Nix as a declarative and synchronised solution to embedded security challenges and system administration problems for multiple embedded devices

UNIVERSITY OF TURKU
Department of Computing
Master of Science (Tech) Thesis
Laboratory Name
December 2024
Eino Korte

UNIVERSITY OF TURKU
Department of Computing

EINO KORTE: Nix as a declarative and synchronised solution to embedded security challenges and system administration problems for multiple embedded devices

Master of Science (Tech) Thesis, 64 p.
Laboratory Name
December 2024

Embedded devices are an integral part of our everyday lives; household machines, automotives, and thermal sensors make use of embedded devices. They are subject to the global, developing worlds security problems. This thesis focuses on those found in public information screens. Embedded devices are particularly vulnerable to security problems as they have challenges receiving constant, reliable updates. This thesis' focal point is maintaining, updating and upgrading embedded devices in a public setting. A proposed architecture of a public media screen system is provided with example snippets to cover most of common security issues found in similar setups. The architecture with its content is then evaluated through with QuERIES methodology. The central theme of this thesis is Nix, which is a Linux distribution that forms itself from a set of configuration files, supporting features like atomic rollbacks and reliable dependency handling. Most definitive academic sources in this particular subject by Eelco Dostra are used extensively, as well as papers regarding both embedded security and measuring security. Ideas for further study are presented, as security problems may arise in our everyday lives due to the more mainstream paradigms and could be avoided with the use of declarative ones.

Keywords: tähän, lista, avainsanoista

Contents

1 Introduction **1**

1.1 Research methodologies and questions 3

1.1.1 Literature review 4

1.1.2 Research questions 6

1.1.3 Data collection and analysis 6

2 Declarative vs. Imperative systems **8**

2.1 Imperative systems 9

2.1.1 Debian/Apt 10

2.2 Declarative systems 11

2.2.1 Non-declarative components 13

2.2.2 Home manager and flakes 15

2.2.3 Ease of updates 16

3 Embedded system security **18**

3.1 Common embedded pitfalls 20

3.2 Nix as an embedded solution 21

3.3 Imperative and declarative systems from CIA-triad approach 22

4 Proposed architecture solution **25**

4.1 Server 25

4.2 Client 27

4.2.1 MQTT-client 28

4.2.2 Weston/XWayland 28

4.3 Test device 31

4.4 Installing new devices 33

5 Security standpoints 35

5.1 A brief history of security metrics 35

5.2 Choosing security metrics 36

5.3 Measuring security 37

5.4 Methodologies in comparison 39

5.5 Quantitative metrics 40

5.5.1 QuERIES 40

5.5.2 QuERIES as a central methodology 41

5.5.3 Partially observable Markov decision process 43

6 Applying QuERIES 45

6.1 Modeling the problem and quantifying the models 45

6.2 Modeling the possible attacks 46

6.3 Using the results 49

6.3.1 Improving the system using Nix 52

6.3.2 Issues with QuERIES 53

6.4 Generalizing the results 54

7 Further research 56

7.1 Further research questions 56

8 Conclusion 58

List of Figures

2.1 Terminal output from a Debian system when installing an Emacs package. 11
2.2 Relations between different user environments and installed programs [26]. 15

3.1 The CIA-triad, a way to demonstrate conflicting security measures [17]. 23

4.1 Working example of a NixOS client displaying image with Wayland and Feh. 31
4.2 Architectural graph of the test setup. 32

5.1 The QuERIES methodology is used as a reference flowchart for evaluation of security. [17] 42

6.1 Probabilities to breach into the system. 50
6.2 Cost benefits and reductions. The black dots indicate the most optimal times to stop the attack. 52

List of Tables

6.1 Different states, defensive measures, observations and attack measures for the system. 48

6.2 Protections applied in each iteration 49

1 Introduction

A Linux distribution is a bundle of the Linux kernel and a set of software products called packages [1]. A package manager is an instrument that handles building packages from either from source or pre-built binaries, resolving build-time and run-time dependencies of packages and installing, removing, and upgrading packages in user environments [1]. Every Linux device must handle its installed programs with their dependencies and configurations either imperatively or declaratively. Overwhelmingly large portion of Linux distributions fall in to the first category [2]. Both imperative and declarative distributions have their strengths and weaknesses from administrative and security standpoints.

An imperative distribution provides updatability and modification through a destructive instrument. Popular imperative package managers are apt, apk, dnf and zypper [2]. Imperative package managers can remove and overwrite existing files which leaves the system in an inconsistent state. Different installs have by nature different states which causes many problems discussed in this thesis. As files are cross-modified through packages with package managers such as apt, upgrading can be disastrous as such distributions don't support atomic rollback capabilities [1]. Due to the unpredictability, often the result can be a partially or completely broken system [2].

The reference imperative Linux distribution in this thesis is Debian with its

[1] Some Linux distributions using btrfs filesystem can perform a snapshot and rollback [3]

default package manager apt, due to it's popularity and relative simplicity. The reference declarative distribution is NixOS with it's partial namesake package manager Nix which provides declarative configuration of the whole system including the Linux kernel[2]. Nix is configured by the Nix programming language which is inspired by purely functional languages such as Haskell. [5]

The reference architecture depicted in chapter 4 is based on Nix as it is the most popular purely functional, declarative Linux distribution with over 100 000 packages [6]. Architecture in this context refers to the set containing clients and server and the used hardware. The term system means the implementation of the architecture, especially referred in chapter 6. Depending on the context, system can also mean a set of procedures, e.g a Linux system. For the term Linux distribution usually only the word distribution is used.

Another good distribution for the architecture would have been Guix, which has over 28 000 packages [1]. Guix has some improvements over Nix, including richer and more extensible programming environment with a Lisp-dialect configuration language, Scheme [7]. NixOS remains as the distribution of choice, as the number of packages is greater, and general support is found to be better.

This thesis focuses on the systems and information security of a reference architecture created with NixOS. Chapter 4 goes through a reference architecture of a solution that handles the most critical functionalities of an architecture designed for image displaying.

This thesis discusses how declarative distributions can be used as an improvement over imperative distributions. In chapter 2 both approaches to package management is compared, which is followed by an introduction to embedded security in chapter 5. In chapters 4 and 6 a quantitative research is carried out revealing strengths and weaknesses of the reference Nix environment setup. The selected methodology,

[2]There exists an experimental project that has succeeded with BSD interoperability [4]

QuERIES provide quantitative results which can be used to improve the security of declarative architectures. Propositions for further research are gone through in chapter 7 and finally, chapter 8 concludes the thesis.

1.1 Research methodologies and questions

Research methodologies used in this thesis are:

1. a literature review of central papers on subject themes found with prepared search statements
2. an action research using a laboratory setup
3. a quantitative research process using QuERIES methodology

Literature review will be addressed mostly in the next section and in chapter 3. As this thesis' main research methodology is quantitative, the gathered data points will be addressed as variables that are compared with mathematical methods. Quantitative methods are broken down in chapter 5 and 6. The central methodology is derived from QuERIES, and the information security aspect is investigated with CIA-triad. In chapter 6 section 5.4 QuERIES is compared with other metrics which are explained in subsection 1.1.2.

Quantitative methodologies are oftentimes used in conjunction with qualitative methodologies, both approaches having their strengths and weaknesses. One drawback of using qualitative methods in security framework is their inherent subjectivity. For example the Delphi technique, where a set of opinions is gathered and compared from a working group provides subjective substance for a study instead of objective perspectives [8].

The study design in this thesis is *state based*, which refers to the fact that the research methods focus on different state transitions, e.g how probable it's for an intruder to gain from partial leverage to a full control of the system. Qualitative research wouldn't alone satisfy the requirements, as investigating different state transitions without quantitative methodologies would be absurd [9].

1.1.1 Literature review

This thesis has bibliography from 49 sources, most of it gathered with a carefully prepared search statement. Other sources include manuals, material for research methods and other relevant material. The search statements results presented in next subsection, provide good base for action research and analysis.

The literature focuses on four main concepts: embedded systems security with and without declarative components, imperative systems, measuring security and Nix. The main goal is to find literature that combines these concepts to gain platform for comparing different approaches to support the action research.

Central literature revolves around Nix and multiple texts by Eelco Dostra are cited for illustrating the nuances of a Nix ecosystem. Other declarative approaches are discussed, by Endres et. al and Van der Burg [10], [11]. These approaches also contains comparison to imperative systems, which is the central approach in chapter 2. Combining cyber security with declarative approaches were discussed by Specht et. al and Kandoi and Artke [12], [13].

Discussion from Ravi et. al and Fysarakis et. al discuss embedded security extensively [14], [15]. The approach is itself too broad for this thesis' scope, so only the most fitting approaches were selected for use.

The most important literature to measure the security of a system is by Carin et al. and Hughes and Cybenko [16], [17]. The topics from these papers revolve around QuERIES, an original approach for measurably improve security.

Search statement

The main search statement for this thesis is: "embedded linux" OR ”declarative” AND (linux OR *nix) OR deployment OR ”system update” OR (compare* AND declarative AND imperative AND system*) OR security.

The search statement was prepared to provide as relevant results as possible for this thesis. The main goal was to include the hypernyms ”embedded linux”, ”linux” with other terms separated using the "OR" operator. The subterm (compare* AND declarative AND imperative AND system*) was chosen to broaden the search to include articles which compare declarative and interactive systems.

As security is a central theme in this thesis, the term "security" was included. Search was done on Google Scholar, and other useful material was handpicked, such as Nix manuals and wiki pages. Systems security and cyber security material is also included in the bibliography using search statement ”systems security OR cyber security”. Separate search ”cia-triad” and ”partial observable Markov chain” AND ”cybersecurity” were used to provide tangible meters for measuring cyber security. To further back up the research for comparing different metrics, term ”cyber security metric methodology” was searched.

For searching specific material about embedded systems, the search statement ”embedded AND security” was used. As the need for embedded toolchains was needed, statement ”yocto AND buildroot” was searched. All searches were done on Google Scholar platform.

1.1.2 Research questions

The research questions for this thesis are:

1. How can a declarative system be used to measurably improve the security of an embedded system used for displaying public media?

2. What are the advantages and/or disadvantages of such system from a system administrator standpoint?
3. How can a declarative system be updated from different Linux distribution securely and seamlessly?

Research question 1 is the most important and it traverses through themes of the whole thesis. The hypothesis is that traditional imperative embedded device fleets have problems which can be solved with the use of modern declarative systems. First, we aim to gain information from a specific scenario, presented in chapter 4, then in chapter 6 the gained information is analyzed and generalized as suitably as possible.

Research question 2 brings up the human element; how can a system administrator use a new palette of features adequately to provide more secure system and research question 3 handles a situation where existing system should be replaced with a NixOS system. How this could be done securely without risks and preferably easily with existing tooling is answered in chapter 4 section 4.4. The next chapter compares imperative approaches to declarative approaches and provides insight for understanding the central differences, the emphasis being on how declarative systems can be used to solve problems better than with imperative systems.

1.1.3 Data collection and analysis

Data collection is done with simulated red–blue team setup, where either team has a time frame where they must conduct a series of tasks. These tasks are formalized as partially observable Markov chain parameters, and analysed with QuERIES methodology. This methodology is used to gain knowledge and make the system more reliant and better with multiple iterations. Chapter 6 answers research question 1 and 2 and provides analysis for the reference system. Research question 3 is answered in chapter 4 section 4.4.

2 Declarative vs. Imperative systems

There have been different approaches to declarative modeling of systems design. Endres et al. compares declarative and imperative systems from a cloud computing standpoint, and collects systematic information on what are the strengths and weaknesses of TOSCA, IBM Bluemix, Chef, Juju, and OpenTOSCA [11]. Van der Burg and Eelco Dostra use NixOS as a solution for declaratively distributing into cloud, executing integration and system tests [10]. Most approaches researched through literature review focus on distributing to cloud. Distributing to embedded clearly remains as a niche.

Breitenbücher et al. focus on deploying into embedded and discusses the challenges an IoT user face when deploying a system [18]. It's proven that setting up devices with mandatory scripts and other actions is a challenging task, when a number of devices should be set up [18]. Cloud is something that is useful to be used in tandem with IoT but this thesis focuses on an *in-premises* reference solution.

In this chapter, we focus on comparing different declarative approaches to the more traditional imperative models, highlighting the strengths and weaknesses of both. Specifically, examples are provided to illustrate the limitations often observed in imperative systems, particularly in terms of reproducibility, scalability and administration standpoints. Cloud-oriented approaches serve as a prime reference point for how declarative systems can be effectively distributed and automated. I can be argued that similiar approaches as those taken in cloud should be taken in embedded and IoT to increase security through updatability and upgradability.

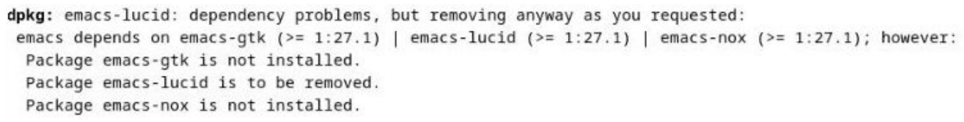
2.1 Imperative systems

Imperative deployment models base their functionalities through a process in which the order of events have a critical significance to the output [18]. In context of virtualization, imperative tooling can be used to form a all activities to be executed, the control flow, their execution order, and the data flow between them [11]. This kind of process is best to be used in conjunction with a formalized workflow or standard such as BPEL [11]. In contrast, declarative models don't have such specific requirements, as these models formalize the processes in the configuration files [11].

An imperative system provides updatability and modification through a destructive instrument. Popular imperative package managers, e.g can remove and overwrite existing files, which leaves the system in an inconsistent state [2]. Different installs have by nature different states, which causes many problems discussed in this thesis.

Imperative systems, while popular, have inherent problems regarding administrative traits contributing to a framework where the underlying system has **no traceability**: the implication that reproducibility is impossible, as changes to a system are not traced. Nix provides a solution for this problem with its Nix generation system. With imperative systems, upgrading is more error-prone than installing from scratch. This is due to the fact that imperative systems have **unpredictable** state, from where the system should migrate to a predictable state. This causes major issues regarding upgradability. [19]

The inability to run multiple configurations side-by-side is an inherent side effect of a *stateful* system. Declarative systems don't have this problem: an



```
dpkg: emacs-lucid: dependency problems, but removing anyway as you requested:
 emacs depends on emacs-gtk (>= 1:27.1) | emacs-lucid (>= 1:27.1) | emacs-nox (>= 1:27.1); however:
  Package emacs-gtk is not installed.
  Package emacs-lucid is to be removed.
  Package emacs-nox is not installed.
```

Figure 2.1: Terminal output from a Debian system when installing an Emacs package.

arbitrary number of configurations can exist side by side, as the system is defined only by the configuration, not with the state as a component. [19]

2.1.1 Debian/Apt

An example of imperative systems' problematic nature is provided with the following demonstration. Executing shell command

```
apt install emacs
```

installs a text editor wrapped as a .deb package. The package emacs has a dependency, emacs-gtk, which can be removed with command

```
apt remove emacs-gtk
```

Another dependency, emacs-lucid can be removed with command

```
apt remove emacs-lucid
```

we can see that after removing, apt automatically installs emacs-gtk to avoid breaking the application. The package manager warns: "emacs-lucid has dependency problems, but removing anyway as you requested" as shown in figure 2.1.1. It's also noteworthy, that the manual page for apt, doesn't say anything about a possible installation side-effect of a package removal command [20]. We could forcibly remove the package by invoking

```
dpkg --remove --force-depends emacs-lucid
```

thus leaving the system in an unreliable state. Dpkg is a low-level tool associated with apt, and doesn't automatically handle dependency resolutions or further package relations [21]. What happens if we had a large number of devices, in-premises or cloud where all system commands are done imperatively? We would have a large number of devices that differ from each other, because as shown, the order of commands affect the state of the system. Time is also a factor that causes systems to diverge, as packages are not up-to-date by default. Invoking

```
apt update
```

updates the local repositories to match the download mirrors. If by technical reasons or possible user error this command is conducted in the wrong order, there will be divergent systems.

Implementing a deployment model with only Debian would be a gruesome task, as the order of events which occur during the setup phase is critical. As presented by Endres et al. a formalized workflow graph would be needed to set up a reliable system. However, a Debian could be used as a host to user-space application deployment, such as Bluemix or Chef, where common DevOps practices can be used [11].

2.2 Declarative systems

Presented problems in chapter 2 can be solved using package manager that is reproducible, reliable and atomic. Package installs in Nix are in isolation from each other so that they don't have conflicting effects which results packages being predictable and assures they work coherently even if underlying install is different. As the packages are declared in a single set of configuration files, it's trivial to reproduce the system in a different environment. The demonstrated effect in snippet 2.1.1 was a problem due to lack of isolation. When dependencies are scattered in the system instead of declared explicitly in a installed package, a faulty state could be achieved.

Nix assures, that these kind of problems are out of the question. A result of this is that in a Nix system installs of same program can reside side-by-side with varying versions [2].

As presented by Endres et al, systems can be declared, even if the underlying infrastructure is imperative by nature [11]. This thesis focuses on purely functional methodologies which fix the most prevalent issues compared with imperative models. Tools such as Chef focus on deploying on a imperative system, which causes an inherent problem with cohesion in a system that should work regardless of the underlying machine or network. Alternative deployment tools are discussed in section 2.2.1.

One benefit from Nix is its lightweight tendency to enable system tests. Integrating system tests with a Debian system would require a considerable amount of work, as setting up such system needs a lot of configuration and executing commands in a correct order [22]. The mentioned distribution in the previous subsection, Debian, definitely fits in an imperative deployment strategy but the requirement for explicit detail of every step would be error prone even for a seasoned administrator [18].

It's also noteworthy that many imperative package managers don't support rollback mechanisms. If the Nix configuration file is changed and the system is rebuilt with command

```
nixos-rebuild switch
```

the previous state could be recovered by

```
nix profile rollback
```

This is an important feature as the Nix configuration files control the whole system, they can also leave the system in an undesired state. Nix switches between *profiles*, which is a way to provide different configurations for different user environments as shown in figure 2.2 and provide atomic upgrades and rollbacks. [23]

A fundamental component of the ecosystem is Nix, a domain-specific language designed for configurations distinguished by its functional nature and lazy evaluation. The concept of purity is central to Nix, where values remain unchanged throughout computation, and every function consistently yields the same output regardless of input [24]. The security implications of using Nix vs. an imperative system is discussed in the next chapter.

2.2.1 Non-declarative components

Declarative distributions such as Nix can't do everything in the system in stateless manner. Some components of the system, such as databases must have a distinct state, which can't be practically declared with package manager apart from initial configurations [5]. Home directories can vary as much as the system administrator desires. For example, a configuration file for text editor vim is usually declared in the file /home/<user>/.vimrc. Nix provides multiple ways to perform the whole configuration process from the Nix configuration files. One way is declaring the desired .vimrc in the Nix configuration, as in the following snippet:

```
{
  environment.systemPackages = [
    (pkgs.vimConfigurable.customize {
      vimrcConfig.customRC = ''
        " arbitrary vim config
      '';
    })
  ];
}
```

Nix provides also provides ways to fetch content to the system from remote URLs, and if the administrator doesn't want the system to remain "pure", they can build the system by

```
  nixos-rebuild switch --impure
```

This results the system having mutable components, which can be desirable from an accessibility point of view, but can result in unpredictable behaviour if the impure components are modified. Purity means that the components are read-only and immutable [25].

User environments (Nix profiles) can be used so that for different needs, or for different users there are multiple environments in which the user can operate as shown in figure 2.2. User environments are a successor to the concept, where installed programs either reside in

```
/usr/bin
```

or

```
/usr/sbin
```

etc. or have a symbolic link to the said directories. They can be figured as trees of symbolic links that reside also in the Nix store hence referred packages are called ”activated packages”. The installed programs reside usually in

```
  /nix/store
```

. [2]

There exists continuous build and integration services, such as Hydra, which include Nix-compatible support for handling runtime configuration and tools, such as Disnix and Charon [1], which focus on setting up complementary infrastructure. Van Der Burg presents these new tools to replace Cfengine, Puppet and Chef, which execute operations in convergent manner, meaning that they capture what changes should be done to the machines in a specified network. [5]

These approaches have two central problems: imperative nature of handling environment difference, and inability to guarantee configuration compatibility with a machine. Disnix is a Nix derivative that can overcome these challenges by separating

[1]Charon is now called NixOps [27]

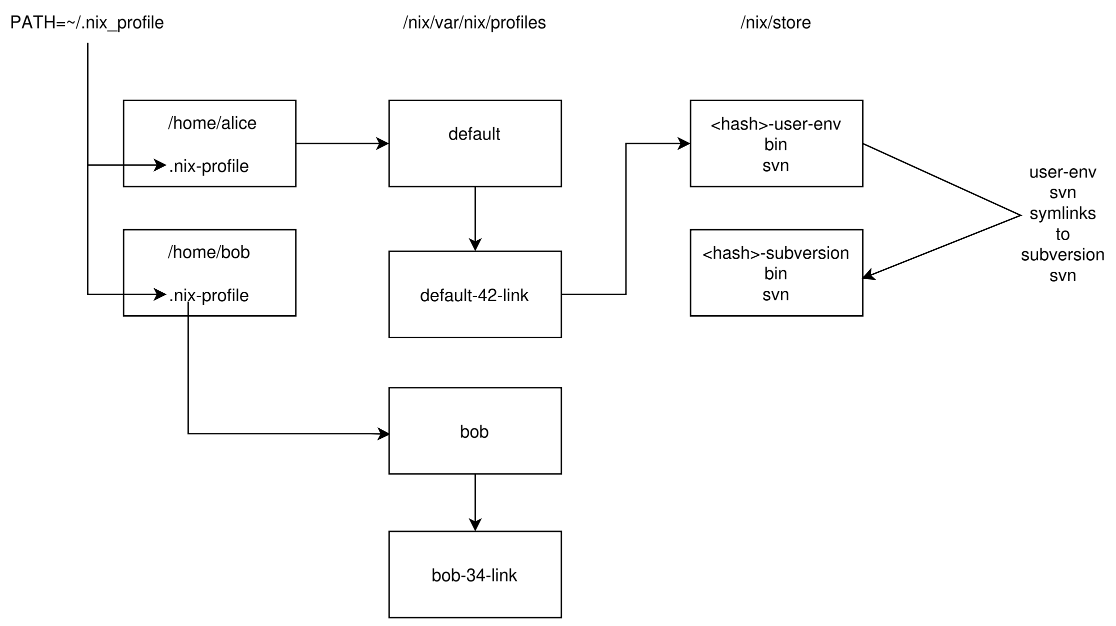


Figure 2.2: Relations between different user environments and installed programs [26].

logical properties from physical, and by capturing the essential aspects which form a system. [5]

2.2.2 Home manager and flakes

Nix environments can be build from a single configuration.nix file, but there are two significant configuration tools for managing Nix systems: home manager and flakes. Home manager is an extension for managing user profiles with a declarative Nix syntax [28]. Home manager has problems with atomic rollbacks and for this reason they are not used in this thesis' examples [28].

Flakes are experimental feature of Nix, providing environments, where dependencies are pinned in a lock file, further improving reproducibility of Nix systems. A flake is a file system tree, which contains a root directory with the Nix file specification called flake.nix. The usage of flakes is a good method for organising different environments within a Nix system, where it can consist of multiple flakes. Flakes are an experimental feature, thus out of this thesis' scope. [29]

2.2.3 Ease of updates

Updating is easy and riskless with NixOS due to atomic rollbacks. Nix handles software providing through something called channels. A channel is a set of latest Git commits in a Nixpkgs repository, where they are divided to stable/unstable and large/small. channels. Unstable channels (large and small) have the latest commits on a rolling basis, but include less conservatively checked functionalities. Stable channels are submitted through a version number (e.g. 23.11), where a new release is published every six months. Large channels contain a full set of Nixpkgs binaries, when small include a subset. If a system administrator decides to submit to small channel, they have more recent updates at their disposal, but have to resort to compiling some needed packages from source. [30]

Updating a Nix system is just a manner of invoking command

```
sudo nix-channel --update
```

and, if stable release is chosen, updating the system.stateVersion from the configuration.nix file [23]. Nixpkgs is a repository of working Nix packages using a continuous integration service called Hydra. Hydra evaluates the needed Nix expression of a package, and ensures its functionality. [23]

It's apparent that purely functional, declarative approaches aren't as popular as imperative systems due to the need of steep learning curves and obscure syntax. This can be seen as historical payload: imperative models have been in use a much longer time than purely functional approaches, such as Nix.

In this chapter we revealed, that declarative systems have inherent strengths in both deployment strategies and system upgrading. The next section brings up the security viewpoint of declarative systems specifically in an embedded Linux setting.

3 Embedded system security

In the previous chapter, the core differences between imperative and declarative approaches were gone through. This chapter first discusses the embedded security field in general, following up on how Nix can be used to improve embedded security, reflecting the CIA-triad in section 3.2.

According to Serpanos et al. the use of embedded devices can be divided into four fields: industrial systems, nomadic environments, private spaces and public infrastructure [31]. This thesis' focus is public infrastructure, specifically information screens in a public environment. Implementing security mechanisms and policies is essential for information screens to function securely from both organizational and technical viewpoints. Implementing those policies and assuring compliance is trivial with declarative approaches with increased benefits from reliability perspective.

Embedded systems are distinct from other type of systems due to their varying nature ranging from programmable logic controllers (PLC) to larger systems, such as servers or routers [15]. An usual embedded device conducts a specific task and possibly demands networking capabilities. Working with embedded is typically working with a limited set of resources, demanding careful design when a multitude of features are needed. Even services such as SSH have had history of vulnerabilities, which prove that upgradability is a fundamental base of a secure system [32]. It can be argued that the security aspect of embedded devices could be improved significantly with the use of declarative systems as seen in chapter 2 section 2.2 and in the section 3.2.

Embedded devices demand precision and security, as their function may be very critical for variety of safety reasons, e.g in automotive industry or healthcare applications [15], [33]. Reliability is a defining requirement for number of embedded applications; a pacemaker that doesn't function all the time reliably is completely useless. While a declarative solution itself can't fulfill all security needs, it definitely could improve the *reliability* of such systems.

As stated by Fysarakis et al. implementing access control is essential for any system to prevent unauthorized access [15]. Defining complex access control is trivial with Nix, as the configuration files denote completely which user has accesses to which resources. Access control in a modern day embedded environment could be hard to implement in traditional imperative systems, as scaling such system that spans multiple devices and changing environments would require a lot of manual intervention. This is definitely one of strengths of declarative approaches: scalability is never an issue when a centralized configuration defines the systems. A declarative approach is often taken in the cloud as stated in section chapter 2 section 2.2, but implementing declarative overlays definitely needs work in the embedded field.

Implementing policies information security perspectives using a policy modeling standard, CIA-triad is discussed in section 3.3, and Nix is reflected with the use of the triads axes. Dolstra states that Nix is policy-free meaning that it contains a set of mechanisms which allow policies to be constructed with and not the other way around [34].

Embedded being a broad field, in this thesis devices are limited to those which can run Linux kernel and provide the most basic networking capabilities. These cover architectures i686, x86_64, arm64 supported by NixOS. PLCs and microcontrollers are outside of scope as NixOS needs a functional Linux kernel and a specific architecture to work.

3.1 Common embedded pitfalls

Common issues regarding embedded devices are their lack of updates, weak data integrity, and the multitude of features [15], [35]. For example, a toy teddy bear may have a audio recorder, data transfer capabilities and ability to geolocate itself. These kind of devices may lack firmware or software updates, and the data-transfer may be insecure.

A solution for secure data transfer would be TLS-encrypted messaging between clients. This could be achieved with MQTT-protocol, but configuring certificates is extra effort. Multitude of features is a definite security problem, as the user may not be aware of them at all times. In an increasing global world, importing embedded devices from unreliable sources can prove to be a security issue. The household items may or may not adhere to latest security compliance. [15]

Attack surface of embedded systems in general range from physical access to network and geolocation problems. One way of manipulating a device, apart from directly gaining access to the operating system, are side channel attacks. Analyzing the power or electromagnetic properties of device input/output can be used to determine critical aspects of a device, e.g key lengths or algorithms of security measures [15], [31]. Attack surface may used to gain access, or performing denial of service attacks. Geolocating is both a privacy and security issue, as location data may be used to trace identities of device users, which can lead to e.g blackmailing, physical intrusion or other means [15]. This means that the principals of this thesis' reference architecture could theoretically be targeted with such malicious intents.

Embedded systems have problems regarding monitoring and system administration. It's very different to have home automation system with less than 20 nodes, than to have public transport embedded fleet in a big city with 2000 nodes. As the number of devices grow, so does the challenge of monitoring and administrative tasks. Home automation has usually one person dedicated to the task: the home owner. Monitoring should be trivial to automatize (e.g by using tools like Prometheus), but administrative tasks are harder to automatize, due to tasks being potentially very challenging even for dedicated system administrators. This is where Nix comes to play, as updating any number of devices becomes trivial.

3.2 Nix as an embedded solution

Declarative systems have advantages over imperative systems in reliability and safety aspects due to two facts: rollout and rollback are equally trivial tasks and desired configuration can be tested in a sandbox environment. It's very accessible to manage a rollout strategy, as the rollout/rollback can be done multiple times or executed completely in a replicated sandbox environment. Simpler and more straight forward practical steps give space for easier strategical planning [13].

Kandoi et. al argue that with declarative systems, it should be nearly impossible to misconfigure in the first place the system and if faulty state is achived, a simple rollback could undo the changes [13]. As stated in chapter 2 section 2.2, it's definitely possible to achieve faulty systems with Nix. I argue that these problems can be mitigated with a well thought rollout/rollback strategy.

Updatability is possible with many different platforms, but it's a problem when updating is a sole duty of a consumer, who may or may not have the adequate knowledge how or why they should update their systems. Lightweight updatability comes out-of-the-box with Nix, and that is something that inherently should make it more secure. Consumer products, however are out of this thesis' scope.

Nix is a double edged sword for system administration tasks. On one hand, it has a steep learning curve, but on the other hand it can make tasks that could be very challenging with traditional systems trivial. In a well built Nix ecosystem security actions such as updating or modifying user or kernel space can be used to enhance security and in such system, any changes could easily be replicated to multiple devices, without the need for manual intervention.

Some other clear disadvantages for Nix in embedded use is the fact that a purely functional, declarative system inherently must use disk space more than it's imperative counterparts. In the worst case scenario, if one derivation of a system takes up 1Gb of space, when making changes, the resulting system will need 2Gbs of space [19]. The worst case scenario rarely occurs, but due to Nix's indestructive nature, this formula of disk space demands has to be considered in an embedded setting.

3.3 Imperative and declarative systems from CIA-triad approach

CIA-triad can be used as a tool to show conflicts between different points of information security interests. It consists of three meters: confidentiality, integrity and availability as seen in the figure 3.1. Confidentiality can be seen as superset of privacy. Confidential data is classified with technologies such as data encryption and user privileges [36]. Integrity means that the data has not been tampered with, and remains untouched by unauthorised parties while it's in transit or stored e.g in a server [36]. A way of providing integrity is checking hashes of downloaded files. Availability is a user viewpoint to the accessibility of the system [36]. When confidentiality and integrity are pushed to the extreme, availability aspect suffers, e.g when a service enforces multi-factor authentication.

Systems with an imperative package manager are more accessible than declarative systems as learning a new programming language with esoteric paradigm can pose extra effort. Configuring a whole Nix system demands a thorough knowledge of Nix language, and that definitely hinders the ease of access to a Nix system from a system administrator standpoint. With NixOS, an easy extent of accessibility can be achieved via planting sufficient configuration files during device setup.

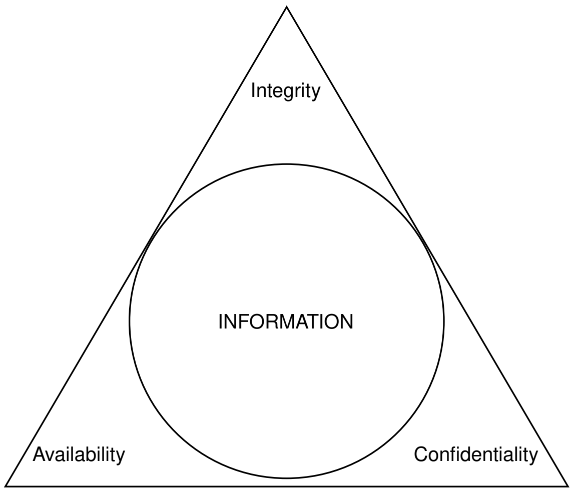


Figure 3.1: The CIA-triad, a way to demonstrate conflicting security measures [17].

Atomic systems such as Nix have great benefits towards integrity. As the Nix store, where every installation is located is read-only, it's impossible for attackers to modify the store. That's not the case in traditional approaches, where a user with root privileges can arbitrarily modify installed programs and files.

According to Nix manual, it has three main strengths in relation with security. **Security by obscurity:** combined with the unusual file system and the usage of user-environments, some malware that rely on the usual locations of installed programs may fail [37]. This is a very thin layer of security, as targeted malware has usually no problem navigating in an unfamiliar environment. **Multi-user installations:** the requirement for root access nearly always widen the potential attack surface. Nix provides a way for multiple users installing their programs through the use of user environments, hence mitigating the need for root access. This both lessens the availability aspect, as well as mitigates the programs root access. When file changes are made in user-specific scope, a thin layer of isolation is achieved [23]. **Data integrity** is achieved both by installed programs residing in the read-only Nix store, but also them having been checked against SHA256 checksums. Moreover, the core installation resources for NixOS are GPG-signed by an administrative Nix team [37].

It's apparent that Nix can be used to improve embedded device security with some hardware constraints. A purely functional declarative solution can improve the rollout/rollback capabilities, testing pipelines and provides features such as read-only Nix store. The next chapter proposes an architecture solution which works as an example of a Nix ecosystem. The main purpose of the architecture solutions is to demonstrate the security implications of declarative systems, provided in chapter 6.

4 Proposed architecture solution

For gaining data points for the research, a test setup must be configured. This chapter presents a Nix ecosystem, which goes through a red team testing in chapter 6. The proposed architecture solution uses a completely declarative approach introduced in section, where pros and cons of both approaches are discussed 2.2.

The proposed architecture is a referential framework for a Nix ecosystem and provide only a basis for a subset of features needed in such system. For example, remote access and tunneling would require extra work in a production setting. Note that presented solution's scalability could be improved with the use of tools referred in section 2.2.1.

The main purpose of the architecture is to construct a system that focuses on reliability and fluid deployment tasks. Kandoi and Hartke discuss the operating large scale IoT solutions through declarative configuration APIs and conclude that it can solve multitude of scalability and extensibility challenges of IoT systems thus ensure reliable and safe operations [13]. As embedded security is the focal point of this thesis, a further analysis is presented in chapter 6.

4.1 Server

Van der Burg et al. provide a reference architecture with OpenSSH, Quake 3 server and Transmission services [5]. Dolstra, Vermaas and Levy build a declarative system with simple web server [24]. This thesis' architecture is moderately complex in comparison with these approaches, as it contains multiple components and a functioning server–client implementation.

A proposed architecture can be viewed in figure 4.2. In simplicity, the NixOS server runs a MQTT-broker for publishing images encoded in base64 format for the client devices. The clients are called as device fleet, and the server is called central server. The client devices submit to configured topics, and display them using a Wayland compositor, Weston.

The central server has two main functions: receiving SSH-connections for admin usage and forwarding bitmaps formatted in base64 to the clients. In production environment, the server would be the element that is in a public network, and the devices would be accessible locally (through a tunnel). SSH-connections to the machine would then be forwarded through the central server to the clients. The central server currently doesn't generate the imagery, but this could be achieved via headless browser, or other graphical tools.

MQTT is a extremely lightweight, machine-to-machine publish/subscribe protocol which can be used on virtually every platform including microcontrollers [38]. The chosen MQTT broker and client for this project is Mosquitto. The following snippet shows how the Mosquitto server is configured in the NixOS server.

```
services.mosquitto = {
  enable = true;
  listeners = [
    {
      users.<user> = { acl = [ "readwrite #" ]; }; settings = {
        cafile = "<ssl-path>/ca.crt";
        certfile = "<ssl-path>/myhostname.crt";
        keyfile = "<ssl-path>/myhostname.key";
      };
    }
  ];
  };
```

The ”services” statement tells us that a SystemD service is being defined. The settings section specify which certificate and key files are to be loaded to the service.

The MQTT-broker (Mosquitto in this case) publishes a message that is forwarded to the subscribing clients via a following script.

```
nix shell nixpkgs#mosquitto --command mosquitto_pub -h localhost -t
images/test -m "$IMG_BASE64"
```

Sending could be automatized with a service using SystemD timer:

```
systemd.timers.publish-image = {
  timerConfig = {
    OnCalendar = "*-*-* *:*:00";
  };
  wantedBy = [ "timers.target" ];
};
```

which invokes a specified service.

4.2 Client

Some IoT solutions prefer client’s such relationship with client and server, where the client automatically searches for suitable server dynamically [13]. This thesis’ client structure is static meaning it follows static addresses and forming initial connections requires manual intervention.

Both server and client are running NixOS. The client has two main functions: subscribing to media receiving and displaying the gained media which can be arbitrary. Currently, the image refreshes every second and through configuration, even displaying animations with this setup could be possible. Media display happens with feh, an image showing tool, that works with X server. However, this example is using Wayland, so compability layer XWayland must be used [39]. Image data messaging functions through MQTT-protocol, which is explained more in the next subsection.

4.2.1 MQTT-client

The MQTT client subscribes to a topic from the following script. The image is received as base64 string, and is converted back to PNG format.

```
IP="<server ip>" TOPIC="images/test" nix shell nixpkgs#mosquitto
--command mosquitto_sub -h $IP -t $TOPIC >" <image directory path>
/image.base64" base64 -d "<image directory path>/image.base64"
>images/latest.png
```

4.2.2 Weston/XWayland

Wayland is a display protocol aiming to replace partially or fully the old X window system. Wayland functions thorugh a "compositor" (server), and that provides a surface for the device to draw graphics. Wayland was selected for this project due to increased security, as the X window system has support for network transparency which broadens the attack surface. Wayland has combined server and client rendering with the Wayland compositor, so that safety-critical throughput between display server and window manager is not a concern. [39]

The example project displays an image from a directory via script:

```
sleep 5 && /nix/store/qc9j6pm6ykyx531s4kb06084mczy2l6g-feh
    -3.10.1 /bin/feh -F -Z -R 1 <image-path>/latest.png
```

As the programs must be found from an absolute path, the system must generate the scripts accordingly. This is done via a SystemD service, specified as:

```
{ config, pkgs, ... }: let
  # Create the feh.sh script to launch feh
  fehLaunch = pkgs.writeText "feh.sh" ''
    echo "sleep 5 && ${pkgs.feh}/bin/feh -F -Z -R 1 <image
       directory>/latest.png" > /home/user/abzug-receiver/weston/
       img.sh
  '';

  # Create the initImg.sh script to compile the C code
  initImg = pkgs.writeText "initImg.sh" ''
    echo "${pkgs.gcc}/bin/gcc <source path> img.c -o <binary path>/
       img" > /home/user/abzug-receiver/weston/init.sh
  '';
in {
  systemd.services."westonl" = {
    enable = true;
    unitConfig = {
      Type = "oneshot";
    };
    serviceConfig = {
      # Set environment variables
      Environment = "XDG_RUNTIME_DIR=/var/run/user/1000";

      ExecStartPre = [
        "${pkgs.bash}/bin/bash ${initImg}"
        "${pkgs.bash}/bin/bash <init.sh path>/init.sh"
        "${pkgs.bash}/bin/bash ${fehLaunch}"
        "${pkgs.bash}/bin/bash <init.sh path>/init.sh"
      ];

      ExecStart = "${pkgs.weston}/bin/weston --config=<weston
         configuration directory>/weston.ini";

      RestartOn = "failure";
    };

    wantedBy = [ "graphical-session.target" ];
  [core] idle-time=0 xwayland=true [shell] panel-location=""
  panel-position=none** [autolaunch] path=<feh launcher path>/img
```

This wouldn't be a problem in traditional Linux distribution, but in NixOS the program locations vary from machine to machine [25]. A program resides in Nix store, with a cryptographic hash of all build inputs in its directory path. [25]. For that reason, one way of proceeding is to write a SystemD service to generate the configuration files. Note that this SystemD configuration differs in how SystemD scripts are declared usually. Most SystemD distributions have SystemD files in /lib/systemd/system directly or via a symbolic link.

In the beginning of the configuration, variables are defined and program locations are expanded from Nix package paths. Then, in the "serviceconfig" part of the configuration, the strings are forwarded to shell, which in part compiles sources and executes scripts. After the "ExecStartPre" section, in the "ExecStart", Weston is launched with very basic kiosk configuration:

the "[autolaunch]" only functions with compiled binaries thus the shell script is not directly executed. Instead, a program written in C is invoked, which in part invokes the shell script with parameters. The autolaunch path can handle switches, but unfortunately not parameters. With these workarounds the kiosk successfully can display media as shown in figure 4.1. Feh is launched with -R 1 parameter, which causes it to refresh the image every second. That way when a new image is uploaded, the display is also refreshed.

The sources and scripts are downloaded from Github, again via a SystemD service. Following snippet shows the "ExecStart" part of the service.

```
ExecStart="${pkgs.git}/bin/git clone <git url> <installation path
    >";
```

This part of the configuration is not purely functional, as the downloaded scripts and configuration can be arbitrarily changed with correct permissions. The SystemD

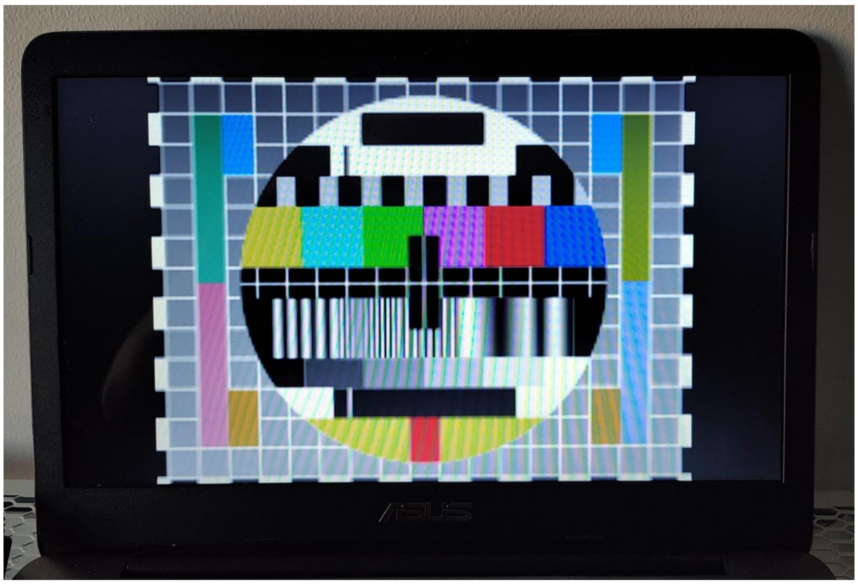


Figure 4.1: Working example of a NixOS client displaying image with Wayland and Feh.

services impurity don't trigger the Nix language evaluation itself, so ”–impure” switch isn't mandatory.

4.3 Test device

A huge benefit from declarative systems is their broad possibilities of automated system tests [22]. The phrase: "if it works on one machine, it will work on another" builds a stable foundation for such tests [23]. Different deployment strategies benefit from slightly different approaches, but as deployment strategies generally are out of this thesis' scope, only a minimal test setup is configured.

NixOS devices can be upgraded and updated through a server or locally with physical access [23]. In this thesis' example, a separate test machine is configured, and it's programs are replicated to other devices as seen on figure 4.2. Updates are done manually with command:

```
nix build --eval-store auto --store ssh-ng://remote-host
```

The test device is thus replicated to all remote hosts manually, but an automation

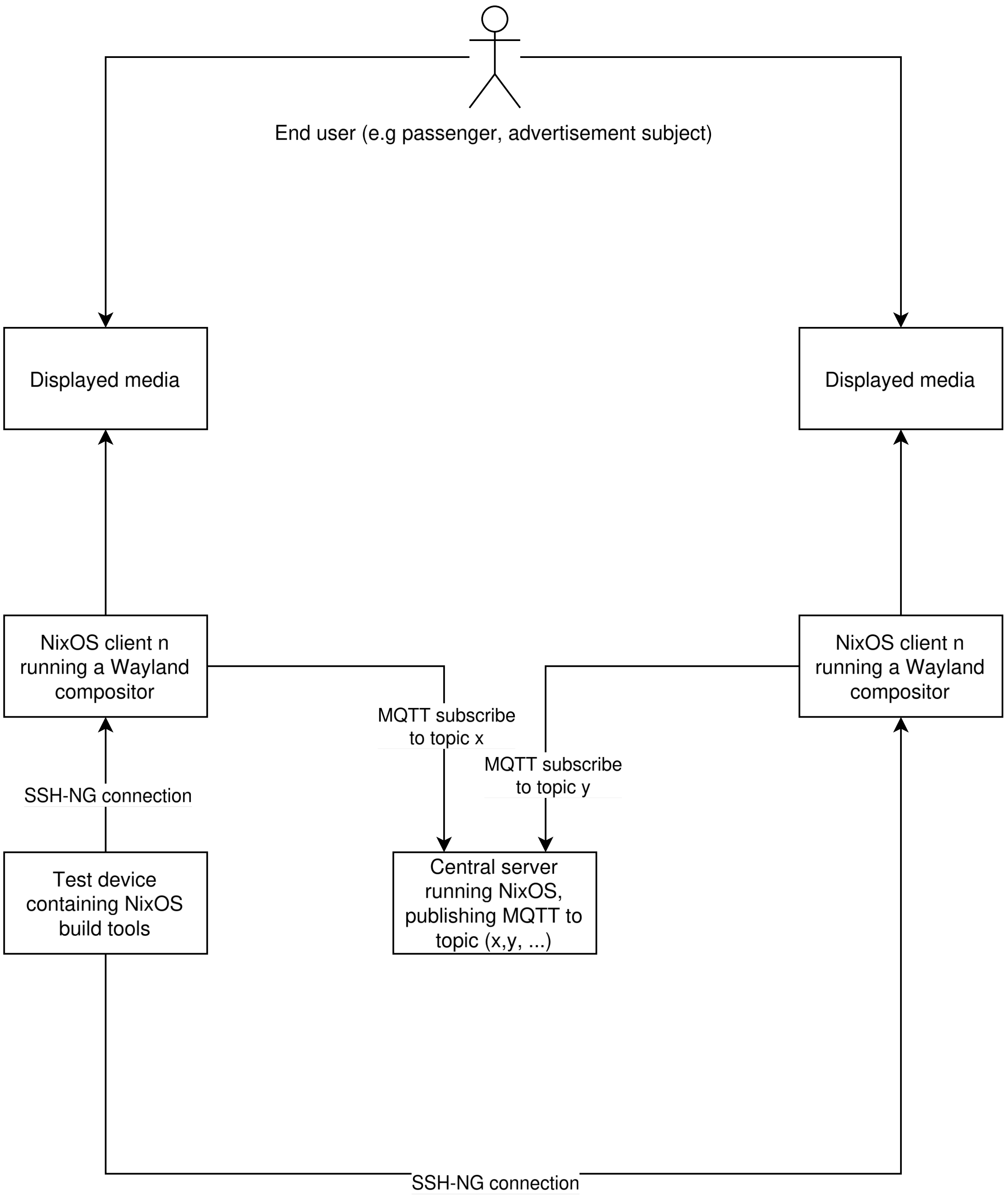

End user (e.g passenger, advertisement subject)
Displayed media
Displayed media
NixOS client n running a Wayland compositor
NixOS client n running a Wayland compositor
MQTT subscribe to topic x
MQTT subscribe to topic y
SSH-NG connection
Test device containing NixOS build tools
Central server running NixOS, publishing MQTT to topic (x,y, ...)
SSH-NG connection

Figure 4.2: Architectural graph of the test setup.

script would be useful as the setup scales.

4.4 Installing new devices

Installing devices could made trivial, as it's trivial to generate new NixOS images from pre-existing configuration files [23]. The problem is, however, that in many cases a legacy architexture exists, which needs to be overwritten. This section focuses on research question 3, providing a solution for migrating from existing architectures.

If the whole setup needs to be replicated to an existing device fleet, a great help would be open source project "NixOS anywhere". NixOS anywhere specifies minimal specifications: "Unless you're using the option to boot from a NixOS installer image, or providing your own kexec image, it must be running x86_64 Linux with kexec support. Fortunately, most modern x86_64 Linux systems have kexec support. By providing your own image you can also perform kexec for other architectures e.g aarch64". NixOS anywhere also has a requirement that the devices should be available at a public network (not only wireless LAN), which wouldn't probably be a problem in production environments. NixOS anywhere also works only with NixOS flakes, so this thesis' configuration would need to be contained in a flake. [40] The installation would contain the following steps:

1. Run "curl -L https://github.com/nix-community/nixos-images/releases/download/nixos-unstable/nixos-kexec-installer-noninteractive-x86_64-linux.tar.gz | tar -xzf- -C /root /root/kexec/run" for booting separate kernel for installation.

2. Generate a minimal configuration file: "nixos-generate-config –no-filesystems –root /mnt"

3. Add ssh-keys to the configuration

4. Upload the flake from the server to the device "nix run github:nix-community/nixos-anywhere – –flake <path to configuration><configuration name> root@<ip address>"

Another way would be setting up NixOS from the installation media and setting up manually or creating a installation media with Nixos generators project (https://github.com/nix-community/nixos-generators). Nixos generators is available from Nixpkgs, and can thus be installed to a NixOS development machine or server. Invoking

```
nixos-generate -f iso -c <configuration.nix path>
```

would result in a ISO format image, ready for installation. [40]

The next chapter provides insight on how to measure security as a whole, transferring the necessary metric systems for this thesis' purposes. The architecture presented in this chapter then is gone through further analysis in chapter 6.

5 Security standpoints

Security is inherently challenging to measure adequately, due to its complex and chaotic nature. Qualitative analysis may also result in subjective output. As such, no unambiguous standard of measuring security can be provided [8]. To overcome these challenges, several metric systems are compared, and the one that provides the most precise answers specifically for this thesis' cause is chosen.

In this chapter, selection process for meters, which in part are used for gaining quantified data, is presented. When finally quantitative metrics are gained with the help of two paradigms, GQM (goal, question, metric) and specifically its superset SMART, a red–blue team layout is erected using QuERIES methodology. SMART is used in conjunction with the literature review to reveal the most suitable methodology for this thesis. SMART is opened more in the section 5.2.

Using an exact metric system is important, as this this thesis' emphasis is on measuring the security of a declarative approach. The presented architecture in the previous chapter works as a example on how a declarative system can be improved by reconfiguration, utilizing a red-blue team setup.

5.1 A brief history of security metrics

The history of security metrics begin from Trusted Computer System Evaluation Criteria (TCSEC), also known as the Orange Book from 1983 which popularized many terms still in use today, such as identification, authentication and authorization. [41]

When US National Bureau of Standards (NBS), the organization later formed as the National Institute of Standards and Technology (NIST) tried to standardize security, in the 1990s it became evident that a system should adhere too strongly to the definitions of the Orange Book and the follow up project, Common Criteria to be used accordingly. Later, multiple standards became popularized, e.g System Security Engineering Capability Maturing Model (SSE-CMM), which can be used as a sort of checklist from system design from the ground up. [41]

Later it was observed that systems design is only a part of a successful security strategy, and operational practices played a bigger role than expected, something to be addressed in 1995 NIST Computer Security Handbook, which has evolved to provide ground to combat modern issues. [41]

As metrics can be used as a tool for decision making, the strategical approach of the mentioned publishes is important. It's noteworthy that the strategies (the Orange Book, SSE-CMM, etc.) begin to measure security by compliance to defined ratings. Later in 2000s more mathematical approaches were taken, one which is delved deeper in section 5.4. [41]

5.2 Choosing security metrics

Security is something that is challenging to measure due to it's complex nature. A GQM (Goal, question, metric) paradigm helps to choose appropriate metrics: first there must be a set goal to a organisation, then a formulated question for each goal. These answers are then reflected to gain the desired metric. This strategical approach is perhaps too broad for this thesis' scope, but aligns well with an usual organisational strategy. [42]

A more appropriate tool for this task would be SMART – a set of inputs to evaluate meter systems' suitability. These inputs describe how specific, measurable, attainable, relevant and timely the methodology is [43].

In cyber security, being specific is very important and a common issue with security meters is that they either cover too many topics and are without precise definitions, or they are too specific to be generalized to broader scope of situations [8]. This thesis' results are important to be measurable, as the research orbits around system states, and the research aims to measure with what outputs do the state transitions resolve to.

To be attainable is relevant to this context for the reason that a thesis has different scope than a big organization, and the proposed setup has to go through a check on the metrics' goals achievabilities. Relevance has to do with risk assessment: how important it's to measure something related to it's value. Risk assessment is gone through thoroughly in 6 section 6.1. Time-bounding signifies the importance of time as a meter; a system that can be penetrated in a minute can definitely be seen as weaker than a system that takes years to be compromised.

5.3 Measuring security

In this section, different methodologies and perspectives gained through literature review for cybersecurity are discussed, and potential methodologies are compared to gain the most adequate metric system for usage through SMART process [43].

Security metrics can be divided to address four separate themes: **System vulnerabilities**; measuring vulnerabilities can be applied to user, interface-induced, password, and software vulnerabilities. Users are always susceptible to e.g phishing attacks or malware infection, where a user of an arbitrary system is the definitive attack vector. Interface-induced vulnerabilities refer to attack vectors related to open ports and endpoints. [44]

Password vulnerabilities refer to situations where password can be computationally cracked. This is relatively simple to measure, as it can be estimated how much time it takes to crack a password, or with the use of statistical password guessability. Software vulnerability on the other hand is a very usual way for a breach to take place. This kind of vulnerabilities can be measured, thus also estimated with the help of exploitations in the past. The essential metric is the time to patch a software vulnerability. [44]

Defense measures can be applied to strength of reactive, preventive proactive and overall defenses. Reactive measures include blacklisting, a lightweight mechanism to prevent e.g a botnet to harm the protected system by blacklisting IP-addresses related to the botnet. For measuring defence, the reaction time is essential, and most importantly, the gained meter to measure preventive defense. Blacklisting can also be used as preventive and proactive measure, as a pre-filled blacklist can be used with desired parameters. [9], [44]

Overall defenses can be measured with the combination of all defensive measures and with the use of penetration testing in a red–blue team setup. Penetration testing aims to gain a result, also known as penetration resistance, which is a meter, indicating cost or time that the red team must spend in case of a successful system compromisal. [9], [44]

Threats: zero-day vulnerabilities can be measured from two perspectives: lifetime of zero-day vulnerability and the number of nodes that are compromised as a result. Malware spreading can be traced with the parameter infection rate, which is defined as infected node per a time unit. Attack evasion is measured using either obfuscation prevalence metric, or structural complexity metric which provide information on obfuscating gained samples e.g by encrypting, or the target system's complexity measured by runtime. [9], [44]

Situations, which can relate to security state, security incidents and security investments. Security state has multiple parameters, including incident rate and blocking rate. Security investments on the other hand measure the budget percentage funneled towards security, and the return of such investment. [44]

5.4 Methodologies in comparison

Today, there exists cybersecurity metrics based on quantified mathematical models, which are prevalent for this thesis. Three different methodologies are discussed, and one is picked for measuring the security of this thesis' architecture implementation. All the following metric systems are **measurable**, but some fit better especially according to **time-related** and **relevance** axes.

The literature review provided three central methodologies from variably different perspectives. Complex mathematical models are presented by Alshammari et al. are too broad [45]. This thesis' scope is limited, and this methodology would fit better a wider cyber security setting.

A methodology based on object-oriented thinking, followed by UML-graphs is adequate in many contexts, but as stated in the paper this measurement the methodology is used to compare similar alternative designs. [45]. As it has been stated, this thesis focus isn't comparative, as all the critical comparison has already done in chapter 2.

Hidden Markov models presented by Wang et al. are close what is the end goal of the research is in this thesis [46]. The **time-related** aspect would be satisfactory, as the hidden Markov model handles a time parameter. The problem, however is regarding the generality of the methodology, and something more **specific** would be a better fit for this thesis. **Relevance** also is an issue, as using presented Hidden Markov models would be mathematically challenging, and perhaps too complex for the scope of this thesis.

The last and the most fitting methodology would be one presented in papers by Carin et al and Hughes, Jeff and Cybenko [16], [17]. The QuERIES methodology is delved deeper in section 5.1, and its **time-related**, **relevance** and **specific** axes are a near-perfect match for our goals as the model itself is relatively simple and provides shifting probabilities from states, which serves this thesis' study design well.

5.5 Quantitative metrics

Selecting carefully a metrics system includes asserting our goals and questions. Our goal is to discover this thesis' architecture proposals tenacity in a simulated setting. The main goals reside in this thesis' two research questions:

- How can a declarative system be used to measurably improve the basic security needs of an embedded system used for displaying public media?
- What are the advantages and/or disadvantages of such system from system administrator standpoint?

The proposed architecture solution presented in chapter 4 will go through a red and blue team inspection, complying with the QuERIES model in chapter 6.

5.5.1 QuERIES

QuERIES model consists of number of steps that

1. model the problem - by conducting a risk assessment of the attack surface and the value of the possible intrusion
2. model the possible attacks - build an attack graph of intruding though vulnerabilities or other means
3. quantify the models - by conducting a controlled red team attack and provide quantified results for the said attack

4. use the results - use blue team methodologies to provide increased protection against the exposed problems

First, risks are assessed of the attack surface due as a blue team task. It's very important for blue team to know what are the most critical points of the attack surface, and it's also used as the base for quantitative analysis. Value of the intrusion can also be used for the reward model for analysis. As seen in figure 5.1, the methodology is applied *iteratively*, i.e the steps are repeated as many times as needed for the system to be secure.

Modeling the possible attacks is a task for the red team – by constructing an attack graph, the opposing forces have a plan, which can be used as a template for analysis. In this thesis, the models are quantified with the use of time framing. Both teams have limited amount of time to conduct their tasks, and probability for succeeding a certain task is calculated with formula

$$\frac{t_e}{t_t}$$

where t_e stands for elapsed time and t_t for maximum time that can be used which is the same for all tasks.

5.5.2 QuERIES as a central methodology

QuERIES draws inspiration from computer science, game theory, control theory, and economics, thus is a complex answer to a complex question. It is stated that it can be used as an alternative to popular methodologies such as red teaming or black-hat analysis used commonly in risk-assessment. [16]

QuERIES is proposed to have potentially significant usage in DoD (Department of Defense) and in private sector [16]. Initial testing of QuERIES in small-scale, realistic scenarios presented by Carin et al. suggest that the methodology can in

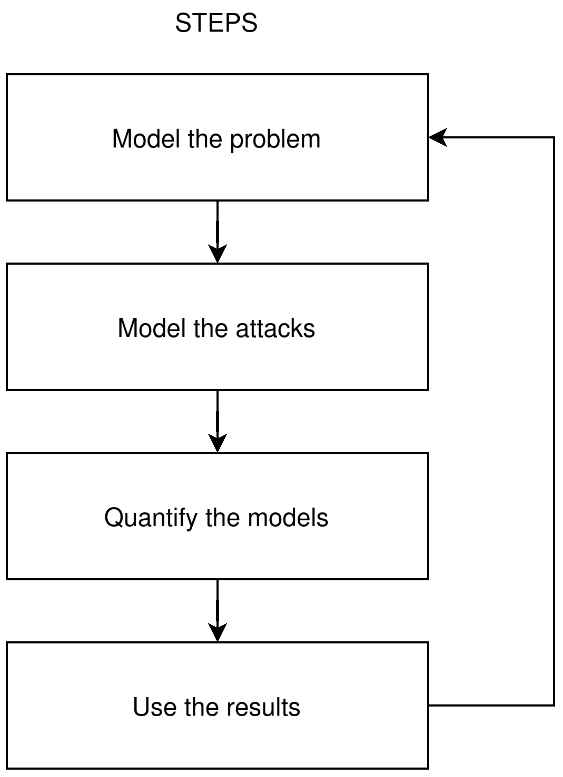


Figure 5.1: The QuERIES methodology is used as a reference flowchart for evaluation of security. [17]

fact be used as to improve risk-assessment more complex settings [16]. This thesis follows similar steps: first the QuERIES methodology is used to assess risks and then they are generalized with strict constraints in mind.

As stated by Hughes and Cybenko the result of QuERIES isn't binary: the attacker must think about the most optimal timeframe to stop the operation [17]. The strategy uses **open** and **closed** loop decision algorithm for deciding when to stop trying. Closed loop decision algorithm constantly evaluates when is the optimal time to stop trying and open loop means that the system has pre-defined goals for evaluation [16]. As as positive side-effect by gaining the probabilities through red-team evaluation the system is thoroughly tested and improved. This is a valuable metric, providing insight on how does the value of the system transition in the worth of breaching it in certain time, thus reflecting the true cost of the attack. As stated by Hughes and Cybenko, if the value is high enough related to the value of the gained value, they will less probably perform the attacks [17].

In this thesis, the value of the whole system architecture is defined as 1, constituting a value of holded intellectual property (IP). Costs are defined as fractions of the value, deflating 0.1 every hour. This is contrary to the original paper, where the value of the intellectual property has a certain value of 30 000$ and the value for the possible intruder is defined as 60$ per hour [16]. This divergency is due to the fact that it's impossible to define a certain value for our system, and it also provides clarity, as it's ergonomic to see how attack estimates relate to the value of the intellectual property.

5.5.3 Partially observable Markov decision process

Lastly, using the results of the POMDP can be used to increase the protection against the discovered problems. POMDP provides us data on two things: found security breaches and their security implications through the reward function [16]. The models and attacks are quantified through a partially observable Markov decision process (POMDP) which contains these six steps:

1. Define possible states the system can be in
2. Define the actions the system can take
3. Define the possible observations the system can take
4. Define the transition probabilities of the system
5. Define the observation probabilities of the system
6. Rewards: guide the system towards the desirable actions and states.

[17]

A POMDP is used widely in this kind of applications, as both blue and red team have only partial observations in relation to the system [47]. The blue team can't be completely certain that the system is secure and the red team cannot perform fully reliably as systems and environments differ from each other. As mentioned in the previous section, the focus of the QuERIES analysis is the time to reverse-engineer the system, thus emphasizing the importance of only partial observations.

In the next section the architecture presented in chapter 4 is improved using the applied QuERIES methodology. An attack graph is constructed and POMDPs are utilized. The result will provide information about the architectures security, and it will undergo several iterations of the QuERIES pipeline with each improving the overall security of the system.

6 Applying QuERIES

As mentioned in the previous chapter section 5.1, the QuERIES methodology is used in this chapters analysis. In table 6.1 is listed all the possible states (S), defender actions (D), attacker actions (A) and observations (O). Then, every transition from state to another state is calculated as a probability.

QuERIES as a model is described as multidisciplinary, as it takes inspiration from traditional red-blue team approaches, mathematical models to economic models [17]. In other literature, particularly by Bojanc and Jerman-Blažič, it's argued that modeling cybersecurity with economic models can provide substance for minimizing risks organization-wide [48]. In this chapter, we will apply the model and discuss the implications in both security and economic standpoints. The main focus however is on how using QuERIES, we can improve the security of the architecture.

6.1 Modeling the problem and quantifying the models

The example project of this thesis is an image showing architecture that could be used e.g for advertisement, public transport timetables or practically anywhere where static media should be presented. The results if an an intruder should gain unauthorised access, would be anywhere from displaying explicit imagery to succeeding in displaying propaganda or other unwanted content. Unauthorised access would have a negative economic effect for the service provider, as every organisation displaying media want to remain credible among stakeholders.

The attack surface of the example project focuses on physical access and vulnerabilities in remote connections. With MQTT-messaging, SSH and display protocols, internal and external messaging takes place.

As mentioned in chapter 5 section 5.3 subsection 5.5.2, the value of intellectual property is capped at 1, for which other parameters refer to as fractions of it. If this was applied to a real setting, the value of intellectual property would be calculated appropriately for the scenario.

6.2 Modeling the possible attacks

In this section, the possible attacks are modeled using an attack graph, depicted as a POMDP, which is a modeling tool presented earlier. In the original paper where QuERIES is presented, an important parameter is the time to reverse-engineer the system without prior information about the protection scheme [16]. Our approach is different; the attacks are considered as successful, if they gain further leverage in the attack graph, e.g transitioning state ”idle” to ”partial loss of system”. This is to maintain cohesion in the study, as we don’t need to define what it means to ”reverse engineer” the system. In our case, system configurations are publicly available as an open source project, thus the information of the system being available also to the attacker.

In table 6.1, the first column describes states the system can be in. The second and the third column state defensive actions, and observations of the system. The fourth column contains template of the attacks that could be conducted. After forming the attack graph depicted in the table, the attacker tries to breach the system, leveraging through the A0-A4 column. The probabilities are then calculated with the model presented in chapter 5 section 5.5 subsection 5.1.

The algorithm produces a reward value for each QuERIES iteration, which is used to improve the setup. For calculating reward values, R script with library ”pomdp” was used.

S0-S7	D0-D4	O0-O4	A0-A4
Idle	Monitor system	Normal operation	Intercept MQTT messaging
Receive media through MQTT	Isolate system	Detected suspicious activity	Compromise Github repository
Set up SystemD services	Shutdown system	Detected unauthorised access	Gain physical access to device
Start Weston	Isolate device	Detected unusual media display	Exploit vulnerabilities in display
Display media			Exploit vulnerabilities in SSH connections
Partial loss of system			
Complete loss of system			

Table 6.1: Different states, defensive measures, observations and attack measures for the system.

A weight of 1 was used for positive results, and -100, if something was to be compromised. This weight distribution is due to the fact that even if blue team succeeds most of time, the results of failure are much worse than a succeeding result from the blue team. The discount constant influences the priority of immediate vs. future rewards [47]. Our case signifies the importance of both, so value of 0.75 was used. Note that the maximum reward value is 4, and the minimum is -400.

6.3 Using the results

The reward score is taken in account on how successful/unsuccessful the setup is from a security perspective. The score itself is rather abstract; its main function is to demonstrate the *overall security* of the system. In short, negative numbers imply there are flaws in security, and positive numbers mean that the system is more secure [47].

As stated by Hughes and Cybenko, using the results means evaluating the gained results to decide if proposed protections are adequate for our means [17]. The QuERIES model was iterated 2 times, and the results were placed in the table 6.2.

The reward function in the first iteration is calculated with the mentioned script. The second value is calculated as the maximum accumulated points; the highest value is 4 points without any discount, as the red team failed to provide any results for the second iteration. This is partially due to time limit, as in the test set up there was only one attacker with very limited time. It can be argued that with more time, it would have been possible for the attacker to breach.

As seen, the reward function is growing as the proposed protections are applied

Iteration	Reward function	Proposed protections
1	-25	Multiple
2	4	None

Table 6.2: Protections applied in each iteration

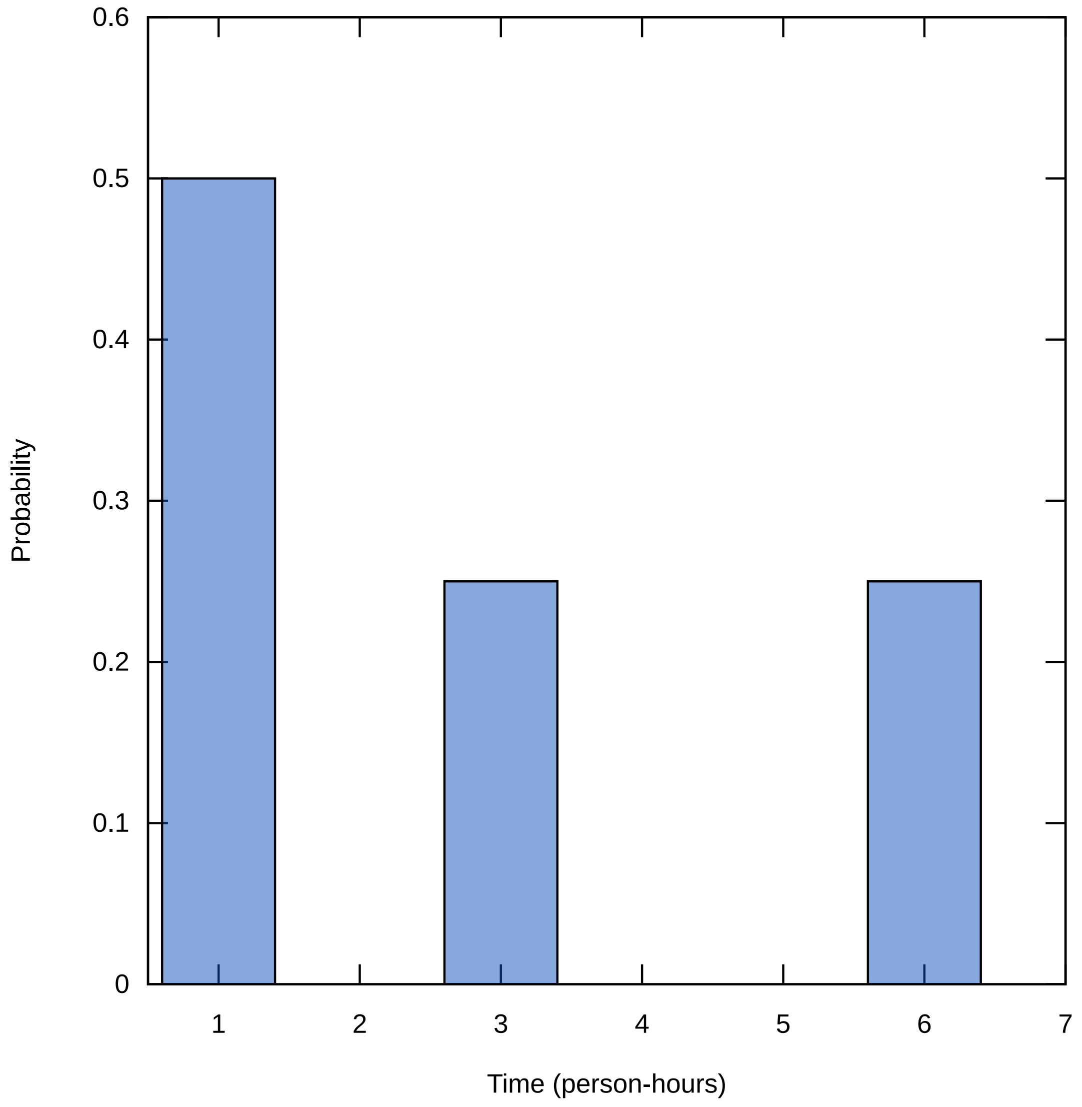


Figure 6.1: Probabilities to breach into the system.

in both iterations. In the figure 6.1 we can see that the time to breach peaks in the first hour.

The figure 6.1 has similar shape as the figure in the paper by Carin et. al, meaning that attacks in their initial phase succeed more frequently [16]. The figure 6.2 is used to decide, when is the optimal time to stop the attacks from the attacker perspective. We use this also to reflect the cost estimate of the blue team.

This economic model can be used to generalize risk in this kind of situations. In the graph 6.2 we can see that the cost for the attacker peaks at about 6 hours

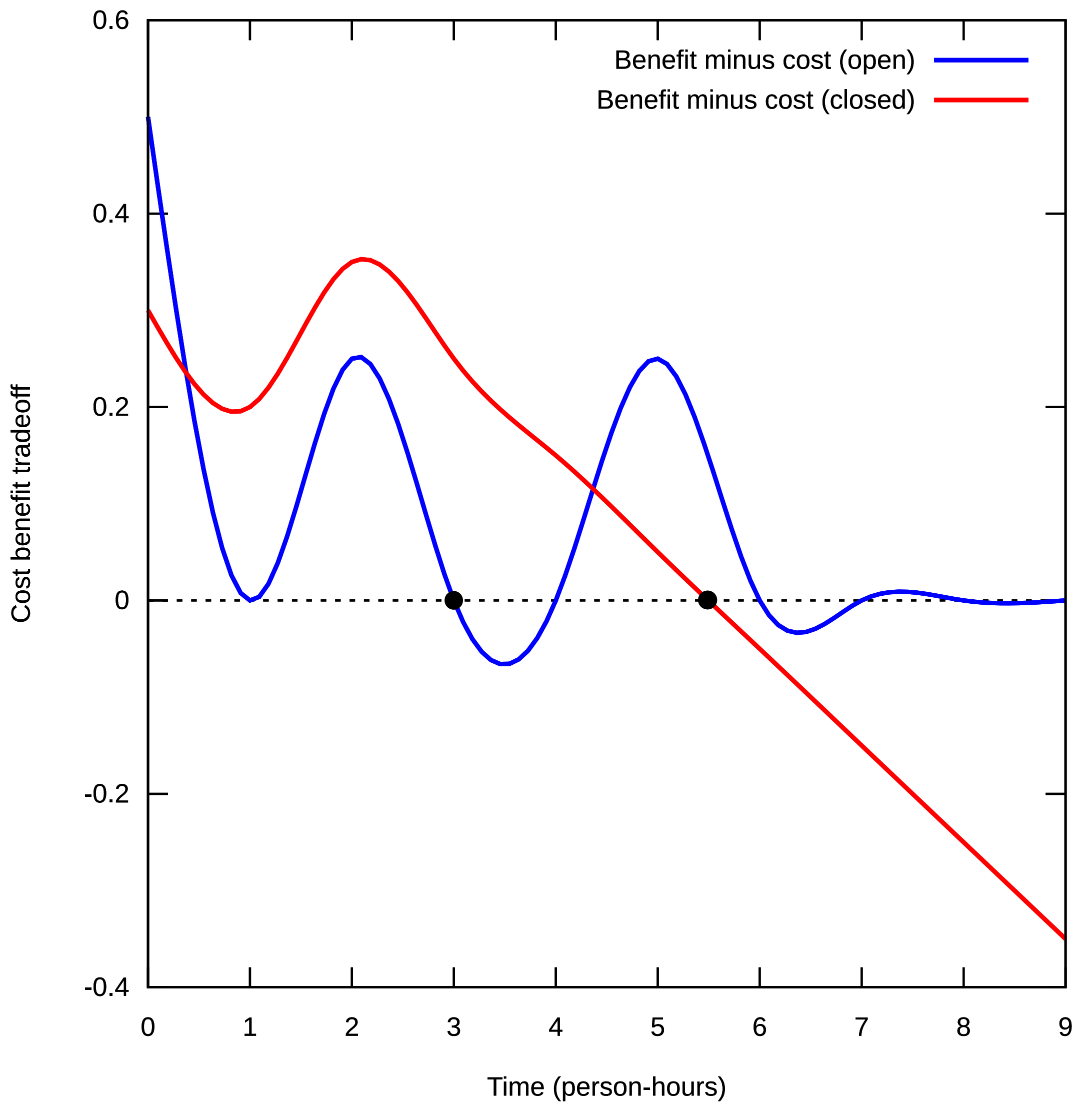


Figure 6.2: Cost benefits and reductions. The black dots indicate the most optimal times to stop the attack.

using closed algorithm, and at 3 hours using open algorithm. Open loop algorithm means that the cost estimate never evaluates feedback from a system [49]. Closed loop behaves in an opposite manner; it constantly modifies its behaviour based on the feedback it gets [49]. In our case, the values of cost using open algorithm are the values as is, and using the closed algorithm, the value deflate by 0.1 every hour.

The blue team should be aware that in the first QuERIES iteration, most of the breaches appeared in the first half of the time segment. The blue team can use this information to estimate the future distribution of breaches. In our case, the second iteration provided no breaches, but it can be argued that in different experimental design, the attacker could have succeeded. The results don't mean that the system is completely secure; instead we can use the first iterations' estimates to decide, how could the blue team measures be improved in such systems' future. The blue team measures depicted in figure 6.1 didn't have the planned effect to protect the system.

We can see that with closed loop algorithm, the attacker can gain more results with worse cost tradeoff, but with open loop, it gets immediate results with cheap cost. This could be important for the holder of the intellectual property, as it must protect itself against attackers using both algorithms. In our case, where the potential threats are e.g hacktivist groups, the cost estimate doesn't matter as much as it's important to be aware that an attacker who doesn't have economic interests, could have practically unlimited time to try to breach the system.

6.3.1 Improving the system using Nix

The following measures were taken in the first iteration: disable USB ports, change MQTT password to token based authentication and change SSH to key based authentication. As we can see, the QuERIES output provided concrete results on what are the weak points of the system.

As an example, the exposed USB-ports of the system were found to be a problem, due to the possibility of infecting the system with direct control to the system. This can be mitigated easily by modifying the client devices configuration.nix, with the following changes:

```
{ config, pkgs, ... }: {
  boot.kernelPackages = pkgs.linuxPackages_latest;
  boot.kernelParams = [
    "nousb"  # Disables USB at the kernel level
  ];
  boot.kernelModules = [];
  boot.extraModulePackages = [];
  services.udev.extraRules = ''
    SUBSYSTEM=="usb", ACTION=="add", OPTIONS+="ignore_device"
  '';
}
```

The configuration could be applied to any number of clients, proving that Nix could be used to rapidly address arising security issues. This supports the argument presented in chapter 2, that through updatability the security could be improved. This highlights the scalability of Nix systems; it doesn't matter if we have one or thousands of devices, updating them is equally simple. A huge contrast between imperative and declarative systems is found, as imperative systems would need linear administration time in relation with the number of devices. The configuration is only an example to mitigate hardware access attack vectors, as there are many more ways for the attacker to leverage the access, e.g by replacing the whole device, depending on the resources of the attacker.

6.3.2 Issues with QuERIES

One issue found using QuERIES with POMDP was that applying the reward functions is in our case rather arbitrary. We assumed, that for positive results, the reward function is defined as 1, and for the negative outcomes -100. This is due to the negative results deemed as catastrophic and the positive results being slightly positive. The choice for both parameters could have been any integer, but the issue is that it demands ”gut-feeling” of the author to select the appropriate parameters. This could be mitigated with some study on selecting the parameters, but it’s definitely a challenge to provide good foundation on such abstract numbers.

One remark using POMDP is that it produces very generalized output. This is why we use methodologies such as QuERIES to improve the system, POMDP being just one component. Other benefit from using POMDP is that it provides us concise attack graph, and it can be argued that using the POMDP calculations may even be redundant. In POMDP, negative rewards signify that the system has issues, and positive meaning that the system is more secure [47]. More ergonomic approach would be calculating rewards without positive outcomes, due to them possibly shadowing the most critical issues. Developing a pessimistic model that highlights explicitly the weak points of a system would be a topic for further research.

6.4 Generalizing the results

The results of this chapter could be generalized to a more complex setting, using a similar methodology. The POMDP model scales well, if more parameters would be supplied. This study proves that while measuring cybersecurity is difficult, remaining in tight constraints we can get adequate estimates for applying economic models for the red team, further projecting the risk vs. reward.

Carin et. al [16] argue that QuERIES can be applied in both public and private sectors to help to improve the security of both software and hardware. I argue that a hand-tailored application of QuERIES can be used as a powerful model that is ergonomic to use in the right hands. However, the use of complex applied mathematic models require proficency in both cybersecurity and mathematics, which can be hard to achieve in a real organizational setting. This is why I propose in the next chapter that a simpler, pessimistic and more concise model would be easier to reach for an organization.

7 Further research

One big limitation of Nix is the fact that it mainly supports only x86_64, i686 and arm64 platforms [23]. This is not generally enough, as many different architectures are used in embedded, thus limiting the use-cases of NixOS [15].

A declarative approach could be used with other processor architectures. This would require work on a declarative package manager, which would be used in conjunction with base systems created with Buildroot or Yocto.

As far as security is concerned, the main methodology of this thesis' research, QuERIES, proved to be an aqequate tool which helped to improve the security of the reference architecture. However, some issues were raised in chapter 6 section 6.3 subsection 6.3.2. These issues could be addressed in developing a new methodology inspired by QuERIES, providing explicit insight on the weak points of the underlying system.

7.1 Further research questions

Questions for further research include:

1. What kind of new methodology would be better, reflecting the found issues in QuERIES for similar use-case as in this thesis?
2. What set of tools would be optimal in creating a secure true cross-platform declarative package manager

Developing a new Linux distribution would be a very big task, so implementing support for different architectures for Nix or Guix for embedded use would be the focus of further research. These approaches would hold the same arguments as in this thesis to improve the security of embedded devices.

8 Conclusion

Nix can be a useful base for issuing a purely functional, declarable systems while providing an advanced rollback mechanism. Configuring it depending on the system administrator may be gruesome, but it can dodge the usual pitfalls of more popular systems. While the usage of Nix language demands proficiency, it is a solution to distribute, update and administrate more secure systems. NixOS handles exceptionally easily features such as Wayland and audio driver setup. In the end, NixOS is very predictable and the statement "if it works on one machine, it works on another" proves to be true.

This thesis' problems could be solved with many different solutions, but the Nix approach is rather unique as it makes reproducible systems very straight forward. As the system is defined in the configuration.nix file, there's little that could go wrong even when underlying hardware varies. The same doesn't apply to the mentioned Debian distribution.

Measuring and improving the architectures solution proved to be challenging but fruitful. With multiple iterations used with QuERIES, the architecture developed from insecure to more secure. This is a testament for applying successfully a certain methodology for a specific task.

As far as security implications are concerned, Nix is a good tool for asserting compliance. In purely functional environment, anything that needs compliance can be checked from the configuration files. As disk encryption is trivial and read-only

Nix store provides increased integrity, a more secure system can be achieved also through the actions of a system administrator. A system administrator must adhere at to Nix philosophy with the Nix language and as all the configured parts can be viewed quickly. For example, a question ”what ports are in use in each client device” can be answered very quickly and in precise manner by just viewing the configuration files.

As mentioned, popular embedded Linux distribution creating tools, Buildroot and especially Yocto have also steep learning curves. They also lack the ease of updatability provided by Nix, which is a significant problem regarding security.

The research proved that the purely functional Nix system is easily updatable and upgradable solution for maintaining a secure system. While NixOS is a niche Linux distribution, and probably remains as such, the concepts, however I assume will live on in future. A purely functional declarative system with atomic rollbacks is something I'll await to be iterated in the future with further user availability in mind.

References

[1] G. G. contributors, *Packages &x2014; GNU Guix — packages.guix.gnu.org*, `https://packages.guix.gnu.org/`, [Accessed 24-09-2024].

[2] E. Dolstra and A. Löh, “Nixos: A purely functional linux distribution”, in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, 2008, pp. 367–378.

[3] *System Recovery and Snapshot Management with Snapper | Reference | openSUSE Leap 15.0 — doc.opensuse.org*, `https://doc.opensuse.org/documentation/leap/archive/15.0/reference/html/book.opensuse.reference/cha.snapper.html`, [Accessed 16-09-2024].

[4] *GitHub - nixos-bsd/nixbsd: An unofficial NixOS fork with a FreeBSD kernel — github.com*, `https://github.com/nixos-bsd/nixbsd`, [Accessed 09-11-2024].

[5] S. van der Burg, *A Reference Architecture for Distributed Software Deployment.* Citeseer, 2013.

[6] *NixOS Search — search.nixos.org*, `https://search.nixos.org/packages`, [Accessed 24-09-2024].

[7] L. Courtès, “Déploiements reproductibles dans le temps avec gnu guix”, *GNU/Linux Magazine*, 2021.

[8] A. J. A. Wang, “Information security models and metrics”, in *Proceedings of the 43rd annual Southeast regional conference-Volume 2*, 2005, pp. 178–184.

[9] A. Ramos, M. Lazar, R. Holanda Filho, and J. J. Rodrigues, “Model-based quantitative network security metrics: A survey”, *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2704–2734, 2017.

[10] S. van der Burg and E. Dolstra, “Declarative testing and deployment of distributed systems”, *Technical Report Series TUD-SERG-2006-020*, 2010.

[11] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, “Declarative vs. imperative: Two modeling patterns for the automated deployment of applications”, in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*, Xpert Publishing Services (XPS), 2017, pp. 22–27.

[12] E. Specht, R. M. Redin, L. Carro, L. d. C. Lamb, E. F. Cota, and F. R. Wagner, “Analysis of the use of declarative languages for enhanced embedded system software development”, in *Proceedings of the 20th annual conference on Integrated circuits and systems design*, 2007, pp. 324–329.

[13] R. Kandoi and K. Hartke, “Operating large-scale iot systems through declarative configuration apis”, in *Proceedings of the 2021 Workshop on Descriptive Approaches to IoT Security, Network, and Application Configuration*, 2021, pp. 22–25.

[14] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, “Security in embedded systems: Design challenges”, *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 3, pp. 461–491, 2004.

[15] K. Fysarakis, G. Hatzivasilis, K. Rantos, A. Papanikolaou, and C. Manifavas, “Embedded systems security challenges”, in *Measurable security for Embedded Computing and Communication Systems*, SCITEPRESS, vol. 2, 2014, pp. 255–266.

[16] L. Carin, G. Cybenko, and J. Hughes, "Cybersecurity strategies: The queries methodology", *Computer*, vol. 41, no. 8, pp. 20–26, 2008.

[17] J. Hughes and G. Cybenko, "Quantitative metrics and risk assessment: The three tenets model of cybersecurity", *Technology Innovation Management Review*, vol. 3, no. 8, 2013.

[18] U. Breitenbücher, K. Képes, F. Leymann, and M. Wurster, "Declarative vs. imperative: How to model the automated deployment of iot applications?", *Proceedings of the 11th advanced summer school on service oriented computing*, pp. 18–27, 2017.

[19] E. Dolstra and A. Hemel, "Purely functional system configuration management.", in *HotOS*, 2007.

[20] Canonical, *Ubuntu Manpage: Apt - command-line interface — manpages.ubuntu.com*, `https://manpages.ubuntu.com/manpages/bionic/man8/apt.8.html`, [Accessed 22-05-2024].

[21] G. K. Thiruvathukal, "Gentoo linux: The next generation of linux", *Computing in science & engineering*, vol. 6, no. 5, pp. 66–74, 2004.

[22] S. Van Der Burg and E. Dolstra, "Automating system tests using declarative virtual machines", in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, IEEE, 2010, pp. 181–190.

[23] *NixOS Manual — nixos.org*, `https://nixos.org/manual/nixos/stable/`, [Accessed 22-05-2024].

[24] E. Dolstra, R. Vermaas, and S. Levy, "Charon: Declarative provisioning and deployment", in *2013 1st International Workshop on Release Engineering (RELENG)*, IEEE, 2013, pp. 17–20.

[25] E. Dolstra, A. Löh, and N. Pierron, "Nixos: A purely functional linux distribution", *Journal of Functional Programming*, vol. 20, no. 5-6, pp. 577–615, 2010.

[26] *User Environment - NixOS Wiki — nixos.wiki*, `https://nixos.wiki/wiki/User_Environment`, [Accessed 09-11-2024].

[27] *Nix/Nixpkgs/NixOS — github.com*, `https://github.com/nixos`, [Accessed 08-11-2024].

[28] *Home Manager Manual — nix-community.github.io*, `https://nix-community.github.io/home-manager/`, [Accessed 24-05-2024].

[29] *Flakes - NixOS Wiki — nixos.wiki*, `https://nixos.wiki/wiki/Flakes`, [Accessed 22-05-2024].

[30] *Nix channels - NixOS Wiki — nixos.wiki*, `https://nixos.wiki/wiki/Nix_channels`, [Accessed 22-05-2024].

[31] D. N. Serpanos and A. G. Voyiatzis, "Security challenges in embedded systems", *ACM Transactions on embedded computing systems (TECS)*, vol. 12, no. 1s, pp. 1–10, 2013.

[32] *History of SSH | SecOps® Solution — secopsolution.com*, `https://www.secopsolution.com/blog/history-of-ssh`, [Accessed 18-03-2024].

[33] N. Turab and Q. Kharma, "Secure medical internet of things framework based on parkerian hexad model", *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 6, 2019.

[34] E. Dolstra, M. De Jonge, E. Visser, *et al.*, "Nix: A safe and policy-free system for software deployment.", in *LISA*, vol. 4, 2004, pp. 79–92.

[35] R. A. Kemmerer, "Cybersecurity", in *25th International Conference on Software Engineering, 2003. Proceedings.*, IEEE, 2003, pp. 705–715.

[36] G. Pender-Bey, “The parkerian hexad”, *Information Security Program at Lewis University*, 2019.

[37] *Security - NixOS Wiki — nixos.wiki*, `https://nixos.wiki/wiki/Security`, [Accessed 22-05-2024].

[38] *MQTT Version 3.1.1 — docs.oasis-open.org*, `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`, [Accessed 22-05-2024].

[39] *Wayland — wayland.freedesktop.org*, `https://wayland.freedesktop.org/docs/html/`, [Accessed 18-09-2024].

[40] *GitHub - nix-community/nixos-anywhere: Install nixos everywhere via ssh [maintainer=@numtide] — github.com*, `https://github.com/nix-community/nixos-anywhere`, [Accessed 01-10-2024].

[41] J. Bayuk and A. Mostashari, “Measuring systems security”, *Systems Engineering*, vol. 16, no. 1, pp. 1–14, 2013.

[42] Y. V. Papazov, “Cybersecurity metrics”, *NATO Science and Technology Organization (STO)*, pp. 1–18, 2019.

[43] S. C. Payne, “A guide to security metrics”, *SANS Institute Information Security Reading Room*, 2006.

[44] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, “A survey on systems security metrics”, *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–35, 2016.

[45] B. Alshammari, C. Fidge, and D. Corney, “Security metrics for object-oriented class designs”, in *2009 Ninth International Conference on Quality Software*, IEEE, 2009, pp. 11–20.

[46] H. Wang, S. Roy, A. Das, and S. Paul, “A framework for security quantification of networked machines”, in *2010 Second International Conference on COMmunication Systems and NETworks (COMSNETS 2010)*, IEEE, 2010, pp. 1–8.

[47] J. C. M. Ashely S. M McAbee Murali Tummala, *The use of partially observable Markov decision processes to optimally implement moving target defense*, `https://scholarspace.manoa.hawaii.edu/server/api/core/bitstreams/86f32196-380c-443a-88b5-07f07649137e/content`, [Accessed 23-05-2024].

[48] B. Jerman-Blažič *et al.*, “An economic modelling approach to information security risk management”, *International Journal of Information Management*, vol. 28, no. 5, pp. 413–422, 2008.

[49] R. Bars, P. Colaneri, C. E. de Souza, *et al.*, “Theory, algorithms and technology in the design of control systems”, *Annual Reviews in Control*, vol. 30, no. 1, pp. 19–30, 2006.

The following R code calculates the result score from the study in 6.

```
library("pomdp")


CYBSEC_POMDP <- POMDP(
    states = c("Idle", "Receiving media through MQTT", "Setting up SystemD serv
    actions = c("Monitor system", "Reboot system", "Update system", "Display me
    observations = c("Normal operation", "Detected suspicious activity", "Detec
    transition_prob = list(
        "Monitor system" = "uniform",
        "Reboot system" = "uniform",
        "Update system" = "uniform",
        "Display media" = "uniform",
        "Recover from error" = "uniform",
        "Intercept MQTT messaging" = "uniform",
        "Compromise Github repository" = "uniform",
        "Gain physical access to the device" = "uniform",
        "Exploit vulnerabilities in display" = "uniform",
        "Exploit vulnerabilities in SSH" = "uniform"
    ),
    observation_prob = list(


        "Monitor system" = "uniform",
        "Reboot system" = "uniform",
        "Update system" = "uniform",
        "Display media" = "uniform",
        "Recover from error" = "uniform",
        "Intercept MQTT messaging" = "uniform",
        "Compromise Github repository" = "uniform",
        "Gain physical access to the device" = "uniform",
        "Exploit vulnerabilities in display" = "uniform",
        "Exploit vulnerabilities in SSH" = "uniform"
        #"Normal operation" = "uniform",
        #"Detected suspicious activity" = "uniform",
        #"Detected system error " = matrix(c(0.85, 0.15, 0.15, 0.85), nrow = 2
        #"Detected unusual media display" = matrix(c(0.85, 0.15, 0.15, 0.85), 
        #"Detected unauthorised access" = matrix(c(0.85, 0.15, 0.15, 0.85), nr
    ),
    reward = rbind(
        R_("Monitor system",  "Idle", "*", "*", 10),
        R_("Monitor system",  "Receiving media through MQTT", "*", "*", 10),
        R_("Monitor system",  "Setting up SystemD services",  "*", "*", 10),
        R_("Monitor system",  "Starting Weston", "*", "*", 10),
        R_("Monitor system",  "Displaying media", "*", "*", 10),
        R_("Monitor system",  "Error state", "*", "*", 10),
        R_("Monitor system",  "Partial loss of system", "*", "*", 10),
        R_("Monitor system",  "Complete loss of system", "*", "*", 10),

        R_("Reboot system",  "Idle",  "*", "*", 10),
        R_("Reboot system", "Receiving media through MQTT", "*", "*", 10),
        R_("Reboot system", "Setting up SystemD services",  "*", "*", 10),
        R_("Reboot system", "Starting Weston", "*", "*", 10),
        R_("Reboot system", "Displaying media", "*", "*", 10),
        R_("Reboot system", "Error state", "*", "*", 10),
        R_("Reboot system", "Partial loss of system", "*", "*", 10),
        R_("Reboot system", "Complete loss of system", "*", "*", 10),

        R_("Update system",  "Idle",  "*", "*", 10),
        R_("Update system",  "Receiving media through MQTT", "*", "*", 10),
        R_("Update system", "Setting up SystemD services",  "*", "*", 10),
        R_("Update system",  "Starting Weston", "*", "*", 10),
        R_("Update system",  "Displaying media", "*", "*", 10),
        R_("Update system",  "Error state", "*", "*", 10),
        R_("Update system",  "Partial loss of system", "*", "*", 10),
        R_("Update system",  "Complete loss of system", "*", "*", 10),
        R_("Display media", "Idle", "*", "*", 10),
        R_("Display media", "Receiving media through MQTT", "*", "*", 10),
        R_("Display media","Setting up SystemD services",  "*", "*", 10),
        R_("Display media","Starting Weston", "*", "*", 10),
        R_("Display media","Displaying media", "*", "*", 10),
        R_("Display media","Error state", "*", "*", 10),
        R_("Display media","Partial loss of system", "*", "*", 10),
        R_("Display media","Complete loss of system", "*", "*", 10),

        R_("Recover from error",  "Idle", "*", "*", 10),
        R_("Recover from error", "Receiving media through MQTT", "*", "*", 10),
        R_("Recover from error", "Setting up SystemD services",  "*", "*", 10),
        R_("Recover from error", "Starting Weston", "*", "*", 10),
        R_("Recover from error", "Displaying media", "*", "*", 10),
        R_("Recover from error", "Error state", "*", "*", 10),
        R_("Recover from error", "Partial loss of system", "*", "*", 10),
        R_("Recover from error", "Complete loss of system", "*", "*", 10),

        R_("Intercept MQTT messaging", "Idle", "*", "*", -100),
        R_("Intercept MQTT messaging","Receiving media through MQTT", "*", "*",
        R_("Intercept MQTT messaging","Setting up SystemD services",  "*", "*",
        R_("Intercept MQTT messaging","Starting Weston", "*", "*", -100),
        R_("Intercept MQTT messaging","Displaying media", "*", "*", -100),
        R_("Intercept MQTT messaging","Error state", "*", "*", -100),
        R_("Intercept MQTT messaging","Partial loss of system", "*", "*", -100)
        R_("Intercept MQTT messaging","Complete loss of system", "*", "*", -10
        R_("Compromise Github repository", "Idle", "*", "*", -100),
        R_("Compromise Github repository", "Receiving media through MQTT", "*",
        R_("Compromise Github repository", "Setting up SystemD services",  "*",
        R_("Compromise Github repository", "Starting Weston", "*", "*", -100),
        R_("Compromise Github repository", "Displaying media", "*", "*", -100),
        R_("Compromise Github repository", "Error state", "*", "*", -100),
        R_("Compromise Github repository", "Partial loss of system", "*", "*",
        R_("Compromise Github repository", "Complete loss of system", "*", "*",

        R_("Gain physical access to the device", "Idle", "*", "*", -100),
        R_("Gain physical access to the device", "Receiving media through MQTT"
        R_("Gain physical access to the device", "Setting up SystemD services",
        R_("Gain physical access to the device", "Starting Weston", "*", "*", -
        R_("Gain physical access to the device", "Displaying media", "*", "*",
        R_("Gain physical access to the device", "Error state", "*", "*", -100)
        R_("Gain physical access to the device", "Partial loss of system", "*",
        R_("Gain physical access to the device", "Complete loss of system", "*"

        R_("Exploit vulnerabilities in display", "Idle", "*", "*", -100),
        R_("Exploit vulnerabilities in display","Receiving media through MQTT",
        R_("Exploit vulnerabilities in display","Setting up SystemD services",
        R_("Exploit vulnerabilities in display","Starting Weston", "*", "*", -
        R_("Exploit vulnerabilities in display","Displaying media", "*", "*", -
        R_("Exploit vulnerabilities in display","Error state", "*", "*", -100),
        R_("Exploit vulnerabilities in display","Partial loss of system", "*",
        R_("Exploit vulnerabilities in display","Complete loss of system", "*",
        R_("Exploit vulnerabilities in SSH", "Idle", "*", "*", -100),
        R_("Exploit vulnerabilities in SSH","Receiving media through MQTT", "*
        R_("Exploit vulnerabilities in SSH","Setting up SystemD services",  "*
        R_("Exploit vulnerabilities in SSH","Starting Weston", "*", "*", -100)
        R_("Exploit vulnerabilities in SSH","Displaying media", "*", "*", -100
        R_("Exploit vulnerabilities in SSH","Error state", "*", "*", -100),
        R_("Exploit vulnerabilities in SSH","Partial loss of system", "*", "*"
        R_("Exploit vulnerabilities in SSH","Complete loss of system", "*", "*
    ),
    start = "uniform",
    discount = 1.0,
    name = "CYBSEC Problem POMDP"
)
sol <- solve_POMDP(CYBSEC_POMDP)
sol
```