CANARY: AN AUTOMATED APPROACH TO SECURITY SCANNING AND REMEDIATION

A Thesis submitted in partial fulfillment
of the requirements for the degree
Master of Science

Department of Computer Science
Western Kentucky University
Bowling Green, Kentucky

By
David Wiles

May 2022

CANARY: AN AUTOMATED APPROACH TO SECURITY SCANNING AND REMEDIATION

_____
Defense Date

_____
Committee Chair

_____
Committee Member

_____
Committee Member

_____
Associate Provost for Research & Graduate Education

To my mother, father, and sister.

&

To the faculty of the Computer Science Department at Western Kentucky University.

ACKNOWLEDGMENTS

My appreciation goes to those who made this thesis possible.

To Dr. Guangming Xing, my thesis advisor. Without your direction and encouragement this thesis would not have been possible. Thank you for all your support from my time as an undergraduate and as a graduate. If not for you, I would not be where I am today.

To Dr. Michael Galloway. Your teaching and leadership through the years has been critical to my development as a software engineer. Thank you for your guidance, insight, and help.

To Dr. Qi Li. For patiently teaching and helping me grow. The knowledge I gained from you has been helpful to this day and it is due to your persistent explanation of these topics that I am able to apply it.

And also to God, my friends, and my family.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CANARY: AN AUTOMATED APPROACH TO SECURITY SCANNING AND REMEDIATION

David Wiles May 2022 74 Pages

Directed by: Dr. Guangming Xing, Dr. Michael Galloway, Dr. Qi Li

Department of Computer Science Western Kentucky University

Modern software has a smaller attack surface today than in the past. Memory-safe languages, container runtimes, virtual machines, and a mature web stack all contribute to the relative safety of the web and software in general compared to years ago. Despite this, we still see high-profile bugs, hacks, and outages which affect major companies and widely-used technologies. The extensive work that has gone into hardening virtualization, containerization, and commonly used applications such as Nginx still depends on the end-user to configure correctly to prevent a compromised machine.

In this paper, I introduce a tool, which I call Canary, which can detect configuration errors as well as fix them or at least alert the user of their presence. Canary could be used when setting up a new machine or application as well as in a regular security audit on an existing system. This program could be used to evaluate whether a system is following security best practices and help the administrator resolve simple issues without spending time investigating them. It could also be used in conjunction with new automation tools to create a security step when provisioning cloud infrastructure.

1INTRODUCTION

1.1 Current Cloud Computing Landscape

1.1.1 Public Cloud and the Linux Kernel

A glance at the offerings provided by Amazon Web Services shows that web hosting in 2022 is heavily based on virtualization. Virtual machines are cheaper for the provider and customer, more efficient, and easier to work with. Entire fleets of cloud services can be created programmatically using tools such as Terraform or Chef. Additionally, after the introduction of Docker in 2013 containers have quickly become the preferred way to test and deploy code due to their ease of use. Since there may be software from many different customers running on a single CPU concurrently, we may assume that this would be a major security issue.

Thanks to the work of many talented developers working on KVM, LXC, and Docker, we can assume that there are a limited number of critical vulnerabilities in the actual virtualization layer. The majority of issues arise from the misconfiguration of the host machine or the virtualization itself. This may include overly-permissive rules for programs or users, exposed secrets, a lack of resource management, and other issues which will be discussed in this paper. Although by default the attack surface for these services is very large, we can use tools to detect issues and automatically fix them to ensure that host systems are secure from attacks from within the virtual machines.

1.1.2 Complicance and Malware Scanning

Compliance scanners exist which detect configuration issues as defined by a standard compiled by an organization defining standards for system configurations. These standards may be required for certain organizations or governments to ensure security, but can also provide a

good starting point for a business or organization which does not have this compliance regulated. OpenShift, a platform provided by Red Hat, does include a compliance scanner with the service. This scanner will determine whether the system configuration meets the standards. Similarly, Docker, Inc. maintains a project on GitHub that will automatically scan a Docker setup for any issues [Docker, Inc., 2022].

These scanners can be combined with dynamic binary analyzers to determine if a program has been infected at runtime, such as through a specially crafted input to a library such as log4j. Choi, Park, Eom, and Chung detail a system that could be used to determine the presence of malware at runtime [Choi, Park, Eom, and Chung, 2014]. Despite these protections, a system should always be built to withstand any attacks as much as possible through careful analysis of the system's configuration.

1.2 Common Attacks

In general, an attacker may wish to achieve one of three goals when compromising a system. These goals are gaining control of the system, reading sensitive data, or denying access to the system. Proper configuration of the system is critical to preventing the attacker from achieving these goals, even if they can exploit one part of the system.

1.2.1 InfoLeak

A related issue to improper password storage is improperly secured sensitive data. Any data generated or stored by a server should be considered sensitive since an attacker could infer a considerable amount from log files or other seemingly innocuous files. A careless setting on these log files could expose customer information or company secrets to an attacker, even if the attacker doesn't have root access to the machine.

Additionally, hardware issues such as timing attacks can also lead to an attacker gaining

knowledge of a system. These attacks are generally harder to achieve since the attack would need to involve multiple components, but can still be mitigated by configuring the resources available to each VM or container

1.2.2 Denial of Service

Denial of service is likely the most common attack on cloud systems today. This may come in the form of ransomware, general degradation of service, or a distributed denial-of-service attack. The most well-known denial-of-service attack comes in the form of a network attack, where a server is flooded with requests to the point that service is degraded or unavailable for actual users. While this kind of attack can be mitigated through a variety of methods, they aren't relevant for this paper and so will not be discussed.

Other forms of denial of service attacks can come from malicious guests on a host. Consider a public cloud environment that allows users full access to a guest VM. The malicious actor could attempt to gain control of many VMs on a single machine and overuse the host's resources to degrade the performance of the other guests, especially if another guest is the target. This technique is also used in side-channel attacks, mentioned previously under InfoLeak.

1.2.3 Privilege Escalation

Privilege escalation is the act of gaining more permissions than intended on a system. Gaining control of a system is one of the most critical issues for a server machine since it could lead to any number of attacks. A malicious actor could gain control in several, but the most well-known is through remote code execution. A bug in a program, such as the recently discovered log4j vulnerability, can allow the attacker to execute an arbitrary code sequence through a carefully-crafted

input [M. Cooney, 2021]. Once the attacker has this capability, they could easily start a reverse shell with root permissions and steal information or demand a ransom to unencrypt data.

2CANARY: A CLI FOR SECURITY

Canary is a generic security scanner and resolver meant to be extendable through the creation of independent packages.

2.1 Canary Background

2.1.1 Purpose

My goal in creating this software was to see if a system could be examined for common vulnerabilities programmatically. Although there are many books written about setting up a system securely and many VM images can be found with best practices already configured there is not a well-known all-purpose program that could be used to securely set up a system. Canary was created in an attempt to demonstrate that this void could be filled relatively simply using a combination of a command-line program and remote repository.

2.1.2 Inspiration

The initial inspiration for Canary came from the package manager for node, 'npm'. Whenever the user installs new software, npm will automatically scan the packages for vulnerabilities and generate a report for the user. The software will also automatically upgrade the affected packages if possible to resolve these issues. A similar program for Linux machines could scan a system for misconfiguration and report to the user issues which were found. If these issues could be easily fixed programmatically then the user could run a script to fix the issues. Although this does not guarantee the safety or even that the system would work, it does provide a good starting point for an engineer who is setting up a system.

Table 2.1: Description of Canary commands

Command	Description
Install	Installs packages from a remote repository
Upgrade	Upgrades packages to their latest version
Remove	Removes packages from local storage
Check	Uses packages to check for issues and resolve them

2.2 Canary Implementation

Canary is a CLI but a related remote repository is crucial for the program's viability as a security solution.

2.2.1 Canary Usage

There are four commands included with this first version of Canary.

With each command the option '–local' can be passed to specify the local storage for packages, otherwise the directory '~/.canary' will be used. The remote repository can also be specified on the command line if the default is not desired.

On the check command, it is possible to specify a package not from the remote repository by using the flag –directory to specify the root directory of the custom package. This command also supports '–auto-fix' to fix issues without prompting the user and '–scan-only' to report issues without prompting the user to fix them.

2.2.2 Design

Canary is a command-line program written in Scala. It contains a custom argument parser which then initializes specific commands based on the user input. The only runtime dependencies of the project are Softwaremill's STTP for an HTTP client and the Compress library Apache Commons to extract gzipped archives downloaded from the remote Canary repository.

The install command checks the remote repository for the specified packages and downloads them into a local directory if they exist. It then verifies the checksum from the remote repository and extracts the archive into the local folder so the package can be used.

Upgrade performs a similar action to install, but only acts on packages that already exist locally. This is to prevent unintentional actions by the user.

Remove will check that a package exists and remove all files related to it in the local Canary root directory.

The check command performs the actual software scanning and issue resolution. The check command will use the specified packages and run each 'analyze' file to determine if there are security issues by checking the exit status of the program. If the exit status was not 0, then Canary will prompt the user and run any provided solution file to resolve the issues identified.

2.2.3 Canary Packages

A Canary package is a directory containing tasks. These tasks are also directories, which contain shell scripts and a configuration file, which should be called 'config'. This file will specify the name of any other scripts to run and include a description of the issue.

2.2.4 The Canary Repository

Crucial to the viability of Canary as a way to automatically scan issues is a remote repository to maintain packages. Successful software today is dependent on a seamless way to distribute and extend itself, such as with npm. Downloading and executing software is always risky, so users can be sure that the packages they download are trusted and have been verified to reduce this risk.

This also makes it more simple for third parties to add security checks for their own programs. An organization that wants to release a package on Canary could submit their package to be stored on the Canary repository instead of trying to distribute their own package and security scan-

Table 3.1: Overview of scanning approaches

Action	Can Be Automated	Needs Only Unix Tools
Software CLI interaction	Yes	Yes
Configuration File Search	Yes	Depends on Software
XML/JSON/YAML File Parse	Yes	No
Unix Tools Action	Yes	Yes

Table 3.2: Overview of redemdiation approaches

Action	Can Be Automated	Needs Only Unix Tools
Software CLI Action	Yes	No
Configuration File Find and Replace	Yes	Yes
Manual Remediation	No	No
XML/JSON/YAML File Update	Yes	No

ner. Approaching security this way allows the project maintainers to update security best practices as soon as they change and immediately distribute the changes to all users of Canary.

3SECURITY SCANNING AND REMEDIATION PATTERNS

The tasks created to scan and fix security issues in software can be reduced to a number of similar patterns. Actions used to scan for software issues can be reduced to the following groups:

Likewise, issue remediation can be reduced into the following groups:

Note that to fix a specific issue some automated tasks may break the system so a manual action would be preferable.

3.1 Scanning Techniques

Software configuration scanning techniques will be defined in the following section along with broad examples of how to implement the technique.

3.1.1 Interaction with a Software's CLI

Some software will be primarily interacted with through a command-line interface. An example of this is Docker. Although all actions can be performed through an HTTP interface to the Docker daemon, the most common and well-documented way to interact is with the Docker command-line client. This client is maintained by Docker itself and is intended to be the primary way that the user manages Docker containers and the Docker daemon.

Software such as this will provide commands that can be used to determine how the software is set up. There will likely be ways to view configuration and modify configuration from the command line. With Docker, it is possible to inspect a container to view all the information related to it, for example. This output can then be checked to ensure that certain policies are in place.

3.1.2 Configuration File Parsing

One common way to configure software in Linux is through configuration files. These files can define the behavior of a program and so parsing these files can help us determine what that behavior will be and whether it matches our requirements.

These configuration files could be as simple as a key-value store as seen in the SSH configuration file. It is relatively simple to use Unix Tools alone to determine whether our ssh configuration meets our requirements since a Regex search would reliably determine whether a certain parameter is enabled. For example, if we want to disable password authentication, we would search for a line such as

```
1 PasswordAuthentication yes
```

This can be done with a regular expression.

Other software may use a DSL to define the configuration. These programs may have

behavior too complex to define with a file as simple as a key-value store and would require extra syntax to concisely define its behavior. The webserver Nginx is a good example of software with a DSL used in configuration files. The .conf files may be easy for the user to read and understand, but a computer would be unable to track the context of the file without using special software to parse the file. This makes scanning configuration files with a DSL more similar to parsing XML files since it is often not possible to reliably determine whether a configuration is correct using only Unix Tools.

3.1.3 XML/JSON/YAML File Parsing

XML, JSON, and YAML are popular markup languages. Due to their widespread use and simple syntax to understand, many software projects may choose to adopt one of them as their configuration file's syntax instead of creating a DSL. This standardization would make it much easier for other programs to interpret the files since parsers would often be included in the language's standard library.

Despite this, scanning a configuration file that uses a markup language would not be possible without help from an external tool. This tool could be generic, like jq, or domain-specific for the particular software. Either way, the file would need to be parsed and checked against a list of requirements to determine if the setup meets standards.

3.1.4 Actions With Unix Tools Only

Setting up a new program also involves configuring the system it will run on. There must be a user to run the process and that process may require AppArmor policies to ensure that the process can't damage other parts of the system. Scanning these parts of the Linux kernel would only require built-in tools.

3.2 Remediation Techniques

Software configuration remediation techniques will be defined in the following section along with broad examples of how to implement the technique.

3.2.1 Interaction with a Software's CLI

Many software projects will provide a CLI which allows for direct modification of its behavior. If we have identified a specific issue we may be able to use this CLI to modify the program's behavior without needing to restart the program. This CLI update would be a relatively simple process since we would only need to determine which specific arguments to use with the CLI.

This is especially true for daemon processes which are used to start other processes, like the Docker daemon. We may have specific requirements for the behavior of the network used by Docker containers. The Docker CLI provides commands to create and manage the networks assigned to specific containers, so we would be able to change this aspect of the program's behavior without stopping and restarting the whole Docker daemon.

3.2.2 Configuration File Find and Replace

Any software that uses simple configuration files is easy target for configuration file remediation. After scanning the file to see if a requirement is met, we will likely be able to use a program such as sed to find the offending configuration and modify it.

The steps involved to remediate this would be

- Modify the configuration issue with sed, e.g. 'sed -i 's/foo/bar/' /var/?/conf
- Restart the program's process

3.2.3 XML/JSON/YAML File Modification

Programs that use a DSL or markup language to define their behavior could also be fixed with a find and replace action, but the specific programs used would need to include a specific program to parse and modify the markup.

The steps involved with this type of configuration file would include

- Read and parse configuration file
- Modify offending attribute
- Update configuration file
- Restart the program

Notice that this is similar to a basic configuration file find and replace, except the 'find and replace' action would need to be done within a parser for the DSL.

3.2.4 Manual Remediation

Some issues are too complex or specific to the system to be resolved through automated means. These issues will require an engineer to analyze the system to determine the best course of action.

4SOFTWARE EXAMPLE: NGINX

Nginx is a popular open-source web server. The default settings when installed through a package manager such as apt are good but can still be improved on by following guidelines set by the Center for Internet Security [The Center for Internet Security, 2019]. Canary can be used to scan Nginx and similar programs for issues and resolve them using the methods described in Chapter 3.

Table 4.1: Scanning and remediation for Nginx issues

Issue	Scanning Type	Remediation Type
Create an Nginx User	Linux Action	Linux Action
Rate limit connections to webserver	DSL Configuration Search	DSL Configuration Search
Set client body timeout	DSL Configuration Search	DSL Configuration Search
Verify Upstream SSL	DSL Configuration Search	Manual
Log access and errors in a secure location	CLI Action	CLI Action
Use SSL	DSL Configuration Search	DSL Configuration Search

4.1 Scanning with Canary

This is a selection of some issues scanned by the Nginx package for Canary.

4.1.1 Setting up the Nginx User

Nginx should be run with a user created specifically for the program, just like any other daemon process. This mitigates the risk of privilege escalation through a bug in the software leading to remote code execution.

The user used by Nginx is defined inside the main nginx.conf file. If this file is in the default location, /etc/nginx/nginx.conf, we can simply use grep to determine whether this configuration exists and is set to 'nginx', such as the line below.

```
grep "user nginx;" /etc/nginx/nginx.conf
```

This may give us a false positive if the administrator has set up a username other than Nginx, but we can create a new user for Nginx by using:

```
sudo groupadd nginx
sudo adduser --system --ingroup nginx --home /var/cache/nginx --shell /sbin/nologin nginx
sudo sed -i 's/^user[^;]*;$/user nginx;/' /etc/nginx/nginx.conf
```

Additionally, this user's account should be locked to ensure it can only be used by the system. To check this, we can run

```
sudo passwd -S nginx | awk '{print $7}'
```

to ensure that the output is -1. If it is not, we can remedy it by using

```
sudo passwd -l nginx
```

Which will then lock the user's account.

4.1.2 Rate limiting connections per user

Rate limiting is important for a web server to reduce the load on the machine and mitigate certain types of denial-of-service attacks. Many other techniques that should be used to mitigate attacks, but we can configure Nginx to rate-limit requests which will minimize the impact on upstream servers.

Determining the specific configuration of rate-limiting requires a DSL parser for Nginx configuration files. However, we can check whether the configuration exists in any .conf file by using:

```
grep -ir limit_req /etc/nginx | wc -l
```

And check whether the result is more than 0. Without a DSL parser for Nginx, however, we will not be able to fix the issue precisely. There exist parsers that can read an Nginx file and generate a JSON payload representing the file or vice-versa, so a similar program could be used to modify that representation without the JSON output step.

We could add a 'limit_req_zone' at the top level of the 'http' directive along with 'limit_req' statements in each 'location' directive we find, but this method would not be precise. With our hypothetical Nginx configuration parser, we could read and modify the configuration with

```
conf-parser add http limit_req_zone='$binary_remote_addr zone=one:10m rate=1r/s;'
conf-parser add location ... limit_req='zone=one burst=5;'
```

4.1.3 Set Connection Timeout

Setting the client body read timeout will have similar effects to rate-limiting requests. A malicious client attacking the server could open a connection without sending data in the body of a request which would reduce the number of real connections the server could keep open. There are also other similar configurations available in the ngx_http_core_module which could be used to mitigate certain attacks on the server, such as the client_max_body_size which limits the amount of data that Nginx will read from a single request.

The analysis and remediation of these issues are almost identical to that of the rate-limiting issue and the same limitation arises without an Nginx configuration parser. We can use grep to determine if the configuration is set anywhere, but we are unable to determine if it is set correctly and fix the issue properly.

4.1.4 Verify Upstream SSL

Nginx should always verify SSL for upstream proxies even if the server is on the local machine, such as in a Docker container, but especially if the upstream servers are on other machines. This ensures that the request is routed to a trusted server and there have been no middleman attacks on the request.

To determine if upstream SSL certs are checked for any server, we can use

```
if [ $(grep -ir proxy_pass /etc/nginx | wc -l) -ne 0 ]
then
  if [ $(grep -ir proxy_ssl_certificate /etc/nginx | wc -l) -eq 0 ]
  then
    exit 1
  fi
fi
```

Which will exit with an error code if a proxy_pass statement is used but there is no

proxy_ssl_certificate statement. This issue would need to be fixed manually even if we had our hypothetical Nginx conf parser since the specific certs for the upstream servers would need to be installed in a secure location.

4.1.5 Ensure Access and Error Logging is Enabled

Access and error logs are a crucial part of the Devops cycle since it allows engineers to be alerted of issues in real-time and can help when finding the source of errors. On Nginx, it is important to ensure that logs are set to info level, are rotated on the organization's desired schedule, and are sent to a central location for analysis and integration with alert systems. This can be determined using the following set of commands.

```
# Check that access and error logs are enabled. We want to find statements uncommented and set to
    ↪ info
grep -ir error_log /etc/nginx
grep -ir access_log /etc/nginx

# Check that log files are compressed and rotated on a schedule
cat /etc/logrotate.d/nginx

# Check that a syslog server is set up
grep -ir syslog /etc/nginx
```

Remedying these issues properly would require an Nginx configuration parser, however, we can fix simple issues using sed. For example, if our log setting is on 'warn' instead of 'info', we could fix this with

```
sed -i 's|error_log /var/log/nginx/error.log warn|error_log /var/log/nginx/error.log info|;
```

It would be more precise to fix this issue with a parser that was aware of the Nginx context, and it is not possible to fix more specific issues with log files without it.

4.1.6 Use SSL and redirect HTTP to HTTPS

SSL ensures that communication between the client and server remains encrypted for the duration of the transmission. This ensures privacy for users and mitigates many man-in-the-middle attacks. We can quickly check whether any SSL certificates are installed on Nginx by running

```
grep -ir ssl_certificate /etc/nginx
```

If this fails to return any lines then the Nginx setup is not using SSL for any hosts. However, this still does not ensure that every domain is serving requests with HTTPS. We need an Nginx configuration parser to ensure that every server or location has an appropriate SSL cert.

We can also check if any host is redirecting HTTP to HTTPS with a similar grep command

```
grep -ir "return 301 https://$host$request_uri;" /etc/nginx
```

Similar to the last issue, this will not ensure that the configuration is set up properly if the test passes.

4.1.7 Restarting Nginx

After making any changes to an Nginx configuration setting, it is necessary to restart the Nginx processes. This can be done gracefully with systemd using the command

```
sudo systemctl reload nginx
```

5SECURITY CONSIDERATIONS ON BARE METAL

This section will discuss the configuration settings to consider when setting up a server on bare metal, including the common attacks vectors. There are many reasons that one would use an off-the-shelf OS image, and even a secured image can't protect the user from their own misconfiguration. We must have some mechanism for finding issues and fixing them to ensure that there are no holes.

Table 5.1: Scanning and remediation for native OS issues

Issue	Scanning Type	Remediation Type
Outdated Software	CLI Action	CLI Action
Unencrypted communications	Configuration File Search	Configuration File Find and Replace
Permissive Firewalls or iptables	DSL Configuration Search	Manual
User Readable Files	CLI Action	CLI Action
Privileged users	CLI Action	CLI Action

5.1 Software Updates and Versioning

5.1.1 Reasoning for Keeping Software Up-to-Date

One of the most important steps to take when maintaining a server is to keep the software up to date. Thankfully, this is simple to achieve with most software today. Native package managers can make upgrading a package or software program a one-line command. Additionally, many programs today will have self-update mechanisms which will migrate data automatically, without input from the user.

We must be sure to apply these updates as soon as they become available. All large software projects likely contains bugs or exploits which could be used by an attacker with knowledge of their existence. A study by the RAND corporation examined over 200 historical zero-day vulnerabilities in commonly-used software and determined that zero-day vulnerabilities have an average life expectancy of 6.9 years [RAND Corporation, 2017]. Once an exploit is found and reported to the project's maintainers, there is generally a period between when the vulnerability is fixed in a software update and the exploit is announced to the public. To gain protection from these exploits as soon as possible, it is critical to update software as soon as updates are available.

5.1.2 Scanning for Available Updates

Scanning a system for software that should be updated is as simple as checking the version of installed software and packages against the newest version in major trusted repositories. Upgrading most software and packages will be simple, but there is always the possibility of incompatible versions of packages or software which would require code changes and will take longer to complete. This may be the most common reason for not upgrading packages once available but leaves a system vulnerable to attacks.

If the system is Debian-based, one can run:

```
apt update
apt list --upgradable
```

To update repository lists and determine which packages have updates available. Then, running

```
apt upgrade -y
```

Will automatically upgrade all available packages. Of course, this should be done with care because there is no guarantee that the new packages will not contain breaking changes.

5.2 Remote control

5.2.1 User Access

Assuming that the user is not accessing the machine from a physical terminal, there must be some kind of remote connection to the machine. This connection should be encrypted and secured by cryptographic keys to prevent attacks. A basic SSH connection will satisfy these requirements, so alternatives must be avoided even if the communication is over a private network. We can check whether the current user has logged in via ssh by using 'who am i'. Additionally, we will want to check the configuration file to ensure that SSH is set up to disallow passwords.

We can check if the current user is logged in via ssh by using:

```
ps aux | grep "sshd: $(whoami)@$(who am i | awk '{print $2}')" | wc -l
```

Which will print a '1' if the current session is using ssh. We can then check whether the ssh configuration is secure by using:

```
cat /etc/ssh/sshd_config | grep "^PasswordAuthentication"
cat /etc/ssh/sshd_config | grep "^PermitRootLogin"
```

We should either have no result or the setting is set to 'no', such as 'PasswordAuthentication no". We can then fix these issues with a simple sed command, such as:

```
sed -i 's/PasswordAuthentication yes/PasswordAuthentication no/g' /etc/ssh/sshd_config
```

5.2.2 Encrypted Communications

Any unencrypted communication is vulnerable to a man-in-the-middle attack, as explained by John Richter [J. Richter, 2017]. This can lead to information leaks and possibly unauthorized access should any passwords be transferred. We can classify this as a vulnerability in the network or firewall of the system. All remote control of the machine should be done over encrypted channels, and if not then anyone who can intercept the packets could determine exactly what was being sent to the system, including plaintext passwords.

If the user is not currently logged in via SSH, the solution would be to disable whatever method was used to access the machine and use SSH instead. Today, the main way to access a remote machine is via SSH, so we can assume that any machine should have the ability to host an SSH server daemon. However, it may be necessary for the host to use some remote GUI system, in which case we should ensure that the communications are over a secure channel such as HTTPS and the authentication meets a standard set by the organization.

Checking whether communications are done over HTTPS should be done at the application

level instead of at the OS level. However, if we wish to do so, it is possible to set up firewalls using iptables to block all outbound traffic that is not HTTPS or SSH. If we wish to do so, we can set this with

```
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A OUTPUT -t filter -p tcp --dport https -j ACCEPT
iptables -A OUTPUT -t filter -p udp --dport https -j ACCEPT
```

However, this will certainly break many applications using the network, which is why it is best to ensure secure communication at the application level.

5.2.3 Remediation

To ensure that ssh is enabled and enforced on a system, the following steps should be taken:

1. Install sshd. Sshd should be pre-installed on most operating systems but can be downloaded from a trusted package repository or be built from source.

2. Start sshd on system boot. If systemctl is used, we can do this by running 'systemctl enable sshd'.

3. Generate keys for all users. Ensure all users have generated private keys and transferred them to the server.

4. Disable password authentication. Passwords should never be used for authentication of a user over ssh. This can be disabled by editing the /etc/ssh/sshd_config file. Additional measures can be taken when editing this file, such as preventing port forwarding or root logins.

5.3 Network Security and Firewall Configuration

5.3.1 Importance of Firewalls

Networks should be configured using principles of least permissions, with only the necessary ports and addresses open for connection. If the machine is hosting virtual machines, we can use it as a firewall for the guests by setting rules using iptables. In a public cloud environment, it may be necessary to allow incoming connections to the guests. If this is the case then we should make sure that the iptables configuration can be changed programmatically, or another solution must be used.

5.3.2 Necessary Configuration and Remediation

We can check the iptables configuration programmatically but the needs of each organization will be dramatically different. Therefore, it is not feasible to determine programmatically whether the configuration is an issue without knowledge of the system requirements. However, we can still check whether firewall rules are set in place and give the user a warning if nothing is configured.

5.4 Reduce User Privileges to a Minimum

5.4.1 Principle of Least-Privilege

The host machine must be set up using principles of least permissions. We hope that a malicious guest VM or compromised container cannot escape from its process, but the host should still anticipate this possibility and limit the attack surface on the native operating system. A sysadmin should take care to ensure that the system is up-to-date and the programs running have been secured but this can be time-consuming and error-prone to do by hand. By using a tool such as Canary, we can quickly evaluate a system and fix trivial errors.

For example, we can check that

5.4.2 Leaky Files

One way that systems become compromised or secrets are leaked is from seemingly innocuous files which contain secrets or system information. Log files can contain information such as passwords in plain text or keys which could be used to gain access to other systems. These files can be put into a list and checked by Canary by checking whether the user running the process can read or write to the files. This can eliminate the vast majority of trivial security issues, and prevent any damage from occurring if a process is compromised or an unauthorized user gains access.

As an exmpale, if a system is using Nginx for a reverse proxy we can check the permissions on the log files with

```
test -r /var/log/nginx/access.log && echo 'readable'
```

Which will print 'readable' if the file is readable. Any user can do this against a list of filenames. If the user finds an interesting log file, it would be simple to send the file contents back to the terminal or download it to analyze later. This issue could be resolved by reducing the permissions on the file or by removing the user's membership from a certain group. If the user was in a group called logreader for example, we could remove this access by running

```
gpasswd -d [username] logreader
```

Or if we wish to reduce the file to only be readable for the file's owners:

```
chmod 600 /var/log/nginx/access.log
```

It is good practice to refrain from passing sensitive data into log files whenever possible, but even without sensitive data a log file could be used to infer information about a system which could give an attacker clues as to weak points in the system's security.

5.4.3 Phishing

We should also consider that a user's passwords or private keys could fall into the hands of an attacker through a simple phish attempt. Phishing attempts are extremely common and an alarming number of people will fall victim to them, as shown by a study by Software World Intelligence [A.P. Publications Ltd, 2020]. Administrators should assume that users will at some point give up their passwords or keys and that accounts could be used to find sensitive information. Although Canary can't detect whether a user or group should have access to a specific file, it can easily determine whether that file is readable and report it to the end-user.

5.4.4 Malicious Actors and Code Vulnerabilities

Consider a system that has an unknown zero-day vulnerability that can be exploited. For example, consider the recent Linux kernel vulnerability CVE-2022-0185, which enables container breakout [National Institute of Standards and Technology, 2022]. The attacker may use this to gain access to the system and execute arbitrary code. However, if we make sure to reduce the permissions for users down to the bare minimum, we may be able to save any sensitive data from leaking and prevent the actor from gaining control of the system. An attack of this manner can be considered privilege escalation since the user who was intended to only do one thing, such as run a file server, would now be able to execute programs they were never intended to execute.

Preventing an attacker from gaining sudo privilege is of utmost importance, so a user with sudo access should never be used to run long-running programs on the host system unless necessary. Canary can check whether the current user has sudo access or whether any processes in a list of PIDs are running with sudo privilege. Of course, this can't be easily fixed with another script so the system administrator must address the issue. Many processes must run with sudo permissions

to work correctly, but in most cases, we can simply create a new user with only the permissions necessary for the program to work.

AppArmor or similar software should be utilized to minimize the damage that a malicious process can cause to a system. These profiles can be used in conjunction with a user with minimal permissions to ensure that intruders are prevented from accessing sensitive data.

6DOCKER CONSIDERATIONS

This chapter will examine security considerations for a system using Linux containers, specifically Docker.

6.1 Introduction

6.1.1 Overview of Docker

This section will examine the default security settings of Docker, how they could be tailored to an individual deployment's needs, and what attack vectors exist when using the Docker engine. Docker is the most commonly used container platform in use today. The ease of use combined with lower performance overhead compared to Virtual Machines has led to container usage to become just as common as VMs. Although Docker is very useful in development situations we also see its use in production systems, which will be the focus of this discussion. Docker relies on many features in the Linux kernel to ensure that the processes are separated from others on the same machine, but does not use any hardware features in the CPU like KVM or other hypervisors would be able to take advantage of.

Kubernetes is a very prominent software used to orchestrate hundreds or even thousands of containers across multiple machines for load balancing and availability purposes. This paper however will focus on simple configuration practices and will not discuss K8s, although the principles still apply to containers running on K8s.

6.1.2 Performance Compared to Virtualization

Docker and other container engines can achieve lower performance overhead compared to virtual machines by only containing the necessary libraries and packages for its use and depending on the host's kernel for the traditional work of the operating system. This creates a much larger attack surface since the entire OS is shared, although kernel features such as namespaces and cgroups are utilized to separate the container's processes from others on the same machine. Linux provides namespaces for system resources, networks, processes, process communication, users, time, and domains. These can be read in greater detail on the Linux namespaces man page [**?**]. Linux namespaces provide the basis for containers themselves and are critical to the security of containers.

Newer CPU technologies like Intel's VT-x help close this gap between virtualization and containers but running a full OS within another OS will certainly have overhead costs compared to a native process.

6.1.3 Deployment Considerations

Docker has become the de-facto containerization solution since its inception and can be credited with the shift of containerization from virtualization [C. Tozzi, 2017]. The tools provided by Docker make it very easy to create new containers, share images, and reproduce the systems used by others. Additionally, container orchestration technologies such as K8s has made adopting microservices much more attractive due to its built-in scaling and balancing.

These tools often choose sane defaults for security, but there can still be gaps in this setup. These issues are compounded by the fact that many developers may overwrite these defaults to make it easier to test. The same developer may forget to go back and configure the system cor-

Table 6.1: Scanning and remediation for Docker issues

Issue	Scanning Type	Remediation Type
Docker Socket Access	CLI Action	Configuration File Search and Replace or Manual Action
Docker Volumes	ClI Action	CLI Action
AppArmor and seccomp	CLI Action	Manual
Docker network setup	CLI Action	CLI Action

rectly and may not even know why the setting was there. Any organization with an experienced engineer would catch this, but there are no guarantees. In this way, issues may creep into production systems.

When evaluating the security settings of a system using Docker, we must take note of where the system will be deployed and its intended uses. Many developers will use Docker on their development system since it is easy to replicate a production environment locally, but this type of deployment does not necessarily need extra security above the default levels unless the system is being tested specifically for vulnerabilities. A production system hand should have the most restrictive settings possible and should be thoroughly evaluated for security gaps.

6.1.4 Relation to Container Software

The security considerations used by Docker to isolate containers from each other and their host can be used as a starting point for other containerization systems. Docker may be the most popular system, but there are others that are used by other platforms and may be made necessary due to platform constraints. In the following section, we will evaluate the configuration needed to set up Docker as well as display how to check whether these parameters are properly set.

6.2 Securing the Docker Socket

The daemon is the entry point to the container runtime and the containers themselves, so any process which can access the socket has control over the entire Docker system. We will also want to ensure that each container uses its own namespace and that the namespace is sufficiently restrictive.

6.2.1 Importance of the Docker socket

Any process which can interact with the Docker daemon could impact any container or image on the system, so it is important to secure access to this UNIX socket or TCP port. If the Docker containers will reside on a single machine, this can be done by simply restricting access to only the user who will be running the daemon processes. If it's required that the system must expose the daemon to the external network, the socket should be secured with TLS to ensure the identity of the connection. It is easy to check whether users can read the docker socket but if the socket is exposed to the network it is more difficult to determine whether it is secure.

We can check whether the current user has access to the docker socket by simply running

```
1 docker ps
```

Docker must be run with root, so having access to the docker socket is similar to having root access on the machine. Most users should not be able to access docker on production instances besides engineers responsible for running the system.

6.2.2 Docker socket protection

If the Docker socket is exposed to the network then additional precautions should be taken to keep it secure. The unix socket requires root permissions by default, but once exposed to the network it may be accessed by any machine which knows the port and IP combination. To miti-

gate attacks, tlsverify should be required by the docker socket, and certificates should be generated and only distributed to trusted systems or users. These steps are outlined in Docker's documentation [**?**].

We can check if the docker socket is exposed over the network with

```
sudo ps aux | grep -E 'dockerd.*?-H.*?tcp://' | wc -l
```

This checks the currently running dockerd process to see if it was started with a tcp host. If so, then the output will be greater than one. If we have started the daemon listening over tcp, we also want to verify tls certificates. This can be checked by running a similar command:

```
sudo ps aux | grep -E 'dockerd.*?--tlsverify | wc -l
```

This vulnerability can be fixed by generating tls certificates and specifying their path in the command line arguments for docker. These instructions are taken from Docker's documentation.

```
openssl genrsa -aes256 -out ca-key.pem 4096
openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem
openssl genrsa -out key.pem 4096
openssl req -subj "/CN=$HOST" -sha256 -new -key server-key.pem -out server.csr
echo subjectAltName = DNS:$HOST,IP:10.10.10.20,IP:127.0.0.1 >> extfile.cnf
openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -
    ↪out server-cert.pem -extfile extfile.cnf
openssl genrsa -out key.pem 4096
openssl req -subj '/CN=client' -new -key key.pem -out client.csr
echo extendedKeyUsage = clientAuth > extfile-client.cnf
openssl x509 -req -days 365 -sha256 -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -
    ↪out cert.pem -extfile extfile-client.cnf
rm -v client.csr server.csr extfile.cnf extfile-client.cnf
chmod -v 0400 ca-key.pem key.pem server-key.pem
chmod -v 0444 ca.pem server-cert.pem cert.pem
dockerd -H tcp://0.0.0.0:2376 --tlsverify --tlsacert=ca.pem --tlscert=server-cert.pem --tlskey=
    ↪server-key.pem
```

The client certificates should be distributed to trusted machines and proper access control settings should be enforced to protect them. The daemon should not be exposed over tcp without these parameters, even if it is on a private network.

6.3 Securing Docker Volumes

Docker volume misconfiguration can pose a threat to a system two ways and could lead to denial-of-service or info leak attacks.

6.3.1 File System Size Constraints

The first is through lack of size constraints. A container using a volume without size constraints could fill up the filesystem with data, preventing other users from writing data and potentially crippling the IO performance of the machine.

Docker containers by default have an associated volume with a 10Gb size limit. However, standalone volumes will not have a default limit and so should be configured as needed. To check if a specific volume has a size limit set, use

```
sudo docker volume inspect [volume] -f '{{ .Options }}' | grep size | wc -l
```

Depending on which storage driver the volume uses, we may not be able to limit its size, so new volumes and containers should use storage drivers which support this feature. To create a new volume and attach it to a container, one should use

```
docker volume create --driver local --opt type=tmpfs --opt device=tmpfs --opt o=size=100m test-
    ↪vol
docker run -d redis:alpine --mount source=test-vol,target=/var/log/redis
```

6.3.2 Shared Files

The second way a volume could pose a threat is through shared files. A Docker volume could also be shared between the host or other containers. Should the shared volume be mounted at a point that contains sensitive files in the subtree, then these files would be visible to the other container and could potentially leak data to a malicious actor.

Any threats exposed this way can be scanned in the same way that they can be scanned on

the native filesystem. They can also be fixed in the same way, which is updating the permissions on the file or reducing that user's permissions.

One docker-specific consideration is the mount location. Any sensitive files on the volume should be kept separate unless the files need to be read by other containers or the host. This remediation must be done on the application level, but we can start Docker containers with separate volumes mounted in a separate location in the container to ensure that the data written is only used by that container. We could do so by using a start command such as

```
docker run -d --name nginx-example --mount source=nginx-logs,target=/var/log/nginx nginx:alpine
```

Which would specify that the Nginx logs should be directed to the new volume containing only log files.

6.3.3 Volume issue remediation

One solution for both problems is to designate a separate partition for the Docker volumes to reside. This will ensure that a runaway container can't attack the host through the file system and no files used by the host will be accessible by a container. If we wish to create a new partition for each Docker container, we could automate the process by using a sequence of commands as shown below.

Partition an existing device:

```
parted [device]
```

Create a docker volume on our new partition:

```
docker volume create --driver local --opt device=:[partition mount point] [volume name]
```

Finally, we should restart the container with the new volume attached:

```
docker run -d --mount source=[volume name],target=[mount point in container] [image]
```

6.4 Kernel Security Features

Containers are more susceptible to issues since the OS is not fully virtualized as it is within a hypervisor. To mitigate issues, Linux namespaces, AppArmor profiles, and seccomp configurations should all be set up properly to allow the bare minimum features required for the container. These are all features included in the Linux kernel.

6.4.1 Default Protections: Namespaces and AppArmor

A container is not a jailed process but does use kernel features to sandbox applications from the rest of the system. Linux namespaces enforce the separation of resources for a well-defined set of processes, which is a necessity for containers to run without full virtualization. Unless otherwise specified, each Docker container will be created with its unique namespace.

```
sudo lsns | grep $(sudo docker top [container name] | awk '{if (NR!=1) {print $2}}')
```

Docker, LXC, and other container runtimes will all use this feature by default. If there is an issue with namespaces on a specific container then it is likely either a bug in the container runtime or an ongoing attack on the system, so resolution of the issue would be non-trivial.

An analysis of container security issues by Sultan, Ahmad, and Dimitriou found that most container security considerations are resolved by features in the Linux kernel, chiefly namespaces and AppArmor settings [Sultan, Ahmad, and Dimitriou, 2019]. They also found that protection of the container from the host itself can be achieved through hardware means. This shows that as long as sane defaults are not removed, the separation between containers and between a container and the host is not an area of concern for the administrator.

6.4.2 Seccomp settings

Each container will receive a default seccomp setup which should prevent the majority of system calls that could undermine the security of the overall system. However, this default configuration could be less permissive than necessary than the container needs since a container may only perform simple tasks during its operation. A technique used by Wan, Lo, and Xia show that we can trace the system calls used by a container and use the result to limit the seccomp configuration of the container to disallow any other system calls [Wan, Lo, Xia, and Cai, 2019]. This should be done whenever possible to limit the attack surface on the container when used in a production environment.

Once we have created a new profile for a specific container, we can start the container with that profile, such as

```
docker run --security-opt seccomp=[path to profile] [image name]
```

6.5 Docker Network Settings

A new container will use the default network, which allows access to all other containers in the network. Unless this is the desired behavior, containers should have their own networks or use a network containing only the containers through which communication is necessary. The network used by a container can be specified when the container is created.

Checking which networks are used by a container can be checked using the Docker client. This will give us a json map of the connected networks and the network information.

```
docker inspect [container name] -f "{{ json .NetworkSettings.Networks }}"
```

If we find containers that are using the default network or are accessible by containers that are not required by the system, we can create a new network and attach it to the container using

```
docker network create [network name]
docker network connect [network name] [container name]
docker network disconnect bridge [container name]
```

7LIBVIRT HOST CONSIDERATIONS

Libvirt is the default interface used with KVM and other virtualization backends on many Linux distributions. These considerations use KVM as an example, but other backends such as Xen or VirtualBox could also be used with libvirt.

7.1 Overview of KVM

7.1.1 Guidelines and Considerations

The first step when setting up a server hosting KVM guests is to verify if general guidelines for Linux machines have been followed. A list of common issues can be found in chapter three of this paper. However, there are some additional considerations when KVM is in use, especially if the code on the guest should not be trusted such as in a public cloud environment. Most of the additional considerations are focused on the additional devices which may be used with a VM as well as IO access from the guest to the external network.

KVM allows for the guest's CPU instructions to be sent directly to the processor if hardware virtualization is supported by the CPU. This drastically reduces the attack surface and allows us to focus on the components enabling communication between a guest and any external device or network. The guidelines outlined by Red Hat, Inc. are a good starting place for determining the most secure system configuration [Herrmann, Zimmerman, Radvan, Richardson, Moore, Seifried, and Jorm, 2019].

7.1.2 KVM Performance

Virtualization today relies heavily on KVM in the Linux kernel. Most cloud providers will use this software in order to create and manage virtual machines, although they may use

proprietary software to manage the virtualization frontend. If the CPU has hardware virtualization enabled then KVM uses hardware virtualization to accomplish the virtualization of an operating system. This has the benefit of separating the entire guest OS from the host OS without much of the performance drawbacks of software virtualization. However, it is important to note that even with hardware acceleration KVM is less performant than Linux containers, as shown by Chae, Lee, and Lee [Chae, Lee, and Lee, 2017]. Therefore, a KVM-based environment should be chosen for the security benefits whenever the performance drawback is acceptable.

On a machine hosting VMs, special attention must be paid to the IO operations allowed by the guest machines since this is the only part of the VM which must be processed by the host OS. Default settings may be acceptable depending on the virtual machine monitor used but should still be verified by the administrator.

7.1.3 Comparision to Container-Based Environments

In a shared-tenant environment, the use of virtual machines is preferred to the use of containers. Albakri, Shanmugam, Samy, Idris, and Ahmed outline several risk assessment considerations for an organization looking to move assets to a public cloud. They found that the responsibility for security in a cloud computing environment is shifted to the cloud provider in a public cloud, and any organization looking to shift resources to this environment should consider this when evaluating security risks [Albakri, Shanmugam, Samy, Idris, and Ahmed, 2014]. Due to this, a cloud provider looking to sell its resources should ensure that all possible risks of attacks between VMs have been mitigated.

Table 7.1: Scanning and remediation for libvirt issues

Issue	Scanning Type	Remediation Type
Libvirt audits	CLI Action	CLI Action
Secure Physical Storage	CLI Action	Manual
Encrypt Storage Pools	Configuration File Search	Manual
Disable unused external devices	N/A	Manual
Avoid Untrusted VM Images	N/A	Manual

7.1.4 Overview of issues

7.2 KVM Host Considerations

7.2.1 Set Up libvirt Audits

Detailed log files are essential to investigating errors that occur on production systems, but can also be helpful when investigating security issues. As recommended by RedHat, usage of auvirt can help by recording all events that happen during operation. In public cloud environments, this should be combined with logging of events relating to the user's use of the system configuration. For example, any changes to networking should be logged, which could then be used to analyze the VMs events to identify anomalies.

We can check if audits are enabled on the system by running

```
cat /etc/libvirt/libvirtd.conf | grep '^audit_level'
```

We should see an entry in the file for audit_level which is not commented. If we are not auditing the system, we can enable it with

```
sudo echo "audit_level=2" | sudo tee -a /etc/libvirt/libvirtd.conf > /dev/null
sudo systemctl restart libvirtd
```

7.2.2 Securing Physical Storage Devices

A guest VM with access to an entire storage device may be able to attack the host or other guests by rewriting volume labels on the device. Unless the system requirements dictate that a guest should access an entire physical device, this should be disallowed. There are a large number of possible file system types allowed with KVM but the only one that could cause issues on the host itself is access to an entire disk. If access to a physical drive is required, we should ensure that it is done through a partition instead of the device. Additionally, we should make sure that the host machine does not use file system labels to identify the file systems in the fstab file.

It is more secure to grant a guest access to a partition instead of an entire device. A partition can be resized or moved as needed, something not possible with a single drive. Additionally, the storage device can be shared between many guests, and partitions will be cleared and reused as needed without affecting other partitions on the same device.

We can check the guest OS's storage settings by using virsh. We want to check that the guests do not have access to entire devices and the host is not using file system labels in the fstab file.

```
for pool in $(virsh pool-list --type disk | tail -n +3 | awk '{print $1}'); do
    virsh pool-info pool
done
cat /etc/fstab
```

This will print out information about each disk device along with the fstab file. The administrator should then check this against the fstab file to ensure the guests do not have access to entire devices and they are not using the labels in the fstab file.

7.2.3 Disable Access to Physical Devices

A similar issue to granting a guest an entire storage device is granting a guest access to any necessary hardware such as USB devices. Depending on system requirements this may be necessary for a special system, but this would still increase the attack surface significantly and should be avoided.

Since this paper focuses on security considerations in cloud environments, we will not discuss the steps needed to secure physical devices and instead study the steps taken to use KVM without support for these devices.

In a public cloud, there is no need to enable any external devices unless it is a business requirement. The vast majority of cloud computing will only require access to block storage and the network. There exist entire virtual machine monitors built around the concept of stripping the VMM down to only what is necessary for cloud computing.

7.2.3.1 Firecracker and gVisor

Firecracker, developed by Amazon for AWS Lambda, is probably the most widely-used example of this today. This VMM replaces QEMU and reduces the attack surface significantly by omitting any device drivers for external devices. gVisor, developed by Google for its App Engine product on GCP, is an example of a similar software that accomplishes the same goal differently. gVisor is more similar to container software as it does not utilize KVM or virtualization. However, it does create a sandboxed execution environment by implementing a 'user-space kernel' which provides a layer between each guest container and the Linux kernel itself through which the guest's system calls are made.

Both of these approaches isolate the guest code from the host's kernel, but in doing so

execute far more kernel code than a simple Linux process on the host machine would [Anjali, T. Caraza-Harter, M. M. Swift, 2020]. This means that although the host will now be more protected from traditional malware, there will be more opportunities for an unknown kernel bug to impact the host and steps should be taken accordingly to protect the host.

7.2.3.2 KVM without QEMU

QEMU is a widely-known and used frontend for KVM virtual machines. However, QEMU has been the source of many bugs which could lead to remote code execution and therefore may not be the most secure choice in a production environment [M. Talbi, P. Fariello, 2021]. It attempts to emulate as many device drivers as possible so that a virtual machine can function similarly to a native machine. This does greatly increase the attack surface since the majority of these devices would not be needed in a server environment.

Firecracker, developed for AWS, addresses these issues by only supporting the bare minimum IO devices needed for a server. This means that it does not even implement a full keyboard driver. Firecracker is purely optimized for server workloads, especially serverless workloads as it boasts a < 125ms startup time [Amazon Web Services, Inc., 2022]. Other than its lack of support for additional IO devices, it works very similar to a QEMU virtual machine. It can be controlled by an HTTP api or via command line. To start a simple Firecracker VM, only a Linux kernel and a file system image are required after Firecracker is installed on a system with KVM enabled.

```
cat <<EOF > ./vmconfig.json
{
  "boot-source": {
    "kernel_image_path": "./vmLinux.bin",
  },
  "drives": [
    {
      "drive_id": "rootfs",
      "path_on_host": "./rootfs.ext4",
      "is_root_device": true,
      "is_read_only": false
    }
  ],
```

```
}
EOF
# start firecracker
sudo firecracker --no-api --config-file ./vmconfig.json
```

7.2.3.3 LXC without system calls

Google's gVisor takes an approach to sandboxing more similar to containers than VMs. gVisor functions by utilizing LXC containerization but handles the container's system calls itself instead of passing them directly to the kernel. This means that gVisor's 'Sentry' component is essentially the VM for the container through which they interact with the host machine [The gVisor Authors, 2022]. The use of gVisor with containers instead of a virtual machine will lead to greater performance than a fully virtualized setup through KVM while minimizing the security drawbacks of a container-only system.

7.2.4 Secure VM Image Storage

VM images should be stored in a single, secure location. Libvirt stores images in /var/lib/libvirt/images by default, which should not be moved unless needed for a system requirement. The images should not be accessible by any users other than the ones who must read or modify them. Images stored at rest are a target for a malicious actor since injecting malicious code can allow an attack to remain ongoing and undetected for a long time, similar to a traditional boot sector virus. This virus would be recreated every time a new VM is created, allowing the attacker to steal information from an organization for an ongoing period. As long as it is not publicly readable or writable and the relevant AppArmor profiles are set up, the images can be considered secured. We can check this along with the other file permission checks.

We can check the security policies on libvirtd by using

```
sudo cat /etc/apparmor.d/usr.sbin.libvirtd
```

Libvirtd is the daemon process enabling the creation of VMs on Linux machines with libvirt installed. The AppArmor policies on libvirtd are considered relatively permissive, but strict policies are automatically created for the guest VM which enforce the separation of the guest VM and host OS.

7.3 Guest VM Considerations

As with a KVM host, all procedures to secure a bare metal machine should be followed when setting up a guest VM. A guest VM will likely not even know it is a VM, so it should be secured in the same way as the host. However, there are some considerations when configuring the guests that should be taken into account to prevent holes in the security setup.

7.3.1 Isolate Guest VM Networks

A guest VM should function as if it was an independent system especially if the VM is in a public cloud. Unless the requirements state that the VM guests should be able to communicate, there should be no private network communication allowed between the guests. This simply means that guest VMs should not be attached to the same private network except in cases where the user wants to configure their VMs this way, however, this would be out of the scope of this paper.

To examine the networks on a host machine, we can use

```
virsh net-list
```

If the network configured allows private communication between guests, we should create a new network for each VM and add the guest to the new network.

```
# Detach the guest from a network interface
virsh detach-interface [VM] [network type] --live
# Create a new network for this guest
virsh net-create [xml file]
# Attach the new network to the guest VM
virsh attach-interface [VM] [network type] [network source]
```

This will likely need to be done by an administrator in a live environment and would likely

break any applications on the guest VMs. A better approach is to create the VM with the proper network interface from the start instead of retroactively modifying it.

7.3.2 Storage Encryption

A guest OS's storage should be encrypted whether it is stored on a physical disk partition or a network drive. In both instances, the physical location of the data would likely be shared with other VMs and could be accessible by another machine with access to the drive. To prevent any unintentional data leak or leaks from malicious actors, the storage should always be encrypted whether it is at rest or in transmission, and the encryption keys should be stored securely.

Any unencrypted data could be read easily during a man-in-the-middle attack if a network drive is used, or directly from the disk if an attacker has root control of the host. To check whether data will be encrypted by the kernel, we can examine the XML definition for the storage volume. In the following XML, the encryption tag tells libvirt to create a volume using a secret key associated with the UUID.

```
<volume>
  <name>example.luks</name>
  <capacity unit='G'>1</capacity>
  <target>
    <path>/var/lib/libvirt/images/example.luks</path>
    <format type='raw'/>
    <encryption format='luks'>
       <secret type='passphrase' uuid='20755d8a-5ec5-4fd8-94cf-95aa7aea9ba4'/>
       <cipher name='twofish' size='256' mode='cbc' hash='sha256'/>
       <ivgen name='plain64' hash='sha256'/>
    </encryption>
  </target>
</volume>
```

We should always be sure to initialize the VM with an encrypted storage device, such as with

```
virsh attach-disk --config [vm name] [storage xml definition]
```

If this is not possible, such as with a VM which has already stored data, then the data

must be copied to an encrypted location and the storage device must be switched. This would not be a trivial operation to accomplish depending on the role of the VM, and certainly difficult to accomplish with incurring downtime.

7.3.3 Hardened VM Images

Hardened VM images are images that have been pre-configured with a set of security rules in place. Use of this type of image can prevent many user errors since the image has been officially verified and is shared by many organizations. This approach is likely preferable to using a stock VM image such as the one provided by Ubuntu or RedHat but should still be supplemented by a security scan to ensure that the image's best practices are not out of date or misconfigured.

This approach might still not be possible when working with an existing system such as a long-running guest VM. There also may not be an image provided that meets the requirements of an organization. If the requirements of the project dictate that a new hardened image can't be used for a guest VM, then it is possible to 'harden' the existing system and create an image of it for the organization's use.

Once a system has been configured to meet some security standard, we can create a new VM image based on the system using

```
virt-install \
  --name [vm name] \
  --cdrom [path to iso] \
  --os-variant [os type]
```

This will create a copy of the current system when installed, so a blank version of the

system should be used to create the ISO. If it is not possible to get a 'blank' version, more complex steps would be required to only copy necessary files to the ISO.

8CONCLUSIONS AND FUTURE WORK

8.1 Future Work

8.1.1 Canary Packages

Canary functions as the engine which runs security audits but does not come with a thorough set of tests included. An important next step for Canary is to complete a production-ready set of tests for an application. Although it would be best for these packages to be maintained by the same organization which maintains the application itself, it is more likely that these tests would be created by open-source contributors. It is critical for the application's viability as a security solution to have a full suite of tests for many commonly-used applications or technologies.

8.1.2 Canary Repository Interace

Open-source repositories such as NPM have succeeded due to their ease of use as well as the ease with which contributors can submit their packages to the repository for hosting. The Canary repository is currently a simple file server, but a full user interface to allow package maintainers to submit packages for review would allow for the repository to function as a viable way for users to share their packages for security audits.

8.2 On the Utility of Security Scanners and Audits

Automated tools like Canary can save an engineer's time and allow them to focus on issues that require attention instead of tasks that can be automated once an issue has been detected. They can also allow for an organization to run regular audits on itself to demonstrate that they have set up a system using the industry's standards to mitigate attacks.

In an environment where vulnerabilities are more likely to come from an overlooked setting

on the part of an engineer instead of a bug in the application in question, an automated scanner to prevent these issues would be welcome. Security in cloud environments is difficult to accomplish. The attack surface can be large even with modern tech stacks and standards are constantly changing to adapt to new research or software. With this in mind, there is a place for automated scanners and configuration tools to improve the workflow and security practices of many organizations.

Figure 2.1: High-level overview of Canary program execution.

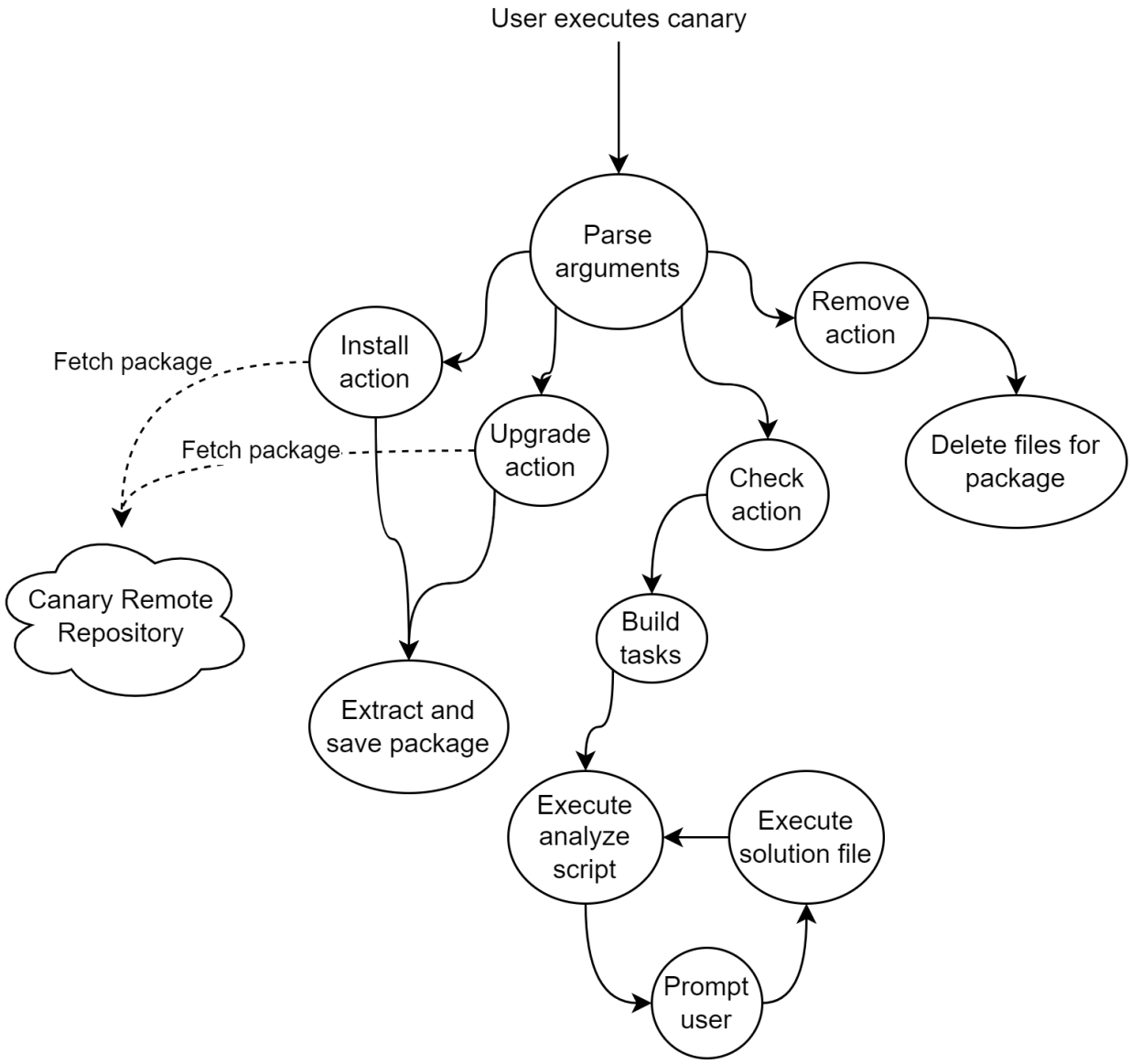


Figure 5.1: Out of date software can lead to a number of different attacks. Once an attacker has determined that a system is using an exploitable version of software, they can craft a payload with input that will trigger the exploit and execute arbitrary code that will damage the system or leak information.

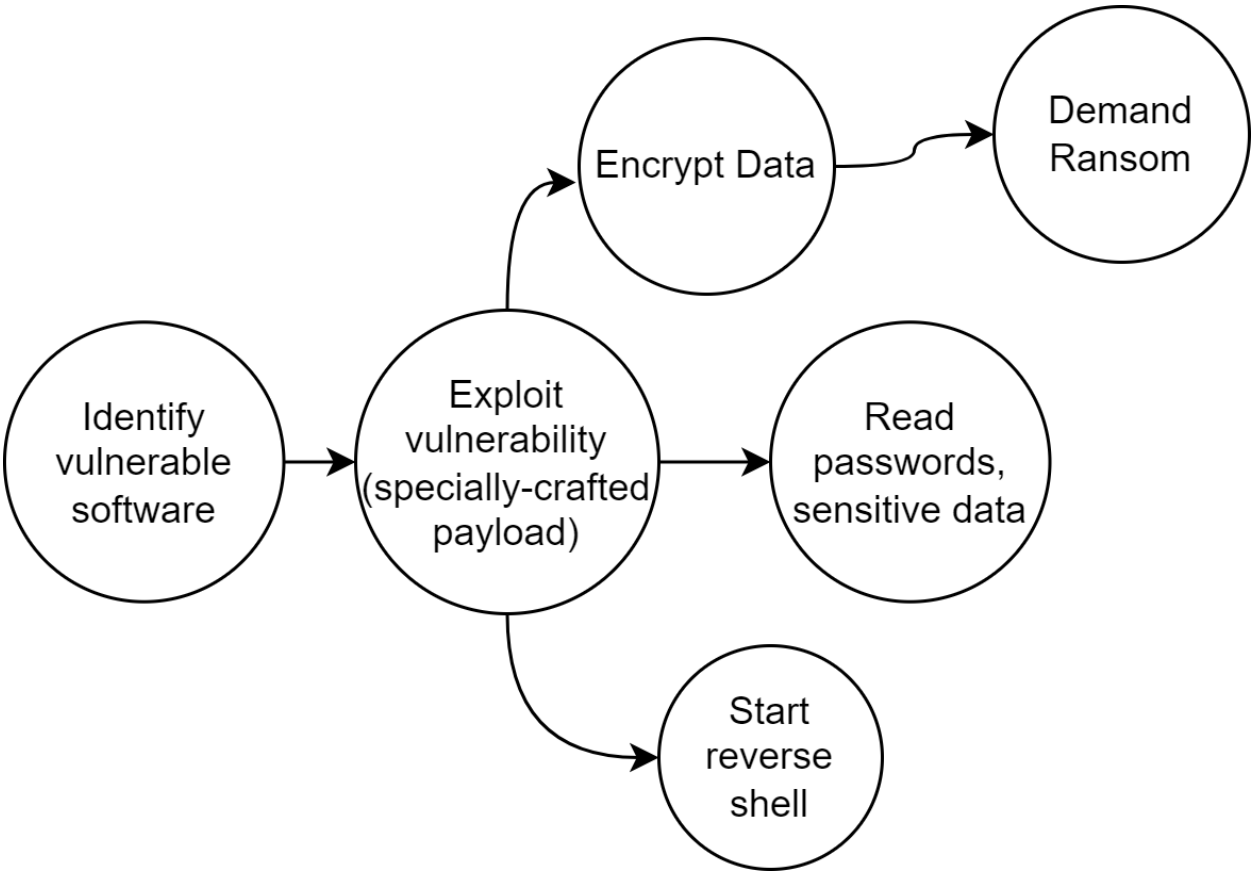


Figure 6.1: Docker and other container runtimes depend on Linux kernel features to isolate processes. This means that the application code is entirely in user space, including the container runtime.

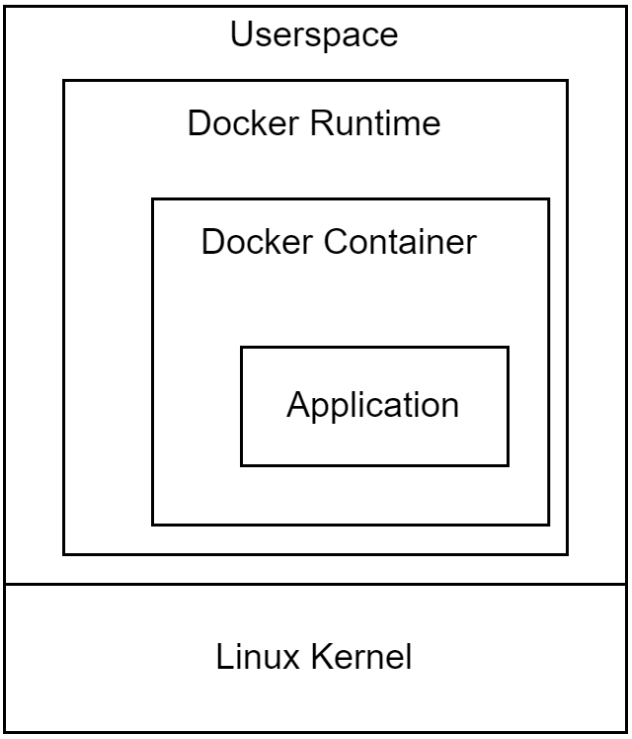


Figure 6.2: The Docker socket is the point of control for everything related to Docker containers. Access to this would enable the attacker full access to the system.

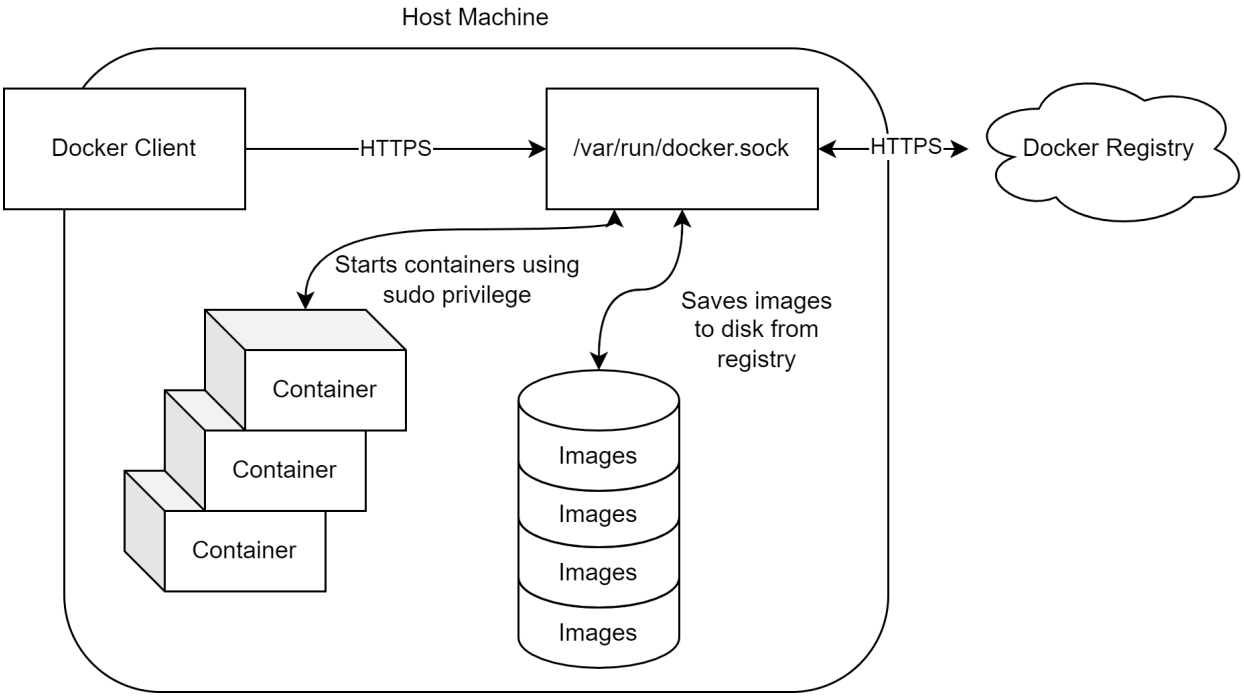


Figure 7.1: Device driver code required for external devices contributes significantly to the attack surface of a hypervisor.

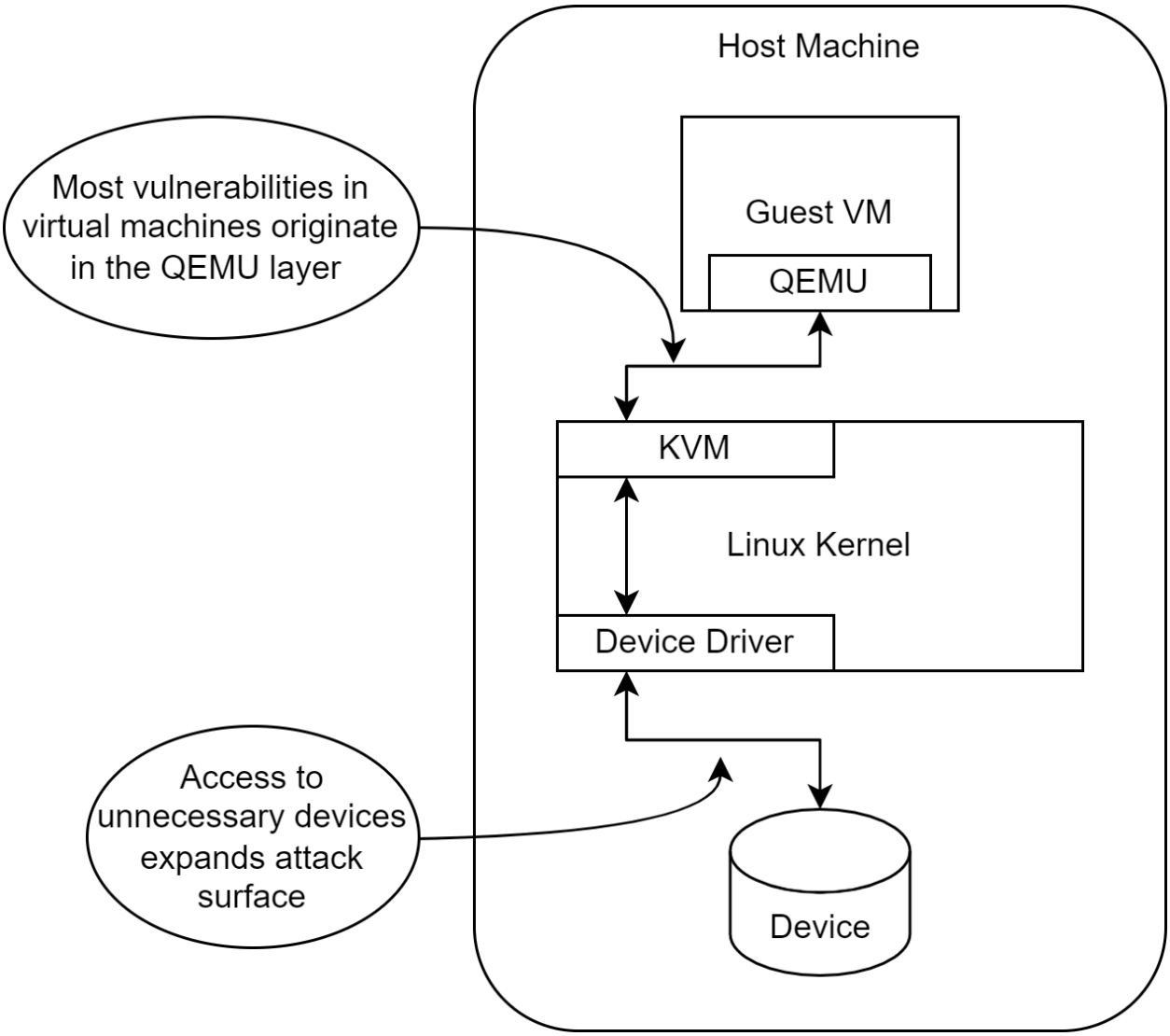
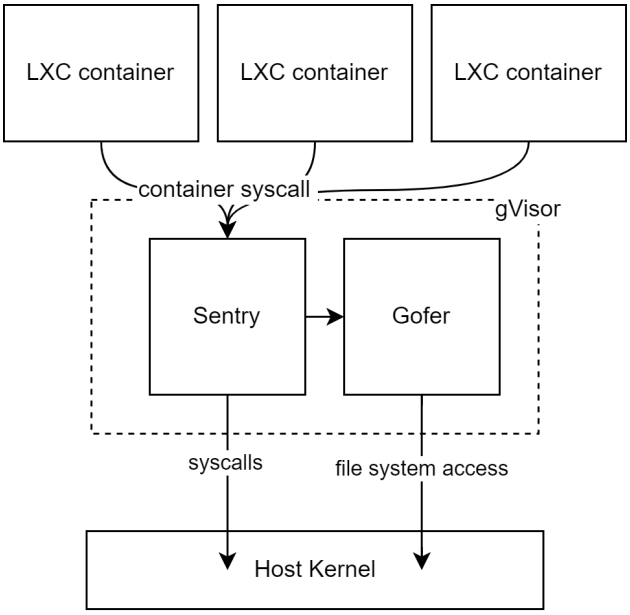


Figure 7.2: gVisor functions as a layer between LXC and the Linux kernel.



REFERENCES

S. H. Albakri, B. Shanmugam, G. N. Samy, N. B. Idris, and A. Ahmed (2014). Security risk assessment framework for cloud computing environments. *Blackwell Publishing Ltd*.

S. Sultan, I. Ahmad, T. Dimitriou (2019). Container Security: Issues, Challenges, and the Road Ahead. *IEEE*.

Z. Wan, D. Lo, X. Xia, L. Cai (2019). Practical and effective sandboxing for Linux containers. *Springer US*.

M. Chae, H. Lee, and K. Lee (2017). A performance comparison of linux containers and virtual machines using Docker and KVM. *Springer US*.

Y. Choi, M. Park, J. Eom, and T. Chung (2014). Dynamic binary analyzer for scanning vulnerabilities with taint analysis. *Springer US*.

J. Herrmann, Y. Zimmerman, S. Radvan, T. Richardson, P. Moore, K. Seifried, and D. Jorm (2019). Virtualization Security Guide Red Hat Enterprise Linux 7. *Red Hat, Inc*.

RAND Corporation (2019). RAND Study Examines 200 Real-World 'Zero-Day' Software Vulnerabilities. *Advantage Business Media*.

RAND Corporation (2019). Monkey In The Middle. *Penn State University*.

National Institute of Standards and Technology (2022). CVE-2022-0185 Detail. *National Institute of Standards and Technology*.

A.P. Publications Ltd (2020). 1 in 3 Fall Victim to Phishing Attacks: Lockdown Amplifies Password Security Flaws. *A.P. Publications Ltd*.

The Linux Foundation (2022). namespaces - overview of Linux namespaces. *The Linux Foundation*.

Docker, Inc. (2022). Protect the Docker daemon socket *Docker, Inc.*.

Docker, Inc. (2022). The Docker Bench for Security *Docker, Inc.*.

M. Cooney (2021). Log4j flaw needs immediate remediation: Exploits of the Log4j vulnerability can lead to loss of data on server systems and denial-of-service attacks. *Network World Inc.*.

C. Tozzi (2017) Understanding Why Docker is So Popular. *Container Journal.*.

Anjali, T. Caraza-Harter, M. M. Swift (2017) Blending Containers and Virtual Machines: A

Study of Firecracker and gVisor. *International Conferance on Virtual Execution Environments.* .

M. Talbi, P. Fariello (2021) VM Escape - QEMU Case Study. *Phrack Magazine.* .

Amazon Web Services, Inc. (2022) Firecracker. *Amazon Web Services, Inc.* .

The gVisor Authors. (2022) What is gVisor? *The gVisor Authors.* .

The Center for Internet Security. (2019) CIS NGINX Benchmark. *The Center for Internet Security.* .

Appendices

ACANARY SOURCE CODE

The source code for Canary. The full repository including install script can be found at github.com/david-wiles/canary-scala.

ArgParse.scala

```scala
import java.nio.file.Paths

/**
 * ArgParse provides a way to specify commands and arguments for Canary without any
 * extra third party packages. Parsing is simple and only full flags are supported.
 * https://stackoverflow.com/questions/2315912/best-way-to-parse-command-line-parameters
 */
class ArgParse {
  val usage: String =
    """
      |Usage: canary [OPTIONS] [COMMAND]
      |
      |A general-purpose security scanner and resolver
      |
      |Options:
      |    --local Path to local canary packages. Default location is ~/.canary
      |
      |Commands:
      |    install  Installs new packages from the list provided
      |    upgrade  Upgrades packages from the list provided
      |    check    Checks security issues using the given packages
      |
      |Run 'canary COMMAND --help' to view more information about a command
      |""".stripMargin

  private val repo: String = "https://canary.wiles.fyi"
  private var local: String = Paths.get(System.getProperty("user.home"), ".canary").toString

  def parse(args: List[String]): Option[Command] = {
    if (args.isEmpty) Option(HelpCommand(usage))
    else {
      args match {
        case Nil => Option(InvalidCommand(usage))
        case "install" :: "--help" :: tail => Option(HelpCommand(InstallCommand.usage))
        case "install" :: tail => parseInstallArgs(tail)
        case "upgrade" :: "--help" :: tail => Option(HelpCommand(UpgradeCommand.usage))
        case "upgrade" :: tail => parseUpgradeArgs(tail)
        case "check" :: "--help" :: tail => Option(HelpCommand(CheckCommand.usage))
        case "check" :: tail => parseCheckArgs(tail)
        case "remove" :: "--help" :: tail => Option(HelpCommand(RemoveCommand.usage))
        case "remove" :: tail => parseRemoveArgs(tail)
        case "--help" :: tail => Option(HelpCommand(usage))
        case string :: tail => Option(InvalidCommand("Invalid command provided: " + string))
      }
    }
  }

  /**
   * Parse args used by all commands
   *
   * @param switch the flag found
   * @param list   the rest of the command line arguments
   * @param next   function to call for the next step in parsing flags
   * @return
   */
  private def parsePersistentArgs(switch: String, list: List[String], next: List[String] =>
    ↪Option[Command]): Option[Command] = {
    switch match {
      case "--local" =>
        if (list.nonEmpty) {
```

```
60          local = list.head
61          next(list.tail)
62        } else {
63          Option(InvalidCommand("Missing local directory associated with --local"))
64        }
65      case _ => Option(InvalidCommand("Invalid option provided: " + switch))
66    }
67  }
68
69  /**
70   * Parse flags used by the install command
71   *
72   * @param args command line arguments
73   * @return
74   */
75  private def parseInstallArgs(args: List[String]): Option[Command] = {
76    var packages: List[String] = List()
77    var urlString = repo
78
79    def _parse(args: List[String]): Option[Command] = {
80      args match {
81        case Nil =>
82          if (packages.nonEmpty) {
83            Option(InstallCommand(packages, urlString, local))
84          } else {
85            Option(InvalidCommand("Must provide a package to install" + sys.props("line.separator
                ↪") + InstallCommand.usage))
86          }
87        case "--repo" :: url :: tail =>
88          urlString = url
89          _parse(tail)
90        case string :: tail =>
91          if (string(0) == '-') {
92            parsePersistentArgs(string, tail, _parse)
93          } else {
94            packages = string :: packages
95            _parse(tail)
96          }
97      }
98    }
99
100   _parse(args)
101 }
102
103 /**
104  * Parse flags used by the upgrade command
105  *
106  * @param args command line arguments
107  * @return
108  */
109 private def parseUpgradeArgs(args: List[String]): Option[Command] = {
110   var packages: List[String] = List()
111   var urlString = repo
112
113   def _parse(args: List[String]): Option[Command] = {
114     args match {
115       case Nil =>
116         if (packages.nonEmpty) {
117           Option(UpgradeCommand(packages, urlString, local))
118         } else {
119           Option(InvalidCommand("Must provide a package to upgrade" + sys.props("line.separator
                ↪") + UpgradeCommand.usage))
120         }
121       case "--repo" :: url :: tail =>
122         urlString = url
123         _parse(tail)
124       case string :: tail =>
125         if (string(0) == '-') {
```

```
126             parsePersistentArgs(string, tail, _parse)
127           } else {
128             packages = string :: packages
129             _parse(tail)
130           }
131       }
132     }
133
134     _parse(args)
135   }
136
137   /**
138    * Parse flags used by the check command
139    *
140    * @param args command line arguments
141    * @return
142    */
143   private def parseCheckArgs(args: List[String]): Option[Command] = {
144     var packages: List[String] = List()
145     var dir: Option[String] = None
146     var autoFix = false
147     var scanOnly = false
148
149     def _parse(args: List[String]): Option[Command] = {
150       args match {
151         case Nil =>
152           if (packages.isEmpty && dir.isEmpty) {
153             Option(InvalidCommand("Must provide one of packages or directory" + sys.props("line.
                  ↪separator") + CheckCommand.usage))
154           } else {
155             Option(CheckCommand(packages, dir, local, autoFix, scanOnly))
156           }
157         case "--scan-only" :: tail =>
158           scanOnly = true
159           _parse(tail)
160         case "--auto-fix" :: tail =>
161           autoFix = true
162           _parse(tail)
163         case "--directory" :: path :: tail =>
164           dir = Some(path)
165           _parse(tail)
166         case string :: tail =>
167           if (string(0) == '-') {
168             parsePersistentArgs(string, tail, _parse)
169           } else {
170             packages = string :: packages
171             _parse(tail)
172           }
173       }
174     }
175
176     _parse(args)
177   }
178
179   /**
180    * Parse arguments to remove packages
181    * @param args command line arguments
182    * @return either a RemoveCommand, InvalidCommand, or HelpCommand
183    */
184   private def parseRemoveArgs(args: List[String]): Option[Command] = {
185     var packages: List[String] = List()
186
187     def _parse(args: List[String]): Option[Command] = {
188       args match {
189         case Nil =>
190           if (packages.isEmpty)
191             Option(InvalidCommand("Must provide a package to remove" + sys.props("line.separator"
                  ↪) + RemoveCommand.usage))
```

```
          else
            Option(RemoveCommand(local, packages))
        case string :: tail =>
          if (string(0) == '-') {
            parsePersistentArgs(string, tail, _parse)
          } else {
            packages = string :: packages
            _parse(tail)
          }
      }
    }

    _parse(args)
  }
}
```

Canary.scala

```
import scala.sys.exit

object Canary {
  /**
   * Used whenever a fatal error occurs and the program should exit immediately.
   * @param message message to print prior to exit
   * @return nothing
   */
  def fatal(message: String): Nothing = {
    println(message)
    exit(1)
  }

  /**
   * Entrypoint for canary
   * @param args command line arguments
   */
  def main(args: Array[String]): Unit = {
    (new ArgParse).parse(args.toList)
      .getOrElse(InvalidCommand("Something went wrong. Please contact the maintainers of canary
          ↪and include steps to replicate the issue"))
      .run()
  }
}
```

CanaryRepository.scala

```
import org.apache.commons.compress.archivers.tar.{TarArchiveEntry, TarArchiveInputStream}
import org.apache.commons.compress.compressors.gzip.GzipCompressorInputStream
import org.apache.commons.compress.utils.IOUtils
import sttp.client3._

import java.io.{BufferedInputStream, File, FileInputStream, FileOutputStream}
import java.nio.file.{Files, Path, Paths}
import java.security.MessageDigest
import scala.io.Source

/**
 * CanaryRepository stores state and methods required to download and update
 * canary packages
 *
 * @param domain    url for the remote canary repository
 * @param localRoot local canary root
 */
class CanaryRepository(domain: String, localRoot: String) {

  private val backend = HttpURLConnectionBackend()

  /**
   * Checks whether the specified package exists in the repository
   *
```

```
   * @param name name of the package. This should correspond to a tarball in the local directory
   * @return true if package exists, false if not
   */
  def hasPackage(name: String, version: String): Boolean = {
    if (version == "latest") {
      getLatestVersionString(name) match {
        case None => false
        case Some(version) => Files.exists(Paths.get(localRoot, name, version))
      }
    } else {
      Files.exists(Paths.get(localRoot, name, version))
    }
  }

  /**
   * Verify the downloaded checksum against the tarball
   *
   * @param tarball  the tarball file
   * @param checksum the checksum file
   * @return true if checksum matches, false if not
   */
  def verifyPackage(tarball: File, checksum: File): Boolean = {
    if (!tarball.exists() || !checksum.exists()) {
      false
    } else {
      val digest = MessageDigest.getInstance("SHA-256")
        .digest(Files.readAllBytes(tarball.toPath))

      val hashBuilder = new StringBuilder()
      for (i <- 0 until digest.length) {
        hashBuilder.append(Integer.toString((digest(i) & 0xff) + 0x100, 16).substring(1))
      }

      val sumSource = Source.fromFile(checksum)
      val sum = (try sumSource.mkString finally sumSource.close()).split("\\s")

      if (sum.nonEmpty) {
        sum(0).equals(hashBuilder.toString())
      } else {
        false
      }
    }
  }

  /**
   * Get latest version of the specified package
   *
   * @param name The package to update
   */
  def updatePackage(name: String): Unit = {
    getLatestVersionString(name) match {
      case None => println(s"Could not update package $name: could not determine latest version")
      case Some(version) =>
        if (hasPackage(name, version)) {
          println(s"Updating $name...")
          downloadPackage(name, version)
        } else {
          println(s"A package named $name could not be found. Please use 'canary install $name'
              ↪instead.")
        }
    }
  }

  /**
   * Get a string representing the latest version for the specified package. The latest version
   * is stored in a file called version.txt in the package's directory on the remote server.
   *
   * @param name The name of the package
```

```
92    * @return String representing the version, e.g. 1.0.2
93    */
94   def getLatestVersionString(name: String): Option[String] = {
95     val resp = basicRequest
96       .response(asString)
97       .get(uri"$domain/$name/version.txt")
98       .send(backend)
99
100    resp.body match {
101      case Left(err) => None
102      case Right(version) => Some(version)
103    }
104  }
105
106  /**
107   * Download the specified version of a package. If no version is specified, the latest will
108   * automatically be downloaded along with the checksum for the package
109   *
110   * @param name    name of the package to download
111   * @param version version to get. Default "latest"
112   * @return error message if the package could not be downloaded or verified
113   */
114  def downloadPackage(name: String, version: String): Unit = {
115    val pkgFilename: String = if (version.equals("latest")) {
116      getLatestVersionString(name) match {
117        case None =>
118          println(s"Unable to determine latest version for package $name. Skipping package...")
119          return
120        case Some(version) => s"${name}_$version.tar.gz"
121      }
122    } else {
123      s"${name}_$version.tar.gz"
124    }
125
126    basicRequest
127      .response(asFile(new File(Paths.get(localRoot, pkgFilename).toString)))
128      .get(uri"$domain/$name/$pkgFilename")
129      .send(backend)
130      .body match {
131      case Left(err) => println("Could not download package: " + err)
132      case Right(tar) =>
133        println("Successfully downloaded package!")
134
135        // Verify checksum
136        basicRequest
137          .response(asFile(new File(Paths.get(localRoot, s"$pkgFilename.sum").toString)))
138          .get(uri"$domain/$name/$pkgFilename.sum")
139          .send(backend)
140          .body match {
141          case Left(err) => println("Could not download checksum to verify package: " + err)
142          case Right(sum) =>
143            println("Successfully downloaded checksum!")
144
145            if (!verifyPackage(tar, sum)) {
146              println(s"Checksum verification unsuccessful. Try downloading the package and
                  ↪verifying manually from $domain")
147            } else {
148              println("Checksum verification successful.")
149              println(untarPackage(localRoot, tar)
150                .getOrElse("Successfully extracted package into " + localRoot))
151            }
152
153            // Clean up files
154            tar.delete()
155            sum.delete()
156            println("Deleted archive files.")
157        }
158    }
```

```
159   }
160
161   /**
162    * Extract a downloaded package tarball into the proper folder in the same directory. If the
          ↪name of
163    * the tarball is pkg_2.1.1.tar.gz, the files will be extracted into pkg/2.1.1/...
164    *
165    * @param root the canary package root
166    * @param file the tarball
167    * @return if any exceptions are caught, they will be returned as a string
168    */
169   def untarPackage(root: String, file: File): Option[String] = {
170     try {
171       val inputStream = new TarArchiveInputStream(
172         new GzipCompressorInputStream(
173           new BufferedInputStream(
174             new FileInputStream(
175               file
176             )
177           )
178         )
179       )
180       var entry: TarArchiveEntry = inputStream.getNextTarEntry
181       while (entry != null) {
182         val outputPath = Paths.get(root, entry.getName)
183         if (entry.isDirectory) {
184           Files.createDirectories(outputPath)
185         } else {
186           Files.createDirectories(outputPath.getParent)
187           IOUtils.copy(inputStream, new FileOutputStream(outputPath.toFile))
188         }
189
190         entry = inputStream.getNextTarEntry
191       }
192
193       inputStream.close()
194
195       None
196     } catch {
197       case e: Throwable => Some(e.getMessage)
198     }
199   }
200 }
```

CheckCommand.scala

```
1 import Canary.fatal
2
3 import scala.collection.mutable.ListBuffer
4
5 /**
6  * CheckCommand gathers canary tasks and executes them
7  * @param packages  canary packages to analyze
8  * @param directory directory to analyze. Optional
9  * @param localPath local canary root
10 * @param autoFix   indicates issues found by tasks should be fixed without a prompt
11 * @param scanOnly  indicates that the user should not be prompted to fix issues. Overrides
       ↪autoFix
12 */
13 case class CheckCommand(packages: List[String],
14                         directory: Option[String],
15                         localPath: String,
16                         autoFix: Boolean,
17                         scanOnly: Boolean) extends Command {
18
19   override def run(): Unit = {
20     val tasks: ListBuffer[Task] = ListBuffer()
21
```

```
    if (packages.isEmpty && directory.isEmpty)
      fatal("Must provide one of directory or packages")

    // Gather all tasks from the directory if one was specified
    if (directory.nonEmpty) {
      val dir = directory.get
      tasks.addAll(
        TaskBuilder.fromDir(dir)
          .getOrElse({
            fatal("Fatal: " + dir + " is not a directory.")
          })
      )
    }

    // Iterate over package list and add all tasks from the package
    for (packageName <- packages) {
      tasks.addAll(
        TaskBuilder.fromPackage(localPath, packageName)
          .getOrElse({
            fatal("Fatal: " + packageName + " was not found. Please check your package install
                ↪location or install it with 'canary install " + packageName + "'")
          })
      )
    }

    // Analyze each issue and fix
    for (task <- tasks) {
      task.doTask(autoFix, scanOnly)
    }
  }
}

object CheckCommand {
  val usage: String =
    """
      |Usage: canary check [OPTIONS] [PACKAGES...]
      |
      |Checks the system against rules defined in the specified packages
      |
      |Options:
      |    --directory Custom package to use to scan system
      |    --auto-fix  Fix issues found without prompting
      |    --scan-only Scan for issues only and don't prompt to fix
      |""".stripMargin
}
```

Command.scala

```
/**
 * A Command is any action that can be run by canary. It has an associated action which
 * should be initiated by the run() method.
 */
trait Command {
  def run(): Unit
}

case class HelpCommand(output: String) extends Command {
  override def run(): Unit = println(output)
}

case class InvalidCommand(message: String) extends Command {
  override def run(): Unit = Canary.fatal(message)
}
```

InstallCommand.scala

```
import java.nio.file.{Files, Paths}

/**
```

```
 * InstallCommand is used to install packages to the local canary root
 * @param packages  list of packages to install
 * @param domain       url of remote canary repository
 * @param localPath path to local canary root
 */
case class InstallCommand(packages: List[String], domain: String, localPath: String) extends
    ↪Command {
  override def run(): Unit = {
    // Ensure canary root exists
    if (!Files.exists(Paths.get(localPath)))
      Files.createDirectories(Paths.get(localPath))

    // Download packages
    val repo = new CanaryRepository(domain, localPath)
    for (name <- packages) {
      name.split('@') match {
        case Array(pkg, version) => repo.downloadPackage(pkg, version)
        case Array(pkg) => repo.downloadPackage(pkg, "latest")
      }
    }
  }
}

object InstallCommand {
  val usage: String =
    """
      |Usage: canary install [OPTIONS] [PACKAGES...]
      |
      |Installs packages to the local system from a remote repository
      |
      |Options:
      |    --repo URL for the repository to pull from
      |""".stripMargin
}
```

RemoveCommand.scala

```
import java.io.File
import java.nio.file.Paths

case class RemoveCommand(packageRoot: String, packages: List[String]) extends Command {
  override def run(): Unit = {
    // Delete the file or directory recursively
    def deleteRecursively(result: Boolean, file: File): Boolean = {
      if (!file.isDirectory)
        !(file.exists && !file.delete) && result
      else
        file.listFiles.foldLeft(result)(deleteRecursively) && file.delete()
    }

    for (pkg <- packages) {
      val pkgPath = pkg.split("@") match {
        case Array(name, version) => Paths.get(packageRoot, name, version)
        case Array(name) => Paths.get(packageRoot, name)
      }

      if (deleteRecursively(result = true, new File(pkgPath.toString)))
        println(s"Removed package $pkg")
      else
        println(s"Unable to remove package $pkg")
    }
  }
}

object RemoveCommand {
  val usage: String =
    """
      |Usage: canary remove [OPTIONS] [PACKAGES...]
```

```
      |
      |Removes packages from the local system
      |
      |""".stripMargin
}
```

Task.scala

```
import java.nio.file.{Files, Paths}
import scala.io.Source
import scala.io.StdIn.readLine

/**
 * A Task is a single action that canary should perform during the CheckCommand run.
 * This action should at least provide a way to scan the system for some issue and
 * include a description of the issue. It may also optionally provide a way to fix
 * the issue programmatically.
 *
 * These are built from canary packages and are specified by a config file in
 * each child directory of the package. The config file can contain:
 *
 * "analyze": the path to the shell file to run to analyze the issue the task describes
 * "solution": path to the shell file to fix the issue if it exists
 * "description": a string describing the task
 */
trait Task {
  def doTask(autoFix: Boolean, skipFix: Boolean): Unit
}

object Task {
  /**
   * Create a task from the files in the directory at location
   *
   * @param location a directory inside a canary package
   * @return
   */
  def create(location: String): Task = {
    val configPath = Paths.get(location, "config")
    if (Files.exists(configPath)) {
      new ScriptTask(location)
    } else {
      EmptyTask(location)
    }
  }
}

/**
 * A properly formatted task will be created as a ScriptTask, which includes scripts to run
 * to analyze and fix issues
 *
 * @param location the filename of the task's directory
 */
class ScriptTask(location: String) extends Task {
  private var analyzeFilename: Option[String] = None
  private var solutionFilename: Option[String] = None
  private var description: String = location

  private val source = Source.fromFile(Paths.get(location, "config").toFile)
  for (line <- source.getLines())
    line.split("=", 2) match {
      case Array("analyze", value) => analyzeFilename = Some(value)
      case Array("solution", value) => solutionFilename = Some(value)
      case Array("description", value) => description = value
    }
  source.close()


  def doTask(autoFix: Boolean, skipFix: Boolean): Unit = {
```

```
    if (analyzeFilename.isEmpty) {
      println(s"Cannot perform task $location: analyze file is not specified. Skipping...")
      return
    }

    val analyzeProcess = new ProcessBuilder(Paths.get(location, analyzeFilename.get).toString)
      .start()

    if (analyzeProcess.waitFor() != 0) {
      println(Console.RED + "[FAIL] " + Console.WHITE + description)

      // Check if all conditions are met before running the solution file. The command must:
      // 1: not be run with --scan-only
      // 2: contain an executable file called solution.sh
      // 3: either be used with --auto-fix or user answered yes on prompt
      if (!skipFix && solutionFilename.isDefined && (autoFix || promptFix())) {
        println("Attempting to fix issue...")
        val fixProcess = new ProcessBuilder(Paths.get(location, solutionFilename.get).toString)
          .start()
        if (fixProcess.waitFor() != 0) {
          println(Console.MAGENTA + "Solution file exited with non-zero status. You may need to
              ↪fix this issue another way." + Console.WHITE)
        } else {
          println("Solution file exited successfully. You may want to run canary again to verify
              ↪the fix.")
        }
      }
    } else {
      println(Console.GREEN + "[PASS] " + Console.WHITE + description)
    }
  }

  private def promptFix(): Boolean = {
    while (true) {
      // Prompt user until they enter a y or n
      println("Attempt to fix this issue? [y/n]")
      val input = readLine()
      if (input.nonEmpty) {
        if (input.equals("n") || input.equals("N")) {
          return false
        } else if (input.equals("y") || input.equals("Y")) {
          return true
        }
      }
    }
    false
  }
}

// A task will require at minimum an analyze file and a description. If these can't be resolved
    ↪then
// an empty task is created to alert the user that the task is invalid and will not run.
case class EmptyTask(location: String) extends Task {
  override def doTask(autoFix: Boolean, skipFix: Boolean): Unit = {
    println(
      s"""
         |Unable to initialize task at $location. If this is a package, please contact the
             ↪package's maintainers.
         |If this is a custom scan, make sure an analyze.sh and description.txt file exist in the
             ↪ directory for this task.
         |""".stripMargin)
  }
}
```

TaskBuilder.scala

```
import java.io.File
import java.nio.file.{Files, Paths}
```

```
3 import scala.collection.mutable.ListBuffer
4
5 /**
6  * TaskBuilder provides methods to create lists of tasks for Canary from a directory or package
7  */
8 object TaskBuilder {
9   /**
10   * Creates a ListBuffer containing tasks using the given directory
11   *
12   * @param dir path to the directory
13   * @return the tasks contained in the specified directory
14   */
15   def fromDir(dir: String): Option[ListBuffer[Task]] = {
16     val directory = new File(dir)
17     if (directory.exists && directory.isDirectory) {
18       val tasks: ListBuffer[Task] = ListBuffer()
19       for (file <- directory.listFiles if file.isDirectory) {
20         tasks += Task.create(file.getAbsolutePath)
21       }
22       Some(tasks)
23     } else {
24       None
25     }
26   }
27
28   /**
29   * Creates a buffer of tasks from a package name. If no version is specified, then the
30   * latest is used. The packages are stored in the canary directory in two levels: the
31   * first level is the name of the package, and the second level is the version of the
32   * package. By listing the packages in the second directory and choosing the version
33   * with the highest number we use the latest package
34   *
35   * @param root    the canary package sources root
36   * @param pkgName name of the package to gather tasks from, formatted as pkg@version
37   * @return list of the tasks contained in the package
38   */
39   def fromPackage(root: String, pkgName: String): Option[ListBuffer[Task]] = {
40     val pkgDir = pkgName.split('@') match {
41       case Array(name, version) => Paths.get(root, name, version)
42       case Array(name) =>
43         if (Files.exists(Paths.get(root, name))) {
44           val version = latestInstalledVersion(new File(Paths.get(root, name).toString).listFiles
                 ↪.map(f => f.getName))
45           Paths.get(root, name, version)
46         } else {
47           Paths.get(root, name)
48         }
49     }
50     fromDir(pkgDir.toString)
51   }
52
53   /**
54   * Get the latest version number from a list of version numbers
55   *
56   * @param versions the list of version numbers
57   * @return string representing latest package version
58   */
59   def latestInstalledVersion(versions: Array[String]): String = {
60     var latest = Array(0, 0, 0)
61     for (version <- versions) {
62       version.split('.') match {
63         case Array(major, minor, update) =>
64           if (major.toInt > latest(0) ||
65             major.toInt == latest(0) && minor.toInt > latest(1) ||
66             major.toInt == latest(0) && minor.toInt == latest(1) && update.toInt > latest(2)) {
67             latest = Array(major.toInt, minor.toInt, update.toInt)
68           }
69         case Array(major, minor) =>
```

```
          if (major.toInt > latest(0) ||
            major.toInt == latest(0) && minor.toInt > latest(1)) {
            latest = Array(major.toInt, minor.toInt, 0)
          }
        case Array(major) =>
          if (major.toInt > latest(0)) {
            latest = Array(major.toInt, 0, 0)
          }
      }
    }
    latest
      .map(v => v.toString)
      .mkString(".")
  }
}
```

UpgradeCommand.scala

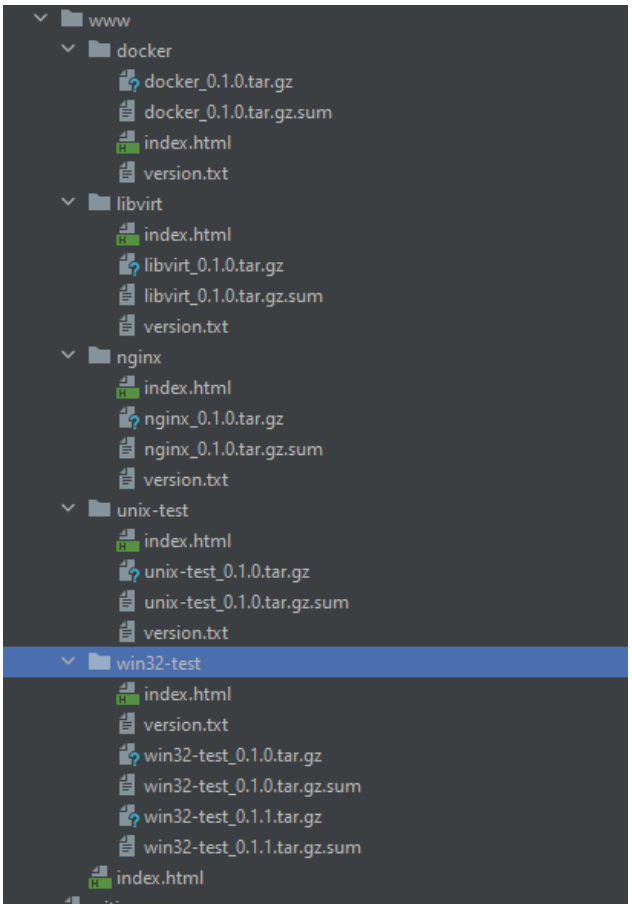
```
import java.nio.file.{Files, Paths}

/**
 * Upgrade canary packages already installed. If the package is not found it will not be
 * automatically installed to prevent unintentional actions on behalf of the user
 *
 * @param packages  list of packages to upgrade
 * @param domain       canary remote repository
 * @param localPath local canary root
 */
case class UpgradeCommand(packages: List[String], domain: String, localPath: String) extends
    ↪Command {
  override def run(): Unit = {
    val repo = new CanaryRepository(domain, localPath)
    for (name <- packages) {
      name.split('@') match {
        case Array(pkg, version) =>
          if (Files.exists(Paths.get(localPath, pkg)))
            repo.downloadPackage(pkg, version)
          else
            println(s"Package has not been installed. You can install it with 'canary install
                ↪$name'")
        case Array(pkg) =>
          if (Files.exists(Paths.get(localPath, pkg)))
            repo.downloadPackage(pkg, "latest")
          else
            println(s"Package has not been installed. You can install it with 'canary install
                ↪$name'")
      }
    }
  }
}

object UpgradeCommand {
  val usage: String =
    """
      |Usage: canary upgrade [OPTIONS] [PACKAGES...]
      |
      |Upgrades packages already installed
      |
      |Options:
      |    --repo URL for the repository to pull from
      |""".stripMargin
}
```

BCANARY REMOTE REPOSITORY

The Canary repository is currently hosted at canary.wiles.fyi. This is a file server containing all canary packages, which canary will pull from by default. Source code can be found at github.com/david-wiles/canary-remote.

Packages are stored in directories under the root as tar.gz files. These are named using the package name combined with an underscore and the package version. The website also includes index.html files to make browsing easier for web browsers.

Figure B.1: Canary repository file structure.



CDOCKER VAGRANTFILE

The Docker Vagrantfile creates a VM with Docker installed using the instructions from the Docker documentation. This configuration is the basis for the Linux machines tested in this thesis.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

$install = <<-'SCRIPT'
sudo apt-get update
sudo apt-get -y install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/
    ↪keyrings/docker-archive-keyring.gpg
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.
      ↪gpg] https://download.docker.com/linux/ubuntu \
```

```
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get -y install docker-ce docker-ce-cli containerd.io

sudo apt update
sudo apt install nginx qemu-kvm libvirt-daemon-system
SCRIPT

Vagrant.configure("2") do |config|
  config.vm.box = "bento/ubuntu-20.04"
  config.vm.boot_timeout = 600 # My computer is slow
  config.vm.provision "shell", inline: $install
  config.vm.provider "virtualbox" do |vb|
    # Enable hardware virtualization on the vm
    vb.customize ["modifyvm", :id, "--nested-hw-virt", "on"]
  end
end
```