on checkpointing method. We will use the same counter example for this demonstration. For the intrusion detection tool, we use Sysdig Falco. The container is checkpointed periodically and when falco detects an intrusion, the latest available checkpoint is used for restoring the state of the container/application. We disable the default rules of falco and use custom rules or a subset of default rules for attack detection. We can define our own rules by specifying the rule name (name of the rule), description (a basic description of the rule), condition (when should falco flag the activity), output (what should be the output stored in the falco log file/printed to terminal) and the priority of the rule. Figure 6.3 describes a simple falco rule that flags all ping activity from containers as suspicious behavior.

- **list**: banned
  - **items**: [ping, ping6]

- **rule**: network_monitor
  - **desc**: Any ping command issued by a container will get flagged
  - **condition**: proc.name in (banned)
  - **output**: "A ping command has been issued by container=%container.id"
  - **priority**: WARNING

**Figure 6.2.** Demonstration of container migration

**Figure 6.3.** Custom Falco rule that flags ping as suspicious activity
For our implementation, we will utilize an exploit that will try reading the /etc/shadow file (refer to 5.3.2). To this effect, we need a falco rule that covers such an exploit. Figure 6.4 displays how the rule can be created so that falco flags such an activity and reports it to the system administrator.

```bash
- list: sensitive_file_names
  items: [/etc/shadow, /etc/sudoers, /etc/pam.conf]

- macro: sensitive_files
  condition: >
    fd.name startswith /etc and
    (fd.name in (sensitive_file_names)
     or fd.directory in (/etc/sudoers.d, /etc/pam.d))

- rule: Read sensitive file untrusted
  desc: >
    an attempt to read any sensitive file (e.g. files containing user/password/authentication information). Exceptions are made for known trusted programs.
  condition: >
    sensitive_files and open_read
    and proc.name exists
    and not proc.name in (user_mgmt_binaries, user_exec_binaries, package_mgmt_binaries,
      cron_binaries, read_sensitive_file_binaries, shell_binaries, hids_binaries,
      vpn_binaries, mail_config_binaries, nomachine_binaries, sshkit_script_binaries,
      in.proftpd, mandb, salt-minion, postgres_mgmt_binaries)
    and not cmp_cp_by_passwd
    and not ancible_running_python
    and not proc.cmdline contains /usr/bin/mandb
    and not run_by_qualys
    and not run_by_chef
    and not user read sensitive_file_conditions
    and not perl_running_plesk
    and not perl_running_up TMP
    and not veritas driver script
  output: >
    Sensitive file opened for reading by non-trusted program - container=%container.id
  priority: WARNING
```

**Figure 6.4.** Falco rule that flags reading sensitive files
Once we have our intrusion detection system set-up, we will start a script that keeps probing the log file of falco. If it finds that the log file is empty (that is, there have been no activities flagged as suspicious by falco), the container is checkpointed. Conversely, the compromised container is killed and a new container with the same image is launched with the latest available checkpoint (if any). Figure 6.5 is a code snippet of the script we use for this purpose.

Figure 6.5. Script that manages the container lifecycle based on Falco’s alerts
Figure 6.6 shows the lifecycle of the application (counter). It is first run in the container called mycounter and after an alert from falco, the application is migrated to another container. There is loss of work associated with this form of migration. This loss of work is proportional to the probe time. The probe time is the time between each check of the falco log file. Figure 6.7 shows the contents of the log file after an intrusion has been detected. In this case, we have set the probe time to 1 second. Hence, we see duplicate work (counter value starts at 5 in the second container) after migration.

Figure 6.6. Container migration based on intrusion detection

Figure 6.7. Log file contents after intrusion detection
6.7 IP Mutation

So far we have discussed checkpoint/restore in userspace and how to link intrusion detection with container migration. To achieve MTD though, we need to change some configuration in the new instance of the container. For our implementation, we will be changing the IP address of the container. Docker allows users to set a static IP address to a container during launch time. This can be done if the user has created a virtual network and assign IP addresses in that range for the container. Hence, we modify the docker run command to include the IP address parameter. An example of a container with a static IP address is as follows -

```bash
docker network create --subnet=172.18.0.0/24 vnet

docker run -d --name nginx_server --net vnet --ip="172.18.0.23" nginx
```

The nginx server is thus launched with the IP address we mention in the run command. We maintain a list of IP addresses in our virtual network that are available for allocation and randomly select one for assignment to the new container. Implementing this approach is easy for our counter example. We just need to modify our script to include the IP address allocation. The problem arises when we have a client server architecture where the server is migrated with a new IP address. For this purpose, we use a proxy server that connects to the client and forwards all traffic to the web server. At a given point in time, there can be only one web server running. We maintain a file that lists the web server’s IP address. This file is updated with the new IP address upon container migration. The proxy server does the following when it receives an incoming request -

1. Checks if container associated with current IP address is still running. (During launch time the current IP address is set to the nginx web server’s IP address)
2. If the container is running, it forwards the request to the web server.
3. Otherwise, it checks the file for the new IP address.
4. This is set as the current IP address and the proxy server starts forwarding all incoming traffic to this address.
We have nginx-proxy servers available on docker hub that perform these actions with detailed configuration. We can use docker compose to create a virtual network and launch a container with a static IP address within that network. Figures 6.8 and 6.9 are code snippets that show how to create a virtual network, assign a static IP address and launch the containers.

Figure 6.8. Docker-compose file for Nginx web server with static IP address configuration

Figure 6.9. Starting and testing Nginx web server
6.8 Application Switching

Handling of address space of processes during application switching and/or migration with Address Space Layout Randomization turned on is a bit tricky. This is because the entire address space of the application, i.e., code, data, stack and the extended segment are randomized. CRIU handles the address space during container migration by using the existing and extending interfaces to the Linux kernel and collect as much information as possible about the process [43] [44] [45] [46] [47]. It uses the PTRACE interface to take control of the process and freezes the process. CRIU then injects a parasite code into the process’s address space using PTRACE and runs it to access and dump/save/checkpoint the memory content of the process. During the restore process, CRIU transforms the CRIU restore into the process that needs to be restored. It manipulates the process ID by modifying the /proc/sys/kernel/ns_last_pid.

Evans et al. discuss how PaX [2] randomizes the base addresses for the executable area containing the program’s code and static data structures, the stack area containing the execution stack, and the mapped memory area containing the heap as well as shared memory and dynamically-loaded libraries [48]. A randomly generated offset is added to the address of each of these areas for randomization. The layout is unchanged within these areas. Hence, this can be implemented by the loader without any changes needed to the executable. There are other implementations which randomize the address layout as well. In [31], Subba Rao et al. discuss how the address space is handled during migration. All the addresses associated with the process are virtual and are translated to physical addresses through page directories and tables. Helper functions are used to transfer data between the kernel space and user space. Smith [32] describes common approaches taken towards process migration.
6.9 Evaluation

From our implementation, we can say with certainty that proactive checkpointing will result in some loss of work. This loss of work depends on how we approach the checkpoint and restore method. In our case, this loss of work is directly proportional to the probe time. When we set our probe time to 1 second, our loss of work roughly equals 1 second. Apart from this, we have downtime associated with container migration.

To evaluate this downtime, we utilize nginx images of different sizes. For each image, we conduct our tests 50 times. We do not copy application data while migrating the container. We assume that the memory which is mapped to the host is available to the new instance of the container as well (through the same volume mappings). Hence, using CRIU, we copy the state of the container and later restore using this state information. We calculate four parameters -

1. Checkpoint and stop time: This is the time taken to checkpoint a container and stop it.
2. Checkpoint time: This is the time taken to just checkpoint a container and leave it running.
3. Stop time: This is the time taken to stop a running container.
4. Restore time: This is the time taken to restore a container.

We can infer from our tests that checkpoint and restore times are not impacted drastically by the container image size. This is because we migrate a container on the same host where we already have all the application memory stored. Thus, the copy time is dominated by just the state information which is very small in size (usually in hundreds of KB). Figure 6.10 shows the time taken for various operations for images with different sizes. Figure 6.11 is another representation of the same data that shows the trend line for each operation across various image sizes.

On an average, the total downtime (time from checkpointing to restoring a container) is roughly 1.5 seconds and the total loss of work is close to 1 second. Therefore, our effective time lost is 2.5 seconds for application switching.
Figure 6.10. Graph displaying various times for different container image sizes

Figure 6.11. Line graph representation of checkpoint/restore times vs container image size
Moving target defense (MTD) is not a technique that should be used to replace traditional passive defense systems. Rather, MTD should be used in conjunction with passive defense systems to leverage the best of both defenses. When MTD is used to change just the network configuration, the attacker can still utilize the same exploit and attempt to bring down the new instance of the application. To prevent such a scenario, passive defense systems should be deployed that would block traffic based on the type of attack that was detected by MTD. This way, we can mitigate the exploit, have low downtime and prevent future exploits of the same nature.

In this thesis, we have discussed different ways to implement intrusion detection tools and container migration method using CRIU. Furthermore, we built testbeds to test the effectiveness of these tools against various exploits and showcase a proof of concept for MTD. Thus, we can infer from our study that MTD is a promising defense system that can help prevent massive downtimes of servers and time-sensitive systems. In our simple scenario where containers are migrated on the same host and application memory is linked through volume mappings on the host, we faced effective downtime of roughly 2.5 seconds.
For our future work, we will focus on building more complex systems with varied exploits that are executed simultaneously. Testbeds that are designed will be tested rigorously for multiple possible system vulnerabilities. Additionally, we would want to work towards achieving MTD in other ways apart from dynamic network. Once these tasks are completed, we would want to migrate containers between different hosts with minimal downtime and look towards optimizing the checkpoint/restore method. If application memory needs to be copied (in the case of different hosts), a trade-off needs to be established between pre-copy migration and post-copy migration.

Another goal is to implement mobile agents based active defense. This system will simulate ants’ foraging behavior and traverse among a set of computers in a local area network to help identify possible computer compromises in an early stage. Specifically, the likeliness of a machine being compromised is modeled at the pheromone level, which is strengthened when additional suspicious behaviors are observed in the machine and otherwise gets decayed for the lack of supporting evidence.
Bibliography


