COMP 4905 Honours Project Report

BPFContain Empirical Policy Development Dashboard

Carleton University
Author: Cody Dudley
Author Student Number: 101089520
Author Email Address: codydudley@cmail.carleton.ca
Supervisor: Anil Somayaji
Date: December 17th 2021
Course Code: COMP3999A [20172]
Email Address: codydudley@cmail.carleton.ca

Abstract

Policy-based containerization frameworks such as BPFContain aim to solve the confinement issues presented by modern container frameworks. That is, these frameworks enforce the principle of least privilege by restricting resources with a specialized policy depending on the software being contained. With BPFContain, these policies are easy to reason about and lightweight, but developing them in the first place can be challenging [1].

While BPFContain and similar frameworks are a significant leap ahead in container security, the burden of policy development hurts their adoptability. We thus present BPFCAudit, an application which assists BPFContain users in policy development and verification. BPFCAudit builds on BPFContain and collects event data from live containers to build a report of the container's behaviour for a period of time. This report's data is then used in a graphical user interface (GUI) to display potential issues with the user's policy, and to display how frequently a given rule of the user's policy is utilized. Using this information, the user can then fine-tune their policy to ensure that it permits access to necessary resources and denies access to others.

This report presents the challenges presented by BPFContain policy development which motivated the creation of BPFCAudit and its associated requirements. It also presents the design and implementation of BPFCAudit, the technologies used and some experimental performance and usability results of the application. The report concludes with a discussion of how well BPFCAudit satisfies its requirements in its current state, a brief presentation of similar policy development interfaces, any limitations, and the future of the project.

Acknowledgements

I express my deepest thanks to my supervisor Dr. Anil Somayaji for his feedback and guidance throughout this project. This endeavour would not have been possible without his dedicated support of BPFContain, its predecessors and to this project itself. I am grateful and proud that I was able to deliver this project under his supervision and I believe that I am a better and more knowledgeable developer for it.

I would also like to thank William Findlay for his encouragement and support throughout the development of this project. William was always willing to help with any query I had with respect to BPFContain and more. Of course, his excellent BPFContain and master's thesis directly inspired and even suggested this project. He also actively and deliberately contributed to the development of this project by creating BPFContain interfaces designed to support tools such as the one presented in this report and for this, I am extremely grateful.

I also want to express my thanks to my family and to my best friend and love Olivia who supported me throughout this project and my higher education. Their encouragement and day-to-day assistance have been critical to the development of this project. I also express specific gratitude to Olivia for sifting through pages of technical jargon to edit this report and help me deliver a better report than I ever could have on my own.

Additionally, I want to thank all other individuals and organizations that have helped to make this project possible. Specifically, I am grateful to the open-source community that continues to develop ground-breaking software at no cost.

Table of Contents

List of Figures

List of Tables

1 Introduction

For many organizations, containers have established themselves as a critical technology. The predictability, scalability, and virtualization of operating system (OS) resources provided by containers has greatly simplified the software deployment process in comparison to virtual machine deployments [2]. Although modern container frameworks such as Docker or Linux Containers greatly improve the software deployment process, they fail to sufficiently isolate their containers from the host OS and other containers [1].

As an added confoundment to the container security problem, some container frameworks combine a variety of complex confinement mechanisms to achieve some degree of confinement [1]. This results in difficult to write and difficult to audit security policies [1]. Other containment frameworks such as Docker use overly general default confinement policies which protect only the most sensitive resources [1]. These default policies often do not translate to sensible security policies on a per-container basis as they can often be much more restrictive. The confinement mechanisms of these container frameworks also often “fail open”. As an example, the default Docker confinement policy relies on AppArmor, and if AppArmor is not available on the host OS, the deployment continues as usual without warning [1].

To address the container security problem, the BPFContain container framework was introduced in which security is a core design philosophy rather than an afterthought. BPFContain allows developers to define security policies on a per-container basis in a concise but nuanced high-level policy language [1]. BPFContain also provides a sensible default policy that is combined with the user-defined policy to ease some of the burden of policy development [1].

Rather than using a variety of confinement mechanisms to enforce its policies, BPFContain uses a combination of the eBPF Linux kernel technology and kernel LSM (Linux Security Module) hooks [1]. A key result using just one security mechanism for policy enforcement is that the resulting security policies are more lightweight and easier to reason about [1].

To enforce the computer security design principle of least privilege [3] on a per-container basis, we would ideally use the most restrictive policy possible in order to reduce attack surface. As BPFContain policies must currently be "hand-written" by a policy author, this is in some circumstances no easy task and is acknowledged as an area of future work by the thesis accompanying BPFContain [1]. This is problematic as users may find authoring restrictive policies for complex systems or for systems that they did author themselves difficult. This burden of writing policies on a per-container basis hurts the adoptability of BPFContain. Thus, we need some solution to promote policy development.

The thesis proposing BPFContain presents an approach for policy generation. This approach suggests deriving policies from BPFContain's existing audit logs [1]. To support policy development, we propose the BPFContainAudit (BPFCAudit) prototype which implements this strategy of deriving policies from audit logs. We also propose a graphical interface to assist the user in policy development. In the remainder of this report, we explore BPFCAudit's architecture implementation, limitations, evaluation methodologies and findings, and we also present some similar policy development tools for consideration.

1.1 BPFContain Policies

BPFContain policies always include the name of the policy, along with optional rules to restrict or grant access to certain resources, and some tunable parameters that affect how enforcement is undertaken [1]. These rules can be specified as either allow, deny, forcequit or taint rules and act to either augment or define exceptions to the default policy [1]. If a policy is in default allow mode, the triggering of a taint rule transitions the container into default deny mode if it is not in it already. A container can start in either default allow, or default deny mode depending on the parameters set in the policy. If a forcequit rule is triggered, the container is immediately killed.

An important tool we can use that can assist with our goal of policy development is BPFContain's concept of complaining mode. Complaining mode causes events that would be treated as a denial to be instead granted access [1]. Because of this, code paths can be run that would otherwise never be executed due to not having some required resource. If complaining mode was not enabled, the code would likely enter some error handling case and not be able to fully exercise all normal code paths. This allows for easier development and analysis of policies since we can exercise all code paths even with an incomplete policy that does not grant sufficient privileges to a container.

1.2 BPFContain Policy Development Challenges

With our understanding of BPFContain's policies, we can define what kinds of policies best support the security design principle of least privilege. Since default deny policies deny access to all resources and interfaces that are defined in the default policy, we want to ensure that all policies that we develop either begin in default deny mode or eventually enter it using taint rules in order to make the most restrictive policies possible. While developing our policies, they should be set to complaining mode so that all code paths can be exercised. Once development is finished, our policies should be in enforcement mode so that deny rules are correctly enforced.

As modern systems grow increasingly more complex, it becomes more and more difficult to reason about what resources they require access to and what actions they should be allowed to take. This makes the task of writing a policy for these systems that much more difficult since the author would have to consider the full set of resources that the program requires and write rules that allow access (since we are assuming a default deny policy is desirable). An alternative to this would be digging through or doing further processing against the BPFContain audit logs which is a further burden on the policy developer.

Another challenge faced by BPFContain policy developers is the policy validation process. That is, if changes are made to a complex application, how would a policy developer ensure that the current policy permits all actions that are required, and simultaneously denies actions that are potentially no longer required? The developer would again have to go through BPFContain's audit logs to check for new rules triggering (to permit an action), or worse check to confirm for the absence of events corresponding to an existing rule (to remove a "dead" rule from the policy). A better solution would be an interface that could automatically identify these changes for users.

1.3 BPFCAudit Requirements

From the challenges developers are faced with when writing BPFContain policies, we can derive a set of high-level functional requirements for our policy development interface:

F1 Display how many times each rule of a policy has been triggered in a GUI (including those covered by the default policy). This removes the work involved with going through the BPFContain audit logs to determine which rules to add or remove from a given policy.

F2 Collect audit event data from live BPFContain containers. This supports F1 such that this provides the data required for displaying rule usages. To trigger audit events for a container, it is expected that either automated tests or manual regression tests would be run against the container to exercise as many code paths as possible to maximize the diversity of audit events.

F3 Persist collected audit event data into "audits" each of which contain a set of audit event data from a specified time period. This supports our goal of enabling policy validation since users can utilize historical audit event data to double-check the validity of their policy. Users should also be able to "tag" these audits so that they can track which audit is associated with a specific version of their contained software.

For each of these requirements, we will explore the architectural and implementation choices that were made to support them. We will also explore how effective BPFCAudit is in implementing these functional requirements. Furthermore, we will detail how these functional requirements support policy development and policy verification in practice.

We can also suggest a set of non-functional requirements for BPFCAudit to ensure that our solution is useful in practice:

NF1 Efficiency. Especially with respect to audit event data collection (F2). We expect that BPFCAudit can ingest audit event data as quickly as BPFContain can provide it.

NF2 Deployment. It should be easy to install BPFCAudit on a system to promote adoptability. Since BPFCAudit is designed to save user's time, it should not waste their time in the deployment phase.

NF3 Extensibility. If another policy-based container framework were created (a framework like BPFContain), it should be easy to modify BPFCAudit so that it can support policy development for this new framework.

NF4 Usability. The BPFCAudit GUI should be clean, easy to understand and intuitive.

NF5 Reliability. BPFCAudit audits (F3) should never fail unexpectedly such that the time invested collecting audit event data is wasted.

2 Design and Implementation

This section introduces the architecture of BPFCAudit, including its components and their implementations. Specifically, we elaborate on the choice of technologies, how each component supports our functional and non-functional requirements and what interfaces BPFCAudit provides. We also present BPFCAudit's data model to support our discussion.

2.1 Overview

2.1.1 Architecture

BPFCAudit's components form a "three-tier" web application architecture where concerns are separated into three distinct components [4]. These concerns are the business logic layer (Section 2.2) where audits are carried out and audit event data is received from the BPFContain daemon, the presentation layer (Section 2.3) where our policies and audit event data are displayed, and finally the persistence layer where data such as audit events are durably persisted for later access. These components and their interactions, as well as interactions with BPFContain are depicted in Figure 2.1.

The frontend (presentation layer) communicates with the backend (business logic layer) using application programming interfaces (APIs) that follow the json:api specification [5]. For discussion of this choice of API specification, see Section 2.3.2. The backend communicates with the datastore using the Java Spring Data JPA library. Furthermore, the backend communicates with the BPFContain daemon using a WebSocket connection whose data follows the json-rpc 2.0 specification [6].

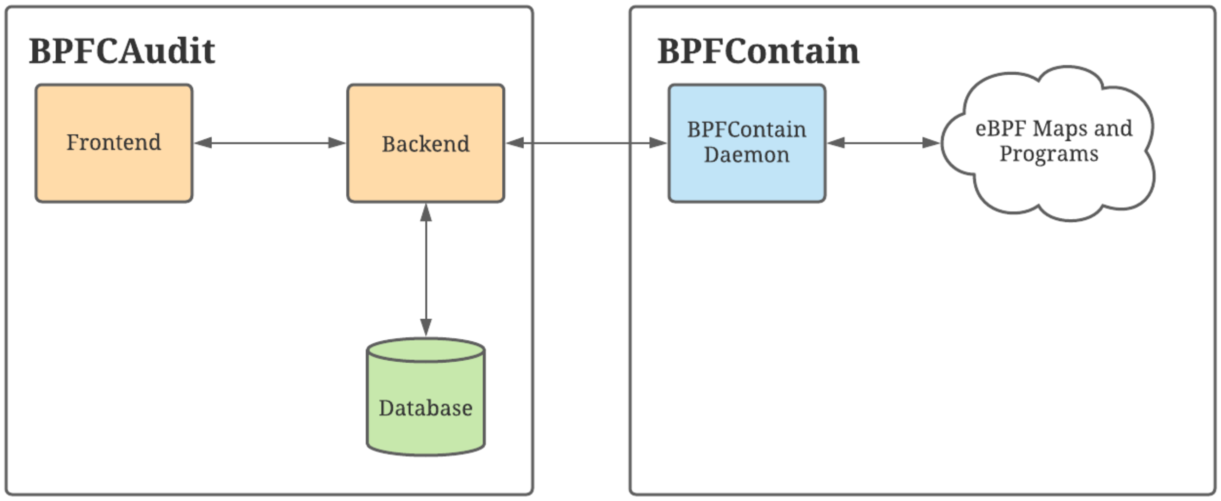


Figure 2.1: BPFCAudit's architecture. Arrows indicate communications between the various components. Only the daemon of the BPFContain architecture is detailed since this is the only component that BPFCAudit interacts with. Details of the communications originating from BPFCAudit components are discussed in Section 2.2.

2.1.2 Data Model

To begin our discussion on the implementation of BPFCAudit, we first need to define its data model so that we may reference these concepts with ease. The service model is the root object of the BPFCAudit data model. A given service model is an abstraction of an application or program that the user contains with BPFContain. An individual service should map to just one contained application, in that it is not the intention for a service to model one application and then again be used to model an unrelated application. Services promote the continuous development and verification of policies since they can be used to build a version history with audits as we will soon explore. Services also help the user organize their audits since they can be associated with something representing their application.

We also define the audit model. A given service can have many audits, and audits are associated with one service. An audit represents a report that tracks a collection of audit events that were captured for a specified period for a given service. The audit contains some summarized information about the audit events that were captured, such as a total count of audit events. Audits also serve to help organize captured events into a logical unit of time. If all audit event data for a service was persisted without associated audits, this audit event data would eventually become stale (say, once the application in question was updated) and the user would be forced to wipe out all persisted audit event data since they would have no concept of when the audit event data was collected. This way the user can capture new audit event data while keeping their old audit event data if they wish.

Finally, we define the rule model. A given rule belongs to one audit, and audits can have many rules. Rules represent a specific rule in either the BPFContain default policy or in the user-defined policy. This model tracks only the hash of the rule in question (provided by BPFContain for later comparison) and how many times this rule had an associated audit event occur during a given audit. Note that a model is not created for each audit event we receive from BPFContain for reasons that we explore in Section 2.2.2.

2.2 Backend

Motivating the creation of the BPFCAudit backend are the functional requirements F2 and F3. That is this backend should be able to collect audit event data from running BPFContain containers and ensure that this data is persisted for later policy analysis. Furthermore, the backend should be able to organize this data into audits which restricts the collection of audit event data to a certain domain of time. The BPFCAudit backend must also publish APIs for the management of these audits and their associated rule data. These APIs are needed so that the BPFCAudit GUI can call these interfaces so that end users can seamlessly manage their audits and use their associated audit event data for policy analysis. We now discuss with what technologies these features were implemented and the details of said implementations.

2.2.1 Technologies

We now discuss some of the technologies that were used to implement BPFCAudit's backend and why they were used over alternatives. Java was the programming language of choice due to the maturity of its libraries and frameworks, and due to my prior experience developing in Java. To avoid spending a large portion of development time configuring the backend and writing boilerplate code (verbose code that is repetitive and carries little functional benefit), the Spring Boot framework was used. Spring Boot is an extension of the Spring framework and takes an opinionated stance which promotes a quick start to new applications by providing sensible defaults for most configuration options [7]. To generate an initial application, Spring Initializr was used which can automatically generate a ready-to-run Maven project [8]. Spring Initializr was also used to add dependencies to the initial application and automatically configure them with sensible defaults.

To support F3, particularly the need to persist audit events, PostgreSQL was chosen as a database. PostgreSQL is an open-source object-relational database that follows the SQL standard to a close degree [9]. While NoSQL alternatives were briefly considered, they were quickly discarded due to their generally greater complexity in comparison to SQL. Since BPFCAudit is intended for development purposes, the eventual scalability limitations of SQL databases were not seen as an important factor. No significant consideration was made about which SQL database to use. PostgreSQL was chosen due to its reliability, efficiency and since it is open source [9].

To define BPFCAudit's tables and perform operations against them, the Spring Data JPA library was used. By simply specifying a database flavour (PostgreSQL) and a connection string for this database in the application's configuration, Spring Data JPA can automatically generate the SQL necessary to create the tables for our database [10]. The database is initialized (if not already) based on this generated schema before the application runs. Spring Data JPA derives its generated database schema from annotated model classes which must specify any relationships to other models. As an additional benefit, Spring Data JPA repositories significantly cut down on boilerplate code by providing implementations for common database retrieval and persistence methods. Furthermore, the library is capable of automatically generating custom queries given only a function signature on a model's repository. This means that no SQL needs to be written and Spring Data JPA can take care of this for us. Spring Data JPA also provides a set of default repository implementations, one of which supports paging and sorting automatically. This was used in the case of the Service model, so that a limit can be imposed upon how many services are shown in the "view all services" GUI that is first discussed in Section 2.3.1.

2.2.2 Audit Flow

We now describe the audit API that is required by F2. That is, BPFCAudit's interface for collecting audit event data from live BPFContain containers. The collection of this audit event data must also be limited to a specific period so that the user can "fire and forget" an audit so that no further management of the audit is required after initiation. The collected audit event data must also pertain to a specific audit and thus a specific service. We tie audits to a specific service so that users can build a historical profile of behavior of their application. Audits are initiated via a POST request made to BPFCAudit with a timestamp indicating the exact time the audit should end, and the identifier of the service the audit is to be associated with. A sequence diagram of the audit process is shown in Figure 2.2.

Audit events occur in BPFContain whenever an action made by a container matches a rule specified by BPFContain. These audit events can occur for either rules in BPFContain's default policy or for rules imposed by the policy that is associated with a given container. BPFContain's daemon publishes these audit events using the WebSocket protocol whose data follows the json-rpc 2.0 specification [6]. To start receiving audit events, the json-rpc 2.0 compliant audit_subscribe message is sent to BPFContain which causes audit events to be sent over the socket. Optionally, a filter parameter is provided to the subscription request which can be used to ensure that only a specific subset of audit events are sent. This filter parameter however currently has no effect, but once implemented it would be used to ensure that only audit events that pertain to the given BPFCAudit service are collected. One proposed filter that could achieve this would be a process identifier (PID) filter. A PID could be passed to BPFContain's run command (which is used to contain a program with a certain policy) and ensure that the spawned container has this PID [11]. BPFCAudit could then use BPFContain's run command with a random PID and use this PID as a filter for audit events.

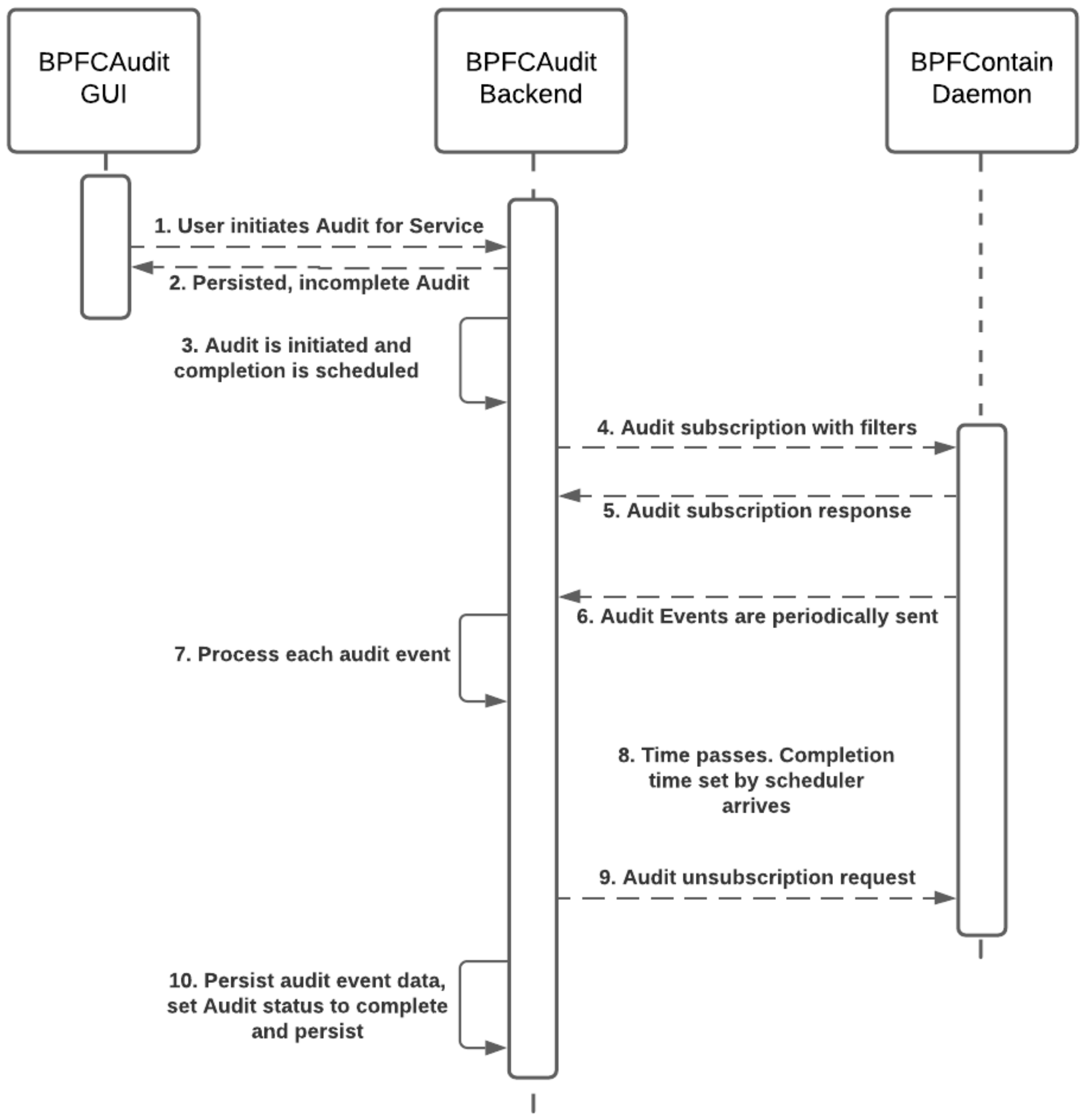

BPFCAudit GUI
BPFCAudit Backend
BPFContain Daemon
1. User initiates Audit for Service
2. Persisted, incomplete Audit
3. Audit is initiated and completion is scheduled
4. Audit subscription with filters
5. Audit subscription response
6. Audit Events are periodically sent
7. Process each audit event
8. Time passes. Completion time set by scheduler arrives
9. Audit unsubscription request
10. Persist audit event data, set Audit status to complete and persist

Figure 2.2: A sequence diagram representing the current implementation of audits (successful case only). Communications with the PostgreSQL database are omitted for simplicity.

Since filters for audit events are currently unimplemented, the invariant that only one audit can be run at a given time by BPFCAudit is enforced using a simple locking mechanism. This is to help enforce that an audit is associated with only one service. There is currently nothing stopping a user from running multiple BPFContain containers and subsequently running an audit which collects events from all these containers, so users must ensure that only one is running at a given time to ensure accurate audits. Thus, when an audit is “kicked off”, BPFCAudit first checks to ensure that no other audits are running, otherwise an HTTP bad request status is returned. Next, the audit is persisted with the status IN_PROGRESS. Audit statuses are used to represent the current state of the audit. This is used to clean up “dangling” audits upon startup of BPFCAudit, that is any audits with the IN_PROGRESS status can be deleted on startup as they should have completed before the program terminated.

After the audit is initially persisted, a worker thread is created which handles the ingestion of audit events from BPFContain’s WebSocket. A separate thread which triggers the audit’s termination and subsequent cleanup is created and added to a ScheduledExecutorService. The ScheduledExecutorService’s schedule function is used to run this audit termination thread once the audit’s end time arrives, ensuring that audit events are only captured for the user’s specified time [12]. Next, a WebSocket connection is established with BPFContain using the AsyncHttpClient library. The WebsocketListener class is implemented which specifies how the WebSocket behaves when it opens (the connection to BPFContain is established), when frames are received from BPFContain and more [13]. When the WebSocket connection is opened, the audit_subscribe message is sent to BPFContain to begin receiving audit events. Whenever BPFContain sends an audit event, the onTextFrame handler is run asynchronously.

Since the handling of audit events is asynchronous, we need thread safe methods of processing them. By processing audit events, we mean that we keep a running total of how many audit events we’ve seen for a given rule and preserve the attributes for at least one of these audit events for a given rule. It is only necessary to preserve one audit event for a given rule since subsequent audit events for a given rule do not have any additional data of interest for BPFCAudit. For this, a combination of the thread safe ConcurrentHashMap and LongAdder classes are used. ConcurrentHashMaps allows for full, non-blocking concurrent reads and highly concurrent updates [14]. The LongAdder class allows for concurrently adding to a numerical sum [15]. The LongAdder class’s documentation suggests using a LongAdder as a value for a

ConcurrentHashMap to concurrently track sums for each key, so we do this for tracking our sum of events seen for a given rule. For a key, the hash for each audit event is used for a given rule since BPFContain guarantees that this hash is the same for any audit event pertaining to a given rule [11]. Another ConcurrentHashMap is used to persist the attributes for the first audit event that is seen for a given rule.

After the audit's scheduled end time arrives, the audit's completion handling is invoked. This includes sending an audit_unsubscribe message to BPFContain to ensure proper cleanup of the socket, marking the audit's status as SUCCESSFUL and persisting the rule data, each of which has a sum indicating the frequency of the given rule's usage. If any exception occurs during the audit, the audit's status field is set to FAILED and a message indicating the reason for the audit's unexpected termination is indicated (based on the exception that occurs). No rule data is persisted for failed audits. Failed audits are persisted for visibility purposes as users may be confused if audits seemingly disappear.

2.2.3 Policy Analysis Flow

Currently, the policy analysis API and its accompanying logic are unimplemented in the BPFCAudit backend. The interface for producing rules from a policy is also not available yet from BPFContain, though it is expected to be easy to implement [11]. The policy analysis API is however mocked in the frontend. This API expects a policy's content and an identifier of the audit whose rule data should be used for the analysis as parameters. As a response, it returns an array of "policy lines" that indicate the line in the policy the data corresponds to (if any) and the number of audit events that occurred for this rule. The processing the backend would have to do for this API is illustrated by the sequence diagram in Figure 2.3.

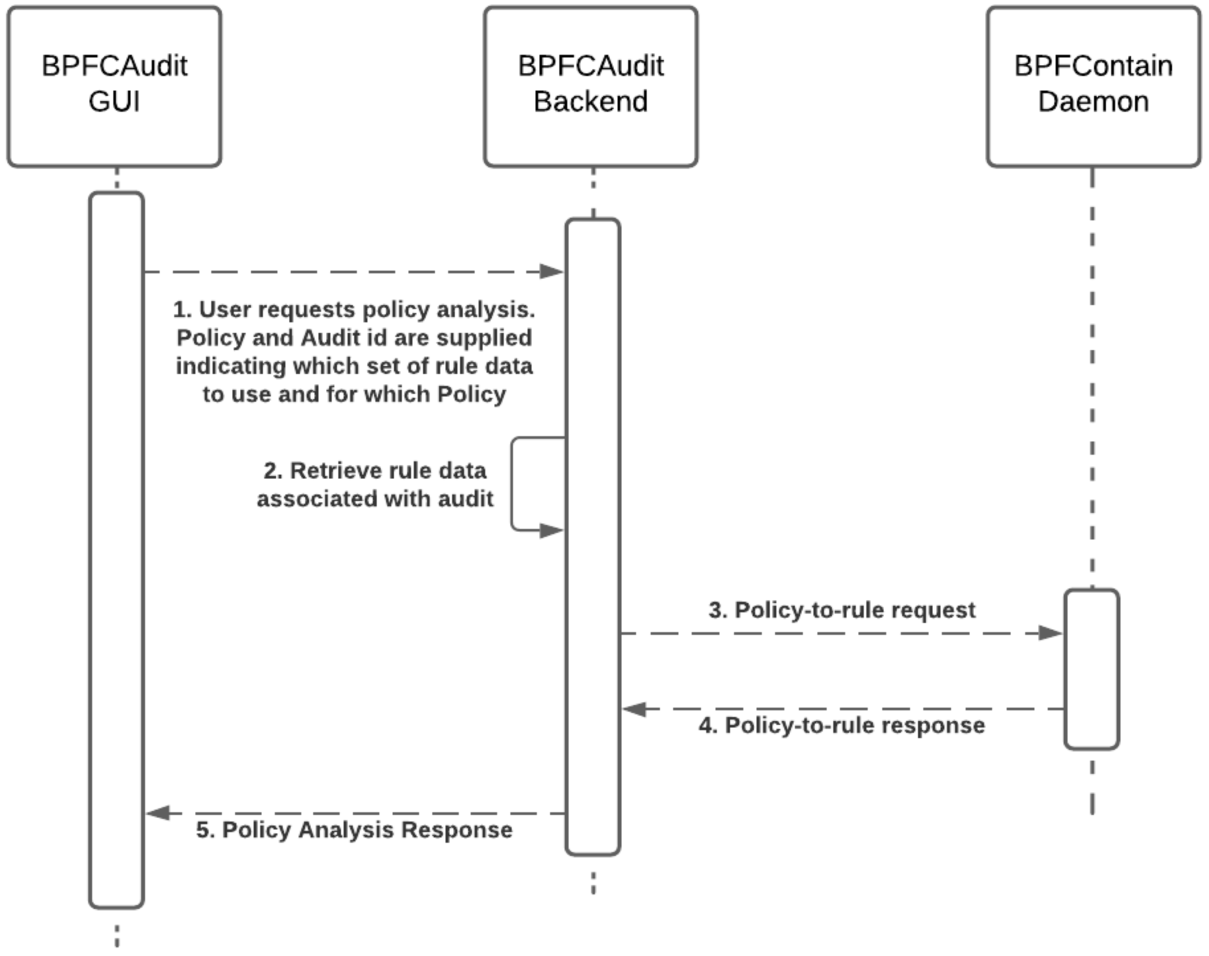

BPFCAudit GUI
BPFCAudit Backend
BPFContain Daemon
1. User requests policy analysis. Policy and Audit id are supplied indicating which set of rule data to use and for which Policy
2. Retrieve rule data associated with audit
3. Policy-to-rule request
4. Policy-to-rule response
5. Policy Analysis Response

Figure 2.3: Policy analysis sequence diagram. Communications with the PostgreSQL database are omitted for simplicity.

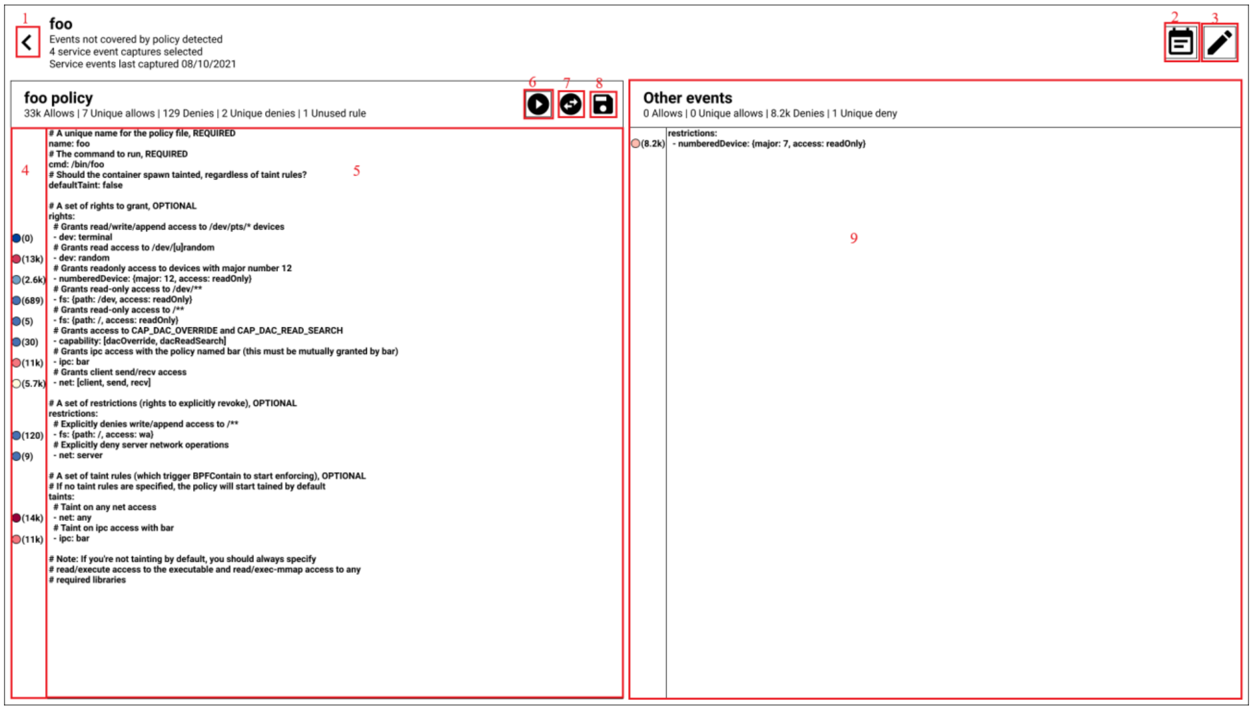
Using the audit identifier passed to the policy analysis request, BPFCAudit would retrieve all rules associated with the provided audit and simultaneously call BPFContain's policy-to-rule interface with the input policy. The policy-to-rule interface that BPFContain would expose would create rule objects for an input policy [11]. These rule objects returned by BPFContain would have a hash that could be compared to the hashes stored with the audit's associated rule data. With this comparison of hashes, BPFCAudit's policy analysis could determine if an audit's rules map to a line in the policy or not. Optionally, BPFCAudit could also summarize the resulting analysis by providing counts such as how many rule violations, and rule allows occurred, how many events are covered by the provided policy, and how many are covered by BPFContain's default policy. Events that are covered by BPFContain's default policy are those that do not have a matching rule hash in the response to the policy-to-rule interface from BPFContain.

2.3 Frontend

The primary responsibility of the BPFCAudit GUI is to implement an interface for displaying all captured rule data based on a policy and a set of audit logs, which allows for policy analysis (F1). There should also be supporting interfaces for this rule data interface such that the usability of the interface is sufficient (NF4). Of note, the user should be able to browse and select all services and select individual audits and policies to use for policy analysis. We now discuss the design of these interfaces, backing technologies, implementations, and limitations.

2.3.1 Design

Initial designs of the interface that displays all rule events and for the interface that displays all services were created in Figma. Figma is a user interface prototyping and design tool [16]. Figma was chosen as an interface prototyping tool because of its ability to model responsive layouts [16]. A responsive layout (or responsive design) is one that adjusts its layout and features according to the context in which it is being presented (e.g., it resizes to fit the size of the browser window in the case of presentation in a web browser) [17]. In addition, Figma allows for templating and re-use of interface components and for the ability to export layouts to CSS styling, both of which were used extensively while designing and implementing the interfaces [16]. These designs were not updated as the project developed since I was the sole developer of BPFCAudit, and a reference for development was not required. In Figure 2.4 we present the initial design for the "policy analysis" interface and annotate it to ease our discussion:



foo
Events not covered by policy detected
4 service event captures selected
Service events last captured 08/10/2021

foo policy
33k Allows | 7 Unique allows | 129 Denies | 2 Unique denies | 1 Unused rule

```
          # A unique name for the policy file, REQUIRED
          name: foo
          # The command to run, REQUIRED
          cmd: /bin/foo
          # Should the container spawn tainted, regardless of taint rules?
          defaultTaint: false

          # A set of rights to grant, OPTIONAL
          rights:
           # Grants read/write/append access to /dev/pts/* devices
●(0)       - dev: terminal
           # Grants read access to /dev/[u]random
●(13k)     - dev: random
           # Grants readonly access to devices with major number 12
●(2.6k)    - numberedDevice: {major: 12, access: readOnly}
           # Grants read-only access to /dev/**
●(689)     - fs: {path: /dev, access: readOnly}
           # Grants read-only access to /**
●(5)       - fs: {path: /, access: readOnly}
           # Grants access to CAP_DAC_OVERRIDE and CAP_DAC_READ_SEARCH
●(30)      - capability: [dacOverride, dacReadSearch]
           # Grants ipc access with the policy named bar (this must be mutually granted by bar)
●(11k)     - ipc: bar
           # Grants client send/recv access
○(5.7k)    - net: [client, send, recv]

          # A set of restrictions (rights to explicitly revoke), OPTIONAL
          restrictions:
           # Explicitly denies write/append access to /**
●(120)     - fs: {path: /, access: wa}
           # Explicitly deny server network operations
●(9)       - net: server

          # A set of taint rules (which trigger BPFContain to start enforcing), OPTIONAL
          # If no taint rules are specified, the policy will start tained by default
          taints:
           # Taint on any net access
●(14k)     - net: any
           # Taint on ipc access with bar
●(11k)     - ipc: bar

          # Note: If you're not tainting by default, you should always specify
          # read/execute access to the executable and read/exec-mmap access to any
          # required libraries
```

Other events
0 Allows | 0 Unique allows | 8.2k Denies | 1 Unique deny

```
          restrictions:
●(8.2k)    - numberedDevice: {major: 7, access: readOnly}
```

Figure 2.4: The initial design of the policy analysis page. Designed in Figma.

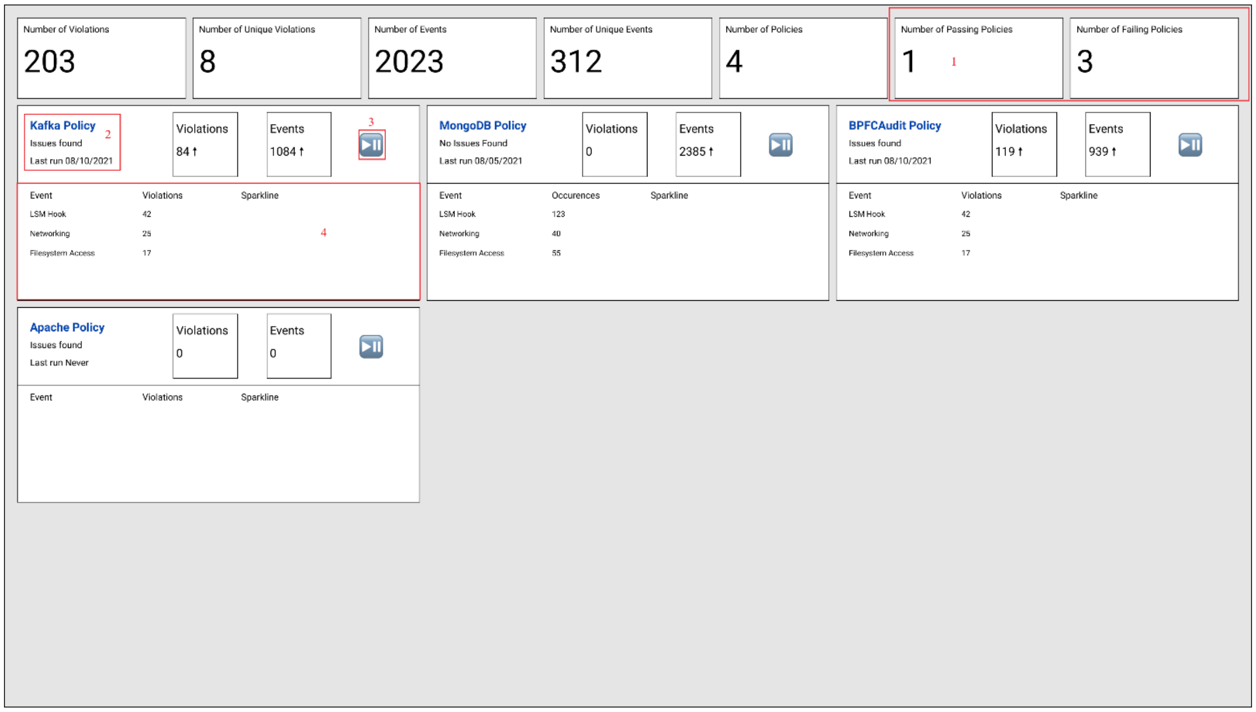
The purpose of the “policy analysis” or “policy heatmap” GUI design presented in Figure 2.4 is to satisfy F1. That is, this is the GUI that presents rule activations, and the user interacts with this interface to develop their policies and validate them. On this page, the user can select an audit to use for analysis with the currently loaded policy for the given service. They can also initiate audits for the service using the policy that is currently associated with the service. The implementation of this interface is discussed in Section 2.3.3. We provide some explanation of the individual elements of the design using the annotations that were made:

1. Represents a “back” button that takes the user back to the page where they selected this service from. This interface will be discussed later in this section.
2. Presents a modal that allows the user to select the audit to use in combination with the currently loaded policy for policy analysis.
3. Allows for the editing of the attributes of the service that this page represents.
4. The “policy heatmap”. For the rule of the policy that is directly adjacent to a given color and number, this represents how many times audit events occurred for this rule (the number) and the relative “strength” of the rule. The strength being how many times the

rule was utilized in comparison to other rules. Warmer colors (oranges, reds) indicate heavier usage of the rule compared to colder colors (purples, blues).

5. Represents the “raw” BPFContain policy that is currently associated with this service. It was planned that the user could edit this policy directly in this interface and later save it.
6. Presents a modal that allows the user to schedule an audit for this service.
7. Presents a modal that allows the user to change the policy that is currently associated with this service.
8. Saves the currently loaded policy to the system.
9. Represents rules that had at least one triggering (i.e., restrictions or allows) from the default BPFContain policy. This indicates rules that the user should add to their policy.

We now present the initial design for the interface that displays all services. Similarly to the policy heatmap interface, we annotate this design in Figure 2.5:



Number of Violations 203

Number of Unique Violations 8

Number of Events 2023

Number of Unique Events 312

Number of Policies 4

Number of Passing Policies 1

Number of Failing Policies 3

Kafka Policy
Issues found
Last run 08/10/2021
Violations 84 ↑
Events 1084 ↑

Event	Violations	Sparkline
LSM Hook	42	
Networking	25	
Filesystem Access	17	

MongoDB Policy
No Issues Found
Last run 08/05/2021
Violations 0
Events 2385 ↑

Event	Occurences	Sparkline
LSM Hook	123	
Networking	40	
Filesystem Access	55	

BPFCAudit Policy
Issues found
Last run 08/10/2021
Violations 119 ↑
Events 939 ↑

Event	Violations	Sparkline
LSM Hook	42	
Networking	25	
Filesystem Access	17	

Apache Policy
Issues found
Last run Never
Violations 0
Events 0

Event	Violations	Sparkline

Figure 2.5: The initial, rough design of the service (policy) selection page. Notably, the service abstraction did not exist at the time of this design, so individual policies are represented on this page rather than services. Designed in Figma.

The purpose of the service selection GUI design presented in Figure 2.5 is to allow the user to select a service to perform policy analysis against. It should be noted that individual policies are modeled by this design, and not individual services. This is because the service abstraction did not exist at the time of this design. The spirit of this design remains the same however in that it provides some "system-level" metrics and allows the user to access policy analysis. The implementation of this interface is discussed in Section 2.3.4. We provide some explanation of the individual elements of the design using the annotations that were made:

1. Some of the "system-level" metrics. "Passing policies" indicates how many services have audit events that are fully covered by the given policy. "Failing policies" indicates the opposite.
2. The policy's name and a high-level summary of the last-run policy analysis. With more current designs, this would instead be the service's name.
3. Allows for the initiation of an audit for this policy or for the cancellation of the currently running audit. Again, with more current designs, this would run an audit against the service with the currently associated policy.
4. Represents a "snapshot" of the rules that had the highest occurrence of audit events. The sparkline column was meant to be a small chart representing audit events over time for this rule, but this element was quickly discarded due to the expected complexity of the implementation.

2.3.2 Technologies

To implement our designs, a web-based GUI (one that can be accessed by a web browser) was favored over a desktop GUI (one that must be installed directly onto a system). This decision was made primarily with portability in mind. Since web applications can be accessed from any browser which is independent of the host operating system, this removes the extra work involved with ensuring that the GUI works on different operating systems [18]. Although BPFCAudit is currently only intended for use on Linux (since BPFContain is Linux-only), if a policy-driven containerization framework was developed that worked on other OSes, then the BPFCAudit GUI could be re-used for these frameworks, supporting NF3 [18]. I also perceived a web application as being easier to develop due to my familiarity with the space as opposed to desktop GUIs with which I have very limited experience.

An important decision to make before the development of most web GUIs is what frontend framework to use (if any). Frontend frameworks help to provide structure to a web app, in that they often dictate where individual components of the app should be and how they interact with other components. Frameworks often break up these components into distinct categories promoting separation of concerns and providing for an easy-to-understand application structure [19]. Because of this, the resulting application is often easier to maintain since individual components are better isolated. Some frameworks also increase developer agility by providing interfaces to quickly generate components, skipping the need to write boilerplate code [19].

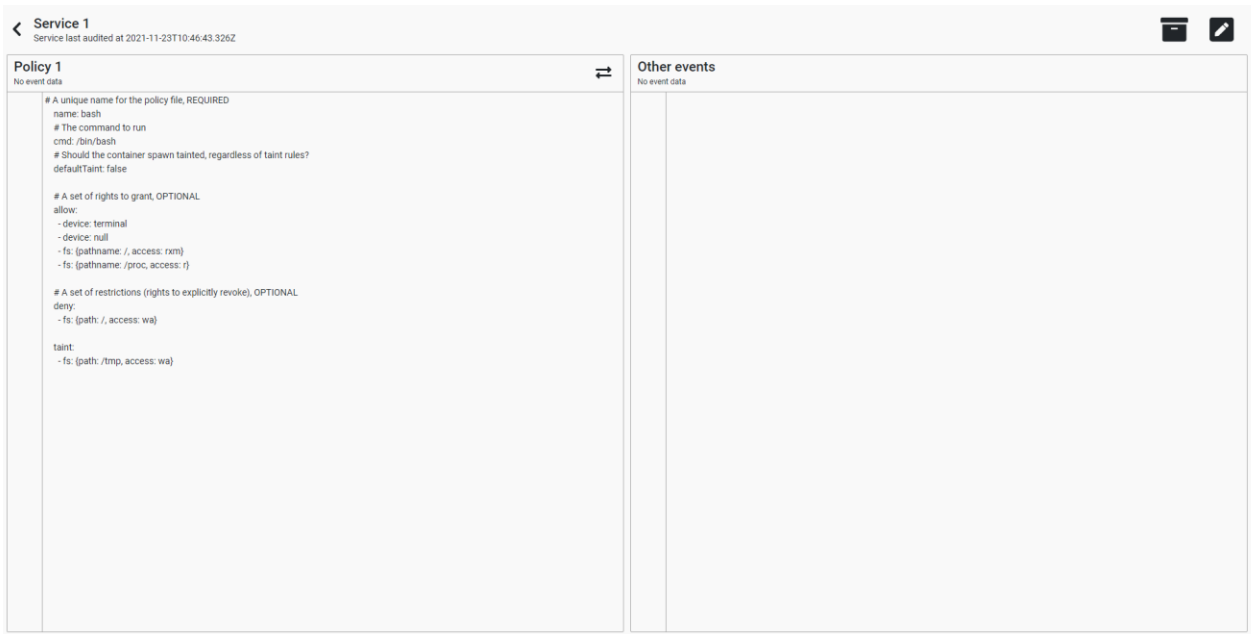
The frontend framework that was chosen for BPFCAudit’s web GUI was Ember. Ember has an excellent track record with respect to backwards-compatibility and upgradability [20]. This was seen as highly desirable as some other web frameworks have been known to require significant time investments from maintainers when upgrading. I also had prior experience working with the Ember framework which made it a clear choice since I could develop faster.

A feature of Ember that saved me a significant amount of time was Ember’s command line interface (CLI). Ember’s CLI includes the ability to generate the boilerplate for new entities on-command and can create a basic application from scratch which includes a development environment [20]. This default development environment provides a test runner and the ability to auto-reload (the app automatically serves new content when changes are made without the need to rebuild the whole app), both of which provided significant time savings since I did not have to do this configuration myself [20]. Ember Data, Ember’s solution for managing data models, their relationships and how they are retrieved also provides out-of-the-box json:api support [21]. This motivated the choice to make BPFCAudit’s APIs conform to json:api since Ember is capable of automatically normalizing responses that follow this specification (turning response data into objects the GUI can use) [21]. An Ember feature that I used extensively during development is the Ember Inspector. The Ember Inspector is a browser extension that allows for the debugging of Ember applications [22]. Ember Inspector can be used to inspect the values of Ember Data models, view the details of individual components and to see the route tree of the app [22].

Some supporting technologies that I used to assist in the development of BPFCAudit were TypeScript (TS) and Sass. TS is a superset of JavaScript (JS) which provides strong typing capabilities [23]. TS allows for static type verification, that is TS can identify certain type errors before the program is run [23]. This contrasts with JS where these issues are often identified at runtime which results in more time spent debugging. Since TS is a superset of JS, TS is interoperable with JS which means JS code can be called by TS files which allows us to leave some files in JS where typing provides little benefit. Sass is a cascading style sheet (CSS) preprocessor meaning that Sass files must be interpreted and output to CSS for use in the resulting web page [24]. Sass reduces repetition by introducing variables to style sheets, modules, and inheritance among other features [24]. Sass also introduces a much more readable visual hierarchy through the nesting of CSS selectors [24].

2.3.3 Policy Heatmap

We now present how the look and feel and feature set of the policy heatmap page has changed since its initial design in Section 2.3.1. We also discuss the actions the user can take on the page whose layouts were not considered in the initial design. All data shown in this section is mocked using mirage.js. Mirage.js enables the testing of JavaScript apps without requiring live backends by mocking these APIs [25]. The layout of the policy heatmap page upon first visiting a given service is shown in Figure 2.6:

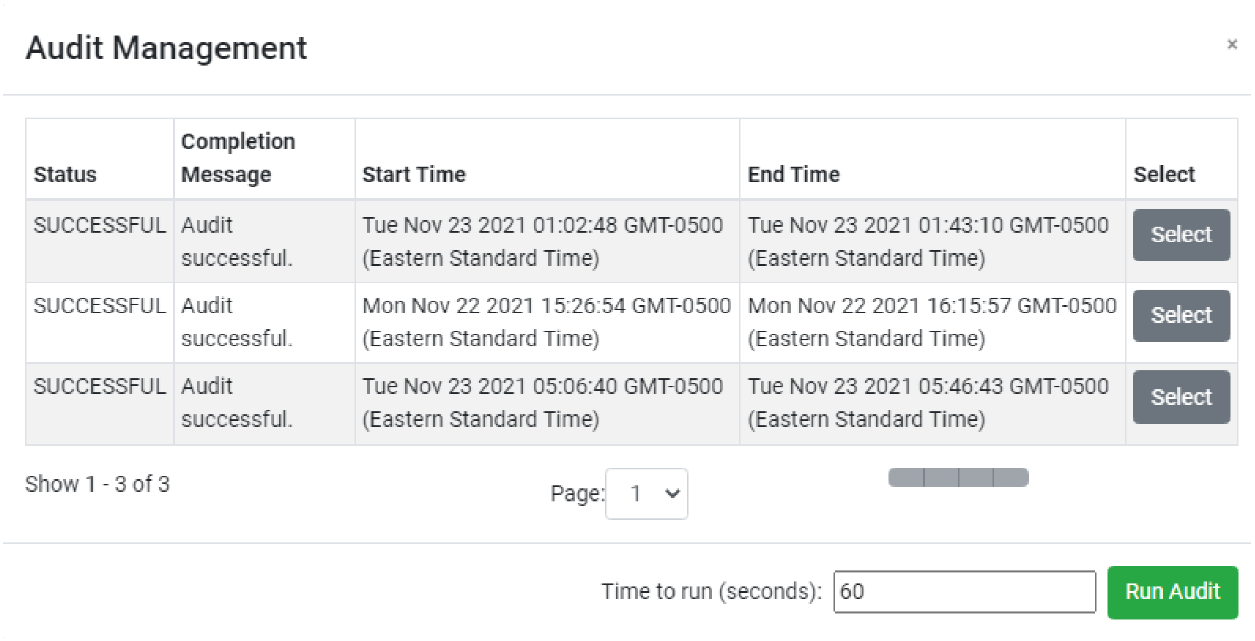


```
# A unique name for the policy file, REQUIRED
  name: bash
  # The command to run
  cmd: /bin/bash
  # Should the container spawn tainted, regardless of taint rules?
  defaultTaint: false

  # A set of rights to grant, OPTIONAL
  allow:
   - device: terminal
   - device: null
   - fs: {pathname: /, access: rxm}
   - fs: {pathname: /proc, access: r}

  # A set of restrictions (rights to explicitly revoke), OPTIONAL
  deny:
   - fs: {path: /, access: wa}

  taint:
   - fs: {path: /tmp, access: wa}
```

Figure 2.6: The initial look of the policy heatmap page upon first visiting (no audit is selected).

Note that no "heat indicators" adjacent to the rules in the policy are present due to no audit being selected. An audit must first be selected to perform policy analysis since a set of audit event data must be present. Notable changes from the initial design of this page include the removal of buttons related to live policy editing, the merging of the audit creation button with the audit selection button (represented by the "archive" button in the top right of Figure 2.6). Live policy editing was omitted from the first iteration of the application due to complexity. Some metadata shown under the service name was also removed due to the perceived complexity of ensuring that this data is consistent (e.g., if the latest audit is deleted then all of these metadata properties associated with the service would have to be recomputed).

To display a policy analysis, an audit must first be selected for the service so that rule data can be provided to the analysis. To select an audit, the audit management button is selected (represented by the "archive" button in the top right of Figure 2.6). From here (depicted in Figure 2.7), the user can view audits associated with a service and select them. Upon selection of a service, a policy analysis is performed with the currently associated policy and the selected audit. In addition, the user can initiate new audits for the service for a period from the audit management modal.



Audit Management

Status	Completion Message	Start Time	End Time	Select
SUCCESSFUL	Audit successful.	Tue Nov 23 2021 01:02:48 GMT-0500 (Eastern Standard Time)	Tue Nov 23 2021 01:43:10 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Mon Nov 22 2021 15:26:54 GMT-0500 (Eastern Standard Time)	Mon Nov 22 2021 16:15:57 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Tue Nov 23 2021 05:06:40 GMT-0500 (Eastern Standard Time)	Tue Nov 23 2021 05:46:43 GMT-0500 (Eastern Standard Time)	Select

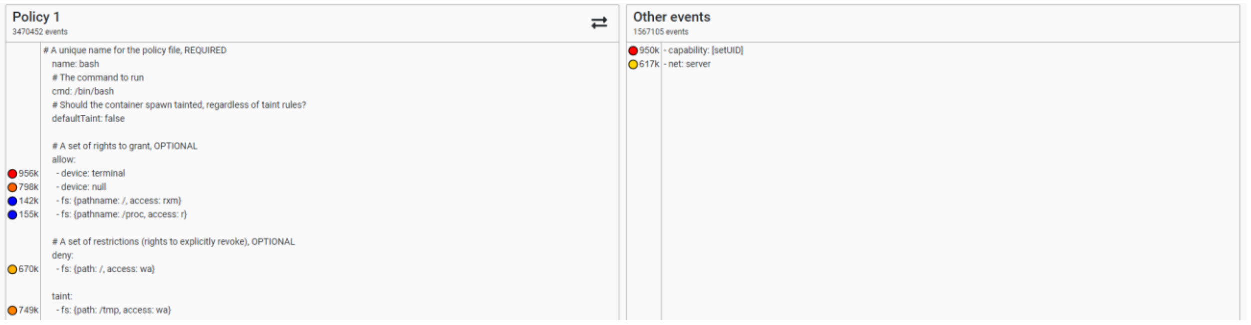
Show 1 - 3 of 3

Page: 1

Time to run (seconds): 60

Figure 2.7: The audit management modal. Here users can view audits, select them, and initiate new audits for the service. Note that the grey bar below the table should have arrow icons for navigating the pages of audit (if any) and this issue is under investigation. How these icons should present can be seen on the ember-models-table demo page, the component used to render this table [26].

After an audit is selected, policy analysis automatically occurs with the audit event data associated with the selected audit. The resulting policy analysis after selecting an audit is presented in Figure 2.8:



Policy 1
3470452 events

```
            # A unique name for the policy file, REQUIRED
              name: bash
              # The command to run
              cmd: /bin/bash
              # Should the container spawn tainted, regardless of taint rules?
              defaultTaint: false

              # A set of rights to grant, OPTIONAL
              allow:
🟥 956k        - device: terminal
🟧 798k        - device: null
🟦 142k        - fs: {pathname: /, access: rxm}
🟦 155k        - fs: {pathname: /proc, access: r}

              # A set of restrictions (rights to explicitly revoke), OPTIONAL
              deny:
🟨 670k        - fs: {path: /, access: wa}

              taint:
🟧 749k        - fs: {path: /tmp, access: wa}
```

Other events
1567105 events

```
🟥 950k - capability: [setUID]
🟨 617k - net: server
```

Figure 2.8: The policy currently associated with the service annotated with data (the colour and numerical indicators next to lines in the policy that represent rules) from the policy analysis.

It is also possible to change the policy that is currently associated with the service on the policy heatmap page. This is done by selecting the policy change button which is present in the header of the policy (represented by the two horizontal arrows in opposite directions as seen in

Figure 2.6). The resulting modal is presented in Figure 2.9. When the policy is changed, a policy analysis is automatically performed for the newly loaded policy and the currently selected audit (if any).

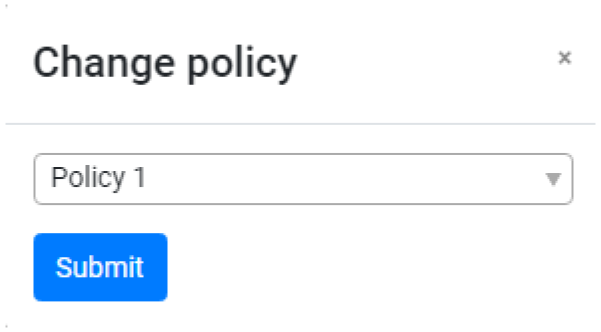


Figure 2.9: The Change Policy modal. Upon opening this modal, a GET request is made to retrieve all available policies which are then presented in the resulting drop-down selector. The “Submit” button makes a PATCH request to update the policy relationship of the service.

The request for the current service (which includes the associated policy and audits) is polled at a cadence of 30 seconds. This ensures that if another user updates the service’s attributes, runs a new audit, or otherwise modifies the resources shown in the page, the user is eventually shown the updated state of the application. To update the service’s attributes the “Update Service” button (the pencil icon in the top right of Figure 2.6) can be selected. The resulting service update modal is presented in Figure 2.10.

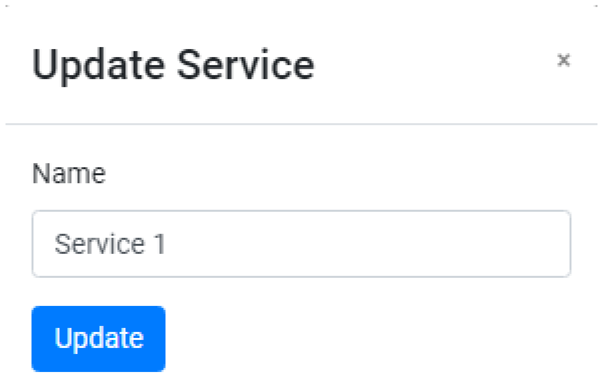
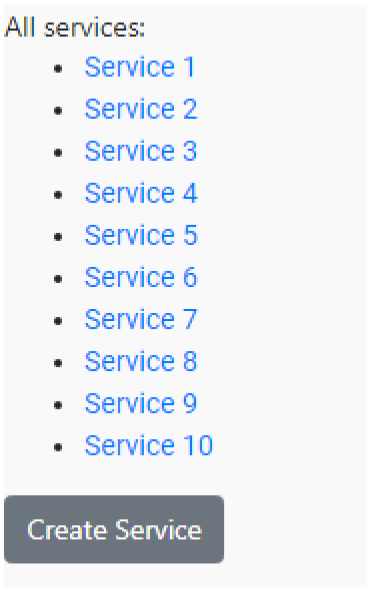


Figure 2.10: The modal that is used for updating a service’s attributes. When the “Update” button is clicked, a PATCH request is sent to the backend to perform this update.

2.3.4 Overview of Services

Due to time constraints, only a very simple HTML list (<ul> and <li> tags) of links were used to present all services, in contrast to the "card" design originally presented in Section 2.3.1. This page was seen as a lower priority item since its main purpose is to link the user to the policy analysis page for a given service and the density of information on this page matters less with respect to promoting policy development. The current state of this page is shown in Figure 2.11:


All services:

- Service 1
- Service 2
- Service 3
- Service 4
- Service 5
- Service 6
- Service 7
- Service 8
- Service 9
- Service 10

Figure 2.11: The content contained in the overview of services page. Displays a list of all services and a button for creating a new service.

Each link in the list of services page routes the user to the associated policy heatmap page for the given service on click. Similar to the policy heatmap page, the API for fetching all services is polled at an interval of 30 seconds, ensuring that if a service is created, deleted or its name is updated by some other user or through a manual request, the user will eventually see this updated information. On click, the "Create Service" button presents the modal shown in Figure 2.12:

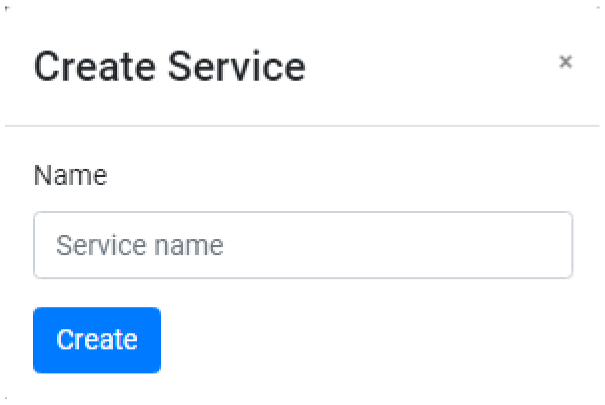


Figure 2.12: The modal that is presented when the "Create Service" button is clicked. When the "Create" button is clicked, a POST request is sent to the backend to create a new service with the provided name. This modal uses the same underlying component as the modal presented in Figure 2.10.

Of note, there is currently no way to delete a service (this API also does not yet exist on the backend). This could be easily added as a button to individual service "cards" once the implementation of this page more closely follows the original design. It is likely that some form of pagination will also be necessary as the number of services grows (although it is expected that a user will only create a few services).

3 Results

In this section, we perform an analysis of the efficacy of BPFCAudit, particularly with respect to some non-functional requirements such as performance. Furthermore, we present the methodology and results of these analyses. We also present some discussion of how these results support BPFCAudit's goals of policy development and verification.

3.1 Audit Performance Analysis

A critically important requirement that determined the feasibility of this project is NF1, or BPFCAudit's efficiency. This is especially relevant with respect to the audit API, whose details are discussed in Section 2.2.2. The reason that the performance of this API is so critical is that BPFCAudit should be able to process audit events as quickly as BPFContain can produce them. If BPFCAudit processes only a portion of the events that BPFContain publishes, then it is possible that the diversity of rule data will be lower in the final policy analysis. While less concerning, it would also mean that the final count of events for a given rule could be inaccurate. This is less of a concern since the count of rules is just extra information that we provide to the user, and in most cases does not directly influence policy development (a rule should either be in the policy, or it should not – there is no middle ground). We now discuss the methods that were used to characterize BPFCAudit's ability to process events sent by BPFContain, and the results of the various methods for processing whose performances were characterized.

3.1.1 Methodology

To benchmark BPFCAudit's performance with respect to its ability to process audit events sent by BPFContain, we are going to compare how many audit events BPFCAudit can process in comparison to how many are sent by BPFContain. This is the chosen metric since BPFCAudit is not intended to be deployed on production systems and only on development systems, so the resources that BPFCAudit uses itself are seen as less critical. These tests were conducted on an Ubuntu virtual machine running a standard version 5.11.0-38 kernel (with slight configuration changes to get BPFContain running). The resources provisioned for this virtual machine are depicted in Table 3.1:

Table 3.1: Allocation of hardware for Ubuntu virtual machine used for performance benchmarks.

Component	Specification
Kernel	5.11.0-38
CPU	Intel i5-6600K @3.50GHz (4 CPUs) (No execution cap, all CPUs allocated)
Memory	16 GB DDR4 (8GB allocated)
Disk	Samsung 970 EVO Plus 1TB (50GB allocated)
GPU	NVIDIA GTX 1070

To sufficiently test BPFCAudit's ability to ingest the audit events that BPFContain publishes, we need a way to produce a large quantity of BPFContain audit events quickly. Since audit events correspond with rules being triggered, it was necessary to develop a BPFContain policy that restricts or allows some specific rule, and then develop a program that causes repetitious triggering of this rule. The policy that was created (herein referred to as the "perfTest" policy) can be found in Appendix A. The only rule in this policy is a filesystem restriction which prevents read-mode access to a specific file on the VM. Of note, this policy is set to complaining mode which as we know does not deny restricted operations, but rather just logs them. This is done because it is expected that BPFCAudit will be used with policies set to complaining mode so that the normal code paths of applications can be fully exercised. Policies that are set to complaining mode often result in the triggering of more audit events for a given rule since they exercise different code paths [11]. In this case, a single read-mode opening of the file covered by the policy resulted in 30 audit events.

It was also necessary to develop a program that very quickly causes triggering of the rule specified in the perfTest policy. This program can be found in Appendix B. This program was written in C since it is a low-level, systems language known for its efficiency. We refer to this program as "eventSpam". It was also desirable to use C's simple wrapper around the open system call rather than risk using some other high-level language doing some other operations alongside this, potentially triggering more audit events. The main thing this program does is spawn a configurable number of threads, each of which opens a specific file rapidly in read mode for a provided amount of time. Additionally, each thread updates a counter every time it opens the file and once the specified amount of time has passed, the parent thread destroys these worker threads and sums the result from each thread into a single count. This count indicates how many times the file was opened. To get the true amount of audit events that were triggered, we must multiply this amount by 30 since this is the number of events a read-mode file open causes with a container that is in complaining mode.

Finally, a test orchestration bash script was created to ensure relatively consistent testing conditions across runs. This test orchestration script can be found in Appendix C. First, the script restarts the BPFContain daemon. A modified version of BPFContain was used to test BPFCAudit's performance which exposes an interface that reports how many audit events have been processed since the daemon started using a global counter. Hence, we must restart the daemon to reset this global counter between runs. The intention of this counter is to allow us to verify that the number of events published by BPFContain aligns with how many audit events occurred. The script then initiates a BPFCAudit audit using a POST request which collects audit events for a configurable amount of time (we always use two minutes). Finally, a BPFContain container using our perfTest policy is spawned which contains our eventSpam program. Our eventSpam program is also run for a configurable amount of time, (we always produce events for 15 seconds) using a configurable number of threads.

3.1.2 Audit Performance Results

The first method for processing audit events that was implemented was an approach where an object was instantiated for each audit event received and then soon after persisted using a JPA repository [27]. This saved data was then periodically flushed (committed to the database) in batches of 1000. Saved audit data is also flushed when the audit completes to ensure that all saved audits are committed to the database upon completion. No significant attempts were made to optimize this approach prior to testing. We present our findings in Table 3.2 with this approach using a few different counts of event producer threads.

Table 3.2: Total events produced, published by BPFContain, and captured by BPFCAudit in a two-minute period, using the "persist immediately" method of audit event capture. Events were produced for 15 seconds with a varying number of threads.

Threads	Event Count	Count of Events Sent by BPFContain	Count of Events Captured by BPFCAudit	% Of Events Sent by BPFContain Captured by BPFCAudit
1	99586290	3509844	22642	0.65%
2	145549950	3441138	31223	0.91%
3	140526000	3356742	21024	0.63%
4	145675830	3396958	29528	0.87%

We can make some observations about our results in Table 3.2. Clearly, the number of producer threads does not influence how many audit events BPFContain is able to send in the two-minute audit period. This is best illustrated by Figure 3.1. Not all audit events that were produced are sent from BPFContain because the audit event ring buffer's tail gets overwritten with new events before they are processed [11]. The synchronous nature of how audit events are processed in BPFContain can be partly attributed to this inability to publish all audit events [11]. Furthermore, the number of producer threads appears to not provide any additional benefit with respect to producing more audit events beyond two threads, though this is less relevant for our testing since BPFContain is a bottleneck with respect to the number of audit events BPFCAudit receives.

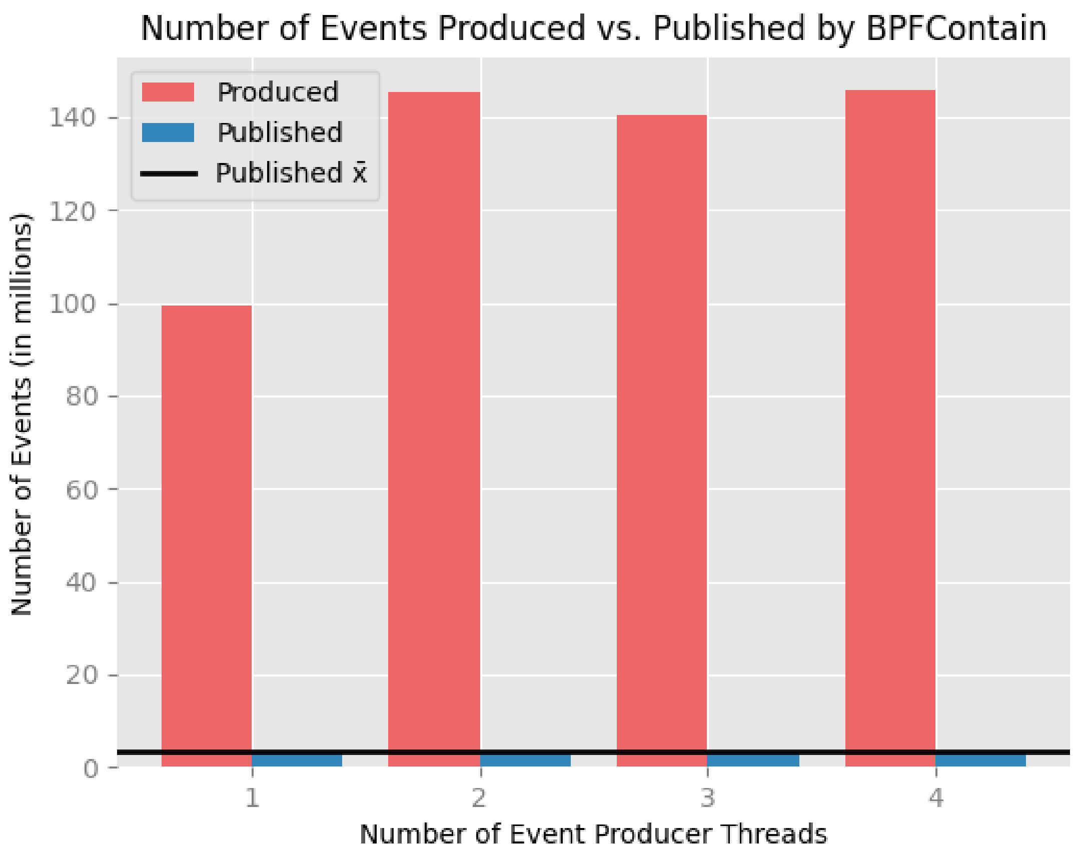


Figure 3.1: A comparison of how many audit events BPFContain produces and how many it publishes for a varying amount of event producer threads. This highlights that BPFContain is only able to publish a fixed amount of audit events during the two-minute production period, regardless of how many events are produced.

In addition, we see that in all four trials the percentage of events that BPFContain sends which BPFContain persists is less than one. It is believed that the slow processing on the BPFCAudit side is partly due to the reflection required to create each audit event object from a given audit event payload from BPFContain. Reflection in Java allows for the runtime inspection of classes, methods and more at runtime, which in this case is used to map the received JSON data into rule objects [28]. It is also possible that using Spring Data JPA to “enqueue” audit events to persist and then committing the audit events in this queue every 1000 events is an improper approach, or that the library needed to be differently configured to support this high-volume use case. Regardless, due to perceived complexity and the unnecessary nature of persisting each audit event as an individual object, this approach was not pursued further.

The second and final approach that was taken for ingesting audit events was the “hash and tally” approach. Since each audit event from BPFContain has a hash field which uniquely identifies each rule, we can “bucketize” each rule by hash. We do note that BPFContain audit event hashes are not yet implemented, so we use a static hash for each audit event. In doing this, we can keep track of how many audit events occurred for a given rule, while sacrificing the granularity of being able to persist individual audit events. Given our functional requirements, there was no need for this level of granularity with respect to audit events and this approach was favoured due to its simplicity and ease of implementation. We present our findings in Table 3.3 with this approach only using two producer threads for our eventSpam program since we know that this cannot impact our findings due to BPFContain “bottlenecking” how many audit events are sent.

Table 3.3: Total events produced, published by BPFContain, and captured by BPFCAudit in a two-minute period, using the “hash and tally” method of audit event processing. Events were produced for 15 seconds with two event producer threads.

Event Count	Count of Events Sent by BPFContain	Count of Events Captured by BPFCAudit	% Of Events Sent by BPFContain Captured by BPFCAudit
138250860	3436073	3436073	100%

As seen in Table 3.3, BPFCAudit can process audit events as quickly as BPFContain can produce them, satisfying NF1 in this case. It should however be noted that BPFContain was only able to send a small fraction of the audit events that were produced (~2.5%). Thus, if improvements were made to BPFContain’s ability to produce audit events in a timely manner, these performance tests would have to be conducted again. In the interest of further characterizing BPFCAudit’s ability to ingest audit events, a server that could quickly produce mock event data was considered for implementation and further testing. However, this was not pursued in the interest of further feature development. As a prototype, it is likely that BPFCAudit would also be serviceable enough in this regard as it is unlikely that many applications could produce audit events as quickly as our eventSpam program does.

3.2 Efficacy of Policy Heatmap

It is somewhat difficult to determine the usability (NF4) of BPFCAudit's frontend without conducting usability testing. Usability testing can be used to evaluate how easy to navigate, easy to use and how "intuitive" a user interface is [29]. We do not consider usability testing due to the difficulty associated with recruiting usability testers, and the time and care needed to collect and interpret these results without bias. One alternative to usability testing to determine some facet of usability is considering how responsive the GUI is. A responsive layout is one that adjusts its layout and features according to the context in which it is being presented [17]. This means that all elements in the BPFCAudit GUI should scale nicely and be displayed in an organized and logical manner regardless of how the user resizes the page, to ensure that the page's usability is not compromised when this is done. To characterize the performance (NF1) of the frontend, we can examine how quickly pages load (render time) and behave under various conditions. Although render time is seen as less important to BPFCAudit since it is intended for development purposes, we still expect some degree of speed to not frustrate users. We only consider the policy heatmap page (Section 2.3.3) in our analysis due to the overview of services page (Section 2.3.4) being unfinished and not currently representing its initial design (Section 2.3.1).

3.2.1 Methodology

To characterize the render time of the policy heatmap page, we are going to examine how the page behaves under "normal" or expected amounts of data and under more extreme conditions. To measure render time, the Ember Inspector browser extension can be used [22]. The "Render Performance" tab of this extension can be used to view the time it takes to render an entire page, or on a more granular level to view the render performance of individual components [30]. It is noted in the documentation of the "Render performance" module of this extension that using the inspector adds a delay to the resulting rendering time [30]. Because of this, the rendering times that we obtain from this extension are very much "upper bounds" on rendering times. In addition, all tests are conducted using the content resulting from "ember serve –environment=production”. This instructs the Ember CLI to process the app’s files into a format that can be rendered by browsers [31]. By using the “–environment=production” flag, we instruct the Ember CLI to minify the resulting files for maximum performance. Since many APIs that are required by the frontend have not yet been implemented by the backend, we are going to use data mocked via mirage.js in our testing.

To determine the responsiveness of the policy heatmap page, we simply resize the page to common, but arbitrary resolutions and subjectively evaluate how well elements of the page can resize. To resize the page to exact resolutions, we use the Window Resizer browser extension [32]. These tests were conducted on a Windows 10 system in a Google Chrome browser. The specifics of this system and the web browser used are depicted in table Table 3.4:

Table 3.4: Hardware and web browser specification used for frontend testing.

Component	Specification
CPU	Intel i5-6600K @3.50GHz (4 CPUs)
Memory	16 GB DDR4
Disk	Samsung 970 EVO Plus 1TB
GPU	NVIDIA GTX 1070
Browser	Google Chrome 96.0.4664.45 (No browser extensions except the previously discussed Ember Inspector and Window Resizer)

3.2.2 Performance Results

To sufficiently test the rendering performance of the policy heatmap page, we are first going to identify which elements on the page scale with the size of the data that these elements display. On the policy heatmap page, the policy and its associated heat indicators scale linearly with the size of the policy. In the audit management modal originally presented in Figure 2.7, the number of audits that can be viewed scales linearly with the number of audits associated with the service. To test the rendering speed of the policy heatmap page, we are going to scale the size of the policy by simply repeating its content. The policy we duplicate and use as the resulting policy is the httpd policy located in the BPFContain repository [33]. We present the rendering performance of the page under various amounts of duplication of this policy averaged across five trials in Table 3.5:

Table 3.5: Average render time of policy heatmap with various policy sizes.

Times policy duplicated	Number of lines in policy	Number of resulting heat indicators	Average render time of policy heatmap over 5 trials (ms)
1	68	35	58.08
5	340	167	88.72
25	1700	827	230.72

Judging from the results in Table 3.5, render performance does indeed decrease as the size of the policy increases. We can visualize this finding by plotting the size of the policy against render time in Figure 3.2. According to Google, two seconds is the maximum amount of time an e-commerce website should take to load [34]. These render times are still well below a second, which is below Google's threshold of acceptance. It should also be noted that our interface is not an e-commerce application, and we can probably expect more tolerance for slow page loads since our application is intended for development. Once the backend API for policy analysis is implemented, we will have to take the time it takes to complete this request into consideration as part of the page's load time as well.

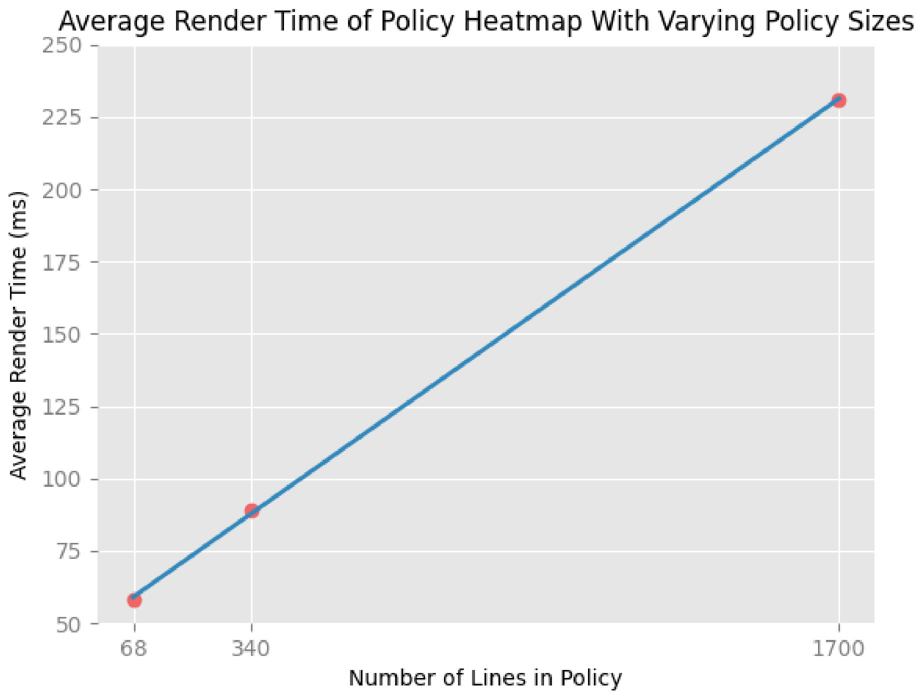


Figure 3.2: Average render time of the policy heatmap page as the size of the policy increases. Average render times are plotted, and a line of best fit is given.

Next, we present the rendering performance of the Audit Management modal under various amounts of audits in Table 3.6:

Table 3.6: Average render time of audit management modal with various amounts of audits.

Number of audits	Average render time of audit management modal over 5 trials (ms)
10	65.86
100	65.32
500	78.3

Based on the results in Table 3.6, we see that the render time of the audit management modal increases to some extent as the number of audits increases. It is suspected that render time maintains a relatively consistent rate as the number of audits increases thanks to the pagination that is built into the table that displays audits. This limits the number of audits that are rendered at a time. It is not expected that this modal would take longer to load if the interface was integrated with a live backend since the request to get the list of audits is performed before this modal is presented.

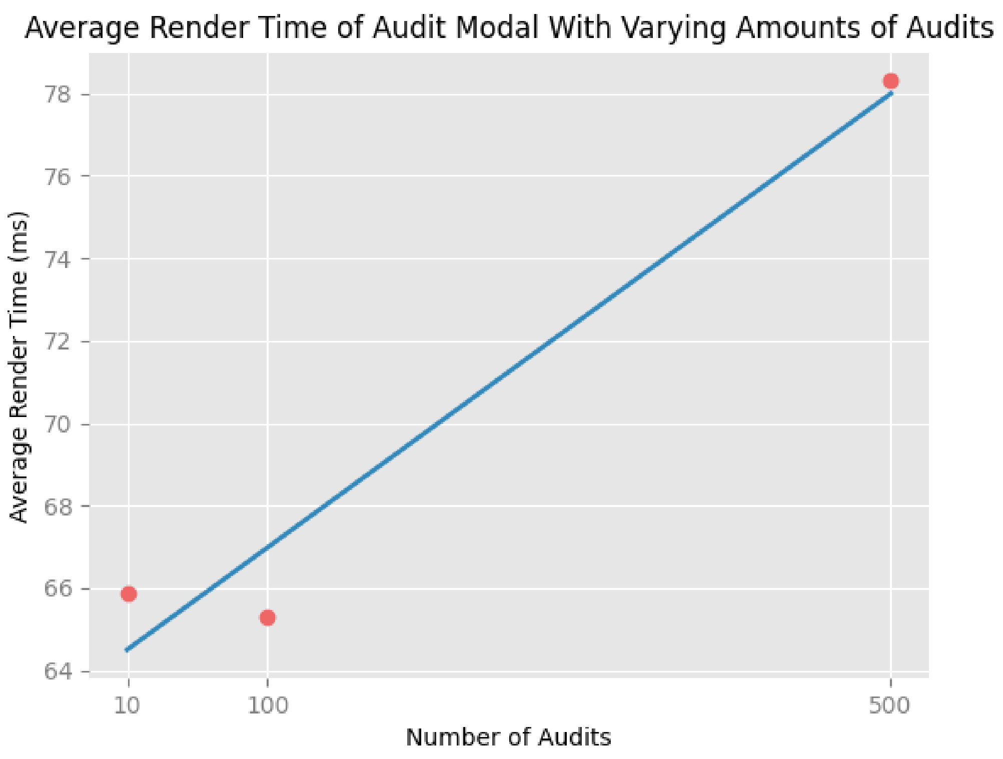
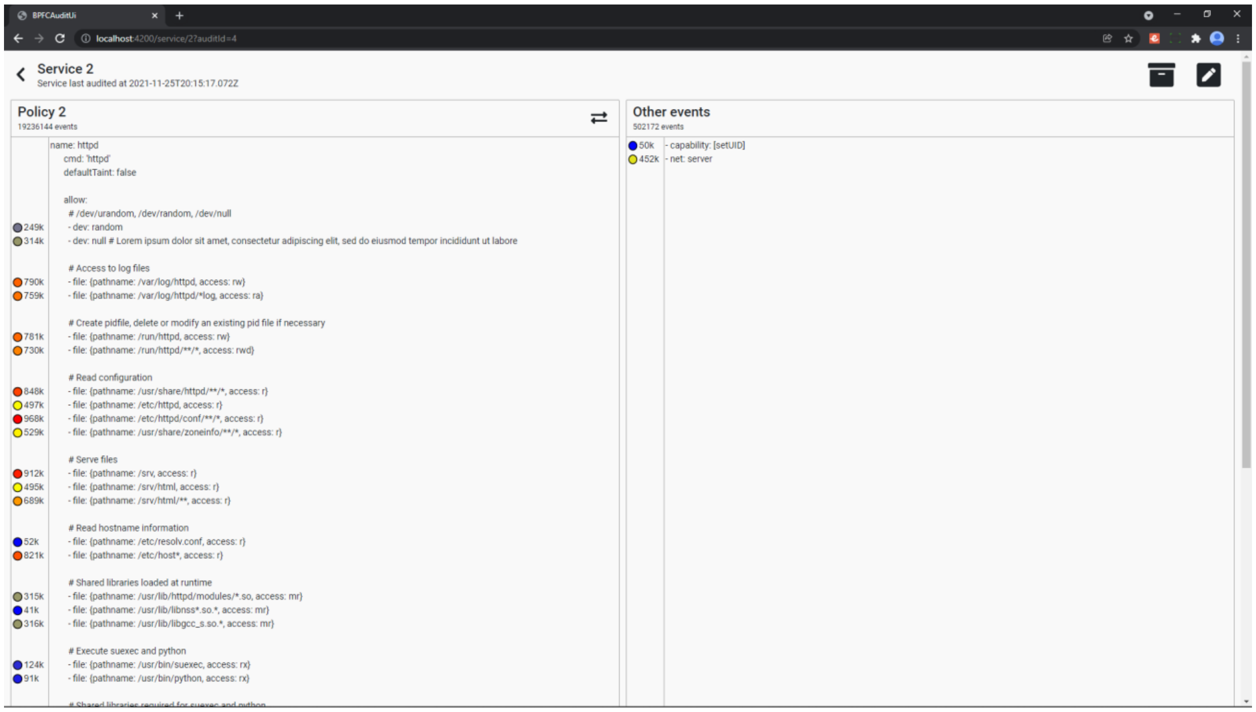


Figure 3.3: Average render time of the audit modal as the number of audits increases. Average render times are plotted, and a line of best fit is given. Render time increases as more audits are passed to the component are minimal.

3.2.3 Responsiveness Results

We first examine the responsiveness of the policy heatmap page using a resolution of 1920x1080 and with a large policy in Figure 3.4:



Service 2
Service last audited at 2021-11-25T20:15:17.072Z

Policy 2
19236144 events

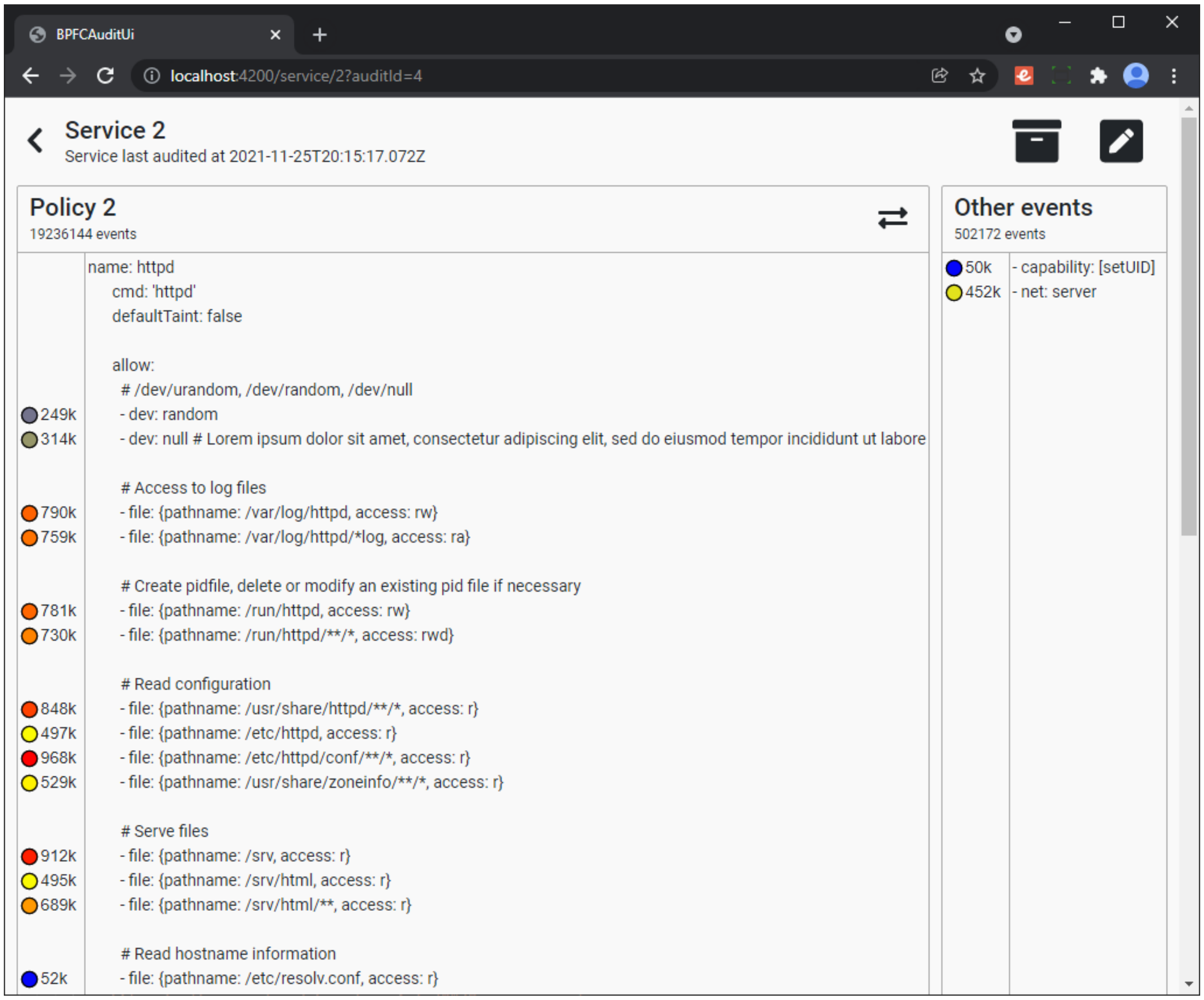
```
         name: httpd
           cmd: 'httpd'
           defaultTaint: false

           allow:
            # /dev/urandom, /dev/random, /dev/null
● 249k      - dev: random
● 314k      - dev: null # Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore

            # Access to log files
● 790k      - file: {pathname: /var/log/httpd, access: rw}
● 759k      - file: {pathname: /var/log/httpd/*log, access: ra}

            # Create pidfile, delete or modify an existing pid file if necessary
● 781k      - file: {pathname: /run/httpd, access: rw}
● 730k      - file: {pathname: /run/httpd/**/*, access: rwd}

            # Read configuration
● 848k      - file: {pathname: /usr/share/httpd/**/*, access: r}
● 497k      - file: {pathname: /etc/httpd, access: r}
● 968k      - file: {pathname: /etc/httpd/conf/**/*, access: r}
● 529k      - file: {pathname: /usr/share/zoneinfo/**/*, access: r}

            # Serve files
● 912k      - file: {pathname: /srv, access: r}
● 495k      - file: {pathname: /srv/html, access: r}
● 689k      - file: {pathname: /srv/html/**, access: r}

            # Read hostname information
● 52k       - file: {pathname: /etc/resolv.conf, access: r}
● 821k      - file: {pathname: /etc/host*, access: r}

            # Shared libraries loaded at runtime
● 315k      - file: {pathname: /usr/lib/httpd/modules/*.so, access: mr}
● 41k       - file: {pathname: /usr/lib/libnss*.so.*, access: mr}
● 316k      - file: {pathname: /usr/lib/libgcc_s.so.*, access: mr}

            # Execute suexec and python
● 124k      - file: {pathname: /usr/bin/suexec, access: rx}
● 91k       - file: {pathname: /usr/bin/python, access: rx}

            # Shared libraries required for suexec and python
```

Other events
502172 events

```
● 50k   - capability: [setUID]
● 452k  - net: server
```

Figure 3.4: The policy heatmap page with a large policy rendered at a resolution of 1920x1080.

As seen in Figure 3.4, no resizing issues are present on the page at 1920x1080 as this was the resolution that the BPFCAudit pages were initially designed for. Some of the content of the policy does not fit on the page but is easily viewed along with its accompanying "heat indicators" by using the vertical scrollbar shown on the right side of the page. All the heat indicators also remain nicely in line with their corresponding line in the policy as the page is scrolled. Since the heat indicators take on a fixed size and the text indicating how many times the rule was triggered is trimmed to at most four characters (e.g., 10^6 renders as 100k and 10^9 renders as 100M), these elements of the page can scale as expected with any number of audit events. We now try rendering the page at a resolution of 1024x768 in Figure 3.5:


BPFCAuditUi
localhost:4200/service/2?auditId=4
Service 2
Service last audited at 2021-11-25T20:15:17.072Z
Policy 2
19236144 events
Other events
502172 events

```
          name: httpd
            cmd: 'httpd'
            defaultTaint: false

            allow:
              # /dev/urandom, /dev/random, /dev/null
● 249k        - dev: random
● 314k        - dev: null # Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore

              # Access to log files
● 790k        - file: {pathname: /var/log/httpd, access: rw}
● 759k        - file: {pathname: /var/log/httpd/*log, access: ra}

              # Create pidfile, delete or modify an existing pid file if necessary
● 781k        - file: {pathname: /run/httpd, access: rw}
● 730k        - file: {pathname: /run/httpd/**/*, access: rwd}

              # Read configuration
● 848k        - file: {pathname: /usr/share/httpd/**/*, access: r}
● 497k        - file: {pathname: /etc/httpd, access: r}
● 968k        - file: {pathname: /etc/httpd/conf/**/*, access: r}
● 529k        - file: {pathname: /usr/share/zoneinfo/**/*, access: r}

              # Serve files
● 912k        - file: {pathname: /srv, access: r}
● 495k        - file: {pathname: /srv/html, access: r}
● 689k        - file: {pathname: /srv/html/**, access: r}

              # Read hostname information
● 52k         - file: {pathname: /etc/resolv.conf, access: r}
```

```
● 50k    - capability: [setUID]
● 452k   - net: server
```

Figure 3.5: The policy heatmap page with a large policy rendered at a resolution of 1024x768.

As illustrated by Figure 3.5, the relative size of the box containing the policy has grown with respect to the size of the box containing events sourced from BPFContain's default policy ("Other events"). This is because the two boxes are styled to take up as much space as is necessary on the page, and the policy box requires more space due to its longer content. It may be better to impose a minimum size on both boxes to ensure that they do not shrink "too much".

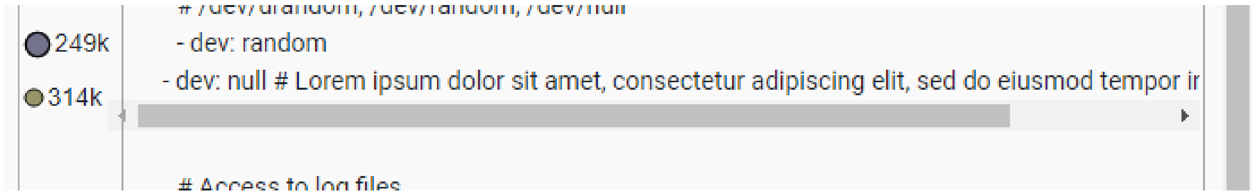
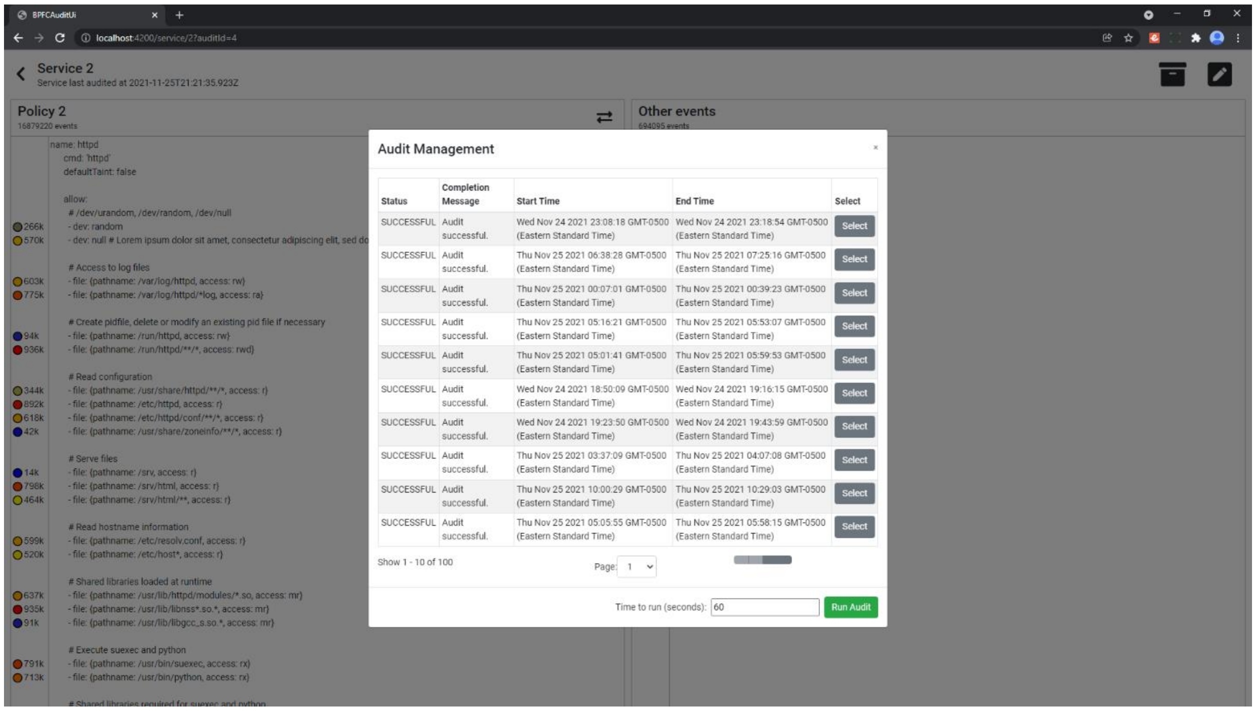

```
249k    - dev: random
314k   - dev: null # Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor ir
```

Figure 3.6: A responsiveness issue that occurs when a long line of the policy cannot fit in the viewport.

Figure 3.6 demonstrates an issue that occurs if the width of the page is shrunk even further. The size of the heat indicator for the line of the policy shrinks in size and is “pushed” to the left. The scrollbar that appears for the line of the policy also overlaps with the line separating the heat indicators from the policy. In addition, the line of the policy appears shifted in relation to the other lines of the policy making for reduced readability. While these responsiveness issues occur when viewport widths start to get very small with our test policy (< 700), this could happen at larger widths if the user provides a policy that has very long lines.

Another element that can be tested for responsiveness is the audit management modal originally presented in Figure 2.7. We suspect that responsiveness issues could occur on this page since it can display many Audits. We test the responsiveness of this modal with a service that has 100 associated Audits at a resolution of 1920x1080 in Figure 3.7. The resulting modal exhibits no issues displaying the resulting audits. This is thanks to the pagination that the table displaying the audits uses as this limits how many audits are displayed at a given time. The modal also exhibits no responsiveness issues at a resolution of 1024x768 and presents similarly to Figure 3.7.


BPFCAuditUi
localhost:4200/service/2?auditId=4
Service 2
Service last audited at 2021-11-25T21:21:35.923Z
Policy 2
16879220 events
Other events
694095 events

```
        name: httpd
          cmd: 'httpd'
          defaultTaint: false

          allow:
            # /dev/urandom, /dev/random, /dev/null
266k        - dev: random
570k        - dev: null # Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do

            # Access to log files
603k        - file: {pathname: /var/log/httpd, access: rw}
775k        - file: {pathname: /var/log/httpd/*log, access: ra}

            # Create pidfile, delete or modify an existing pid file if necessary
94k         - file: {pathname: /run/httpd, access: rw}
936k        - file: {pathname: /run/httpd/**/*, access: rwd}

            # Read configuration
344k        - file: {pathname: /usr/share/httpd/**/*, access: r}
892k        - file: {pathname: /etc/httpd, access: r}
618k        - file: {pathname: /etc/httpd/conf/**/*, access: r}
42k         - file: {pathname: /usr/share/zoneinfo/**/*, access: r}

            # Serve files
14k         - file: {pathname: /srv, access: r}
798k        - file: {pathname: /srv/html, access: r}
464k        - file: {pathname: /srv/html/**, access: r}

            # Read hostname information
599k        - file: {pathname: /etc/resolv.conf, access: r}
520k        - file: {pathname: /etc/host*, access: r}

            # Shared libraries loaded at runtime
637k        - file: {pathname: /usr/lib/httpd/modules/*.so, access: mr}
935k        - file: {pathname: /usr/lib/libnss*.so.*, access: mr}
91k         - file: {pathname: /usr/lib/libgcc_s.so.*, access: mr}

            # Execute suexec and python
791k        - file: {pathname: /usr/bin/suexec, access: rx}
713k        - file: {pathname: /usr/bin/python, access: rx}

            # Shared libraries required for suexec and python
```

Audit Management

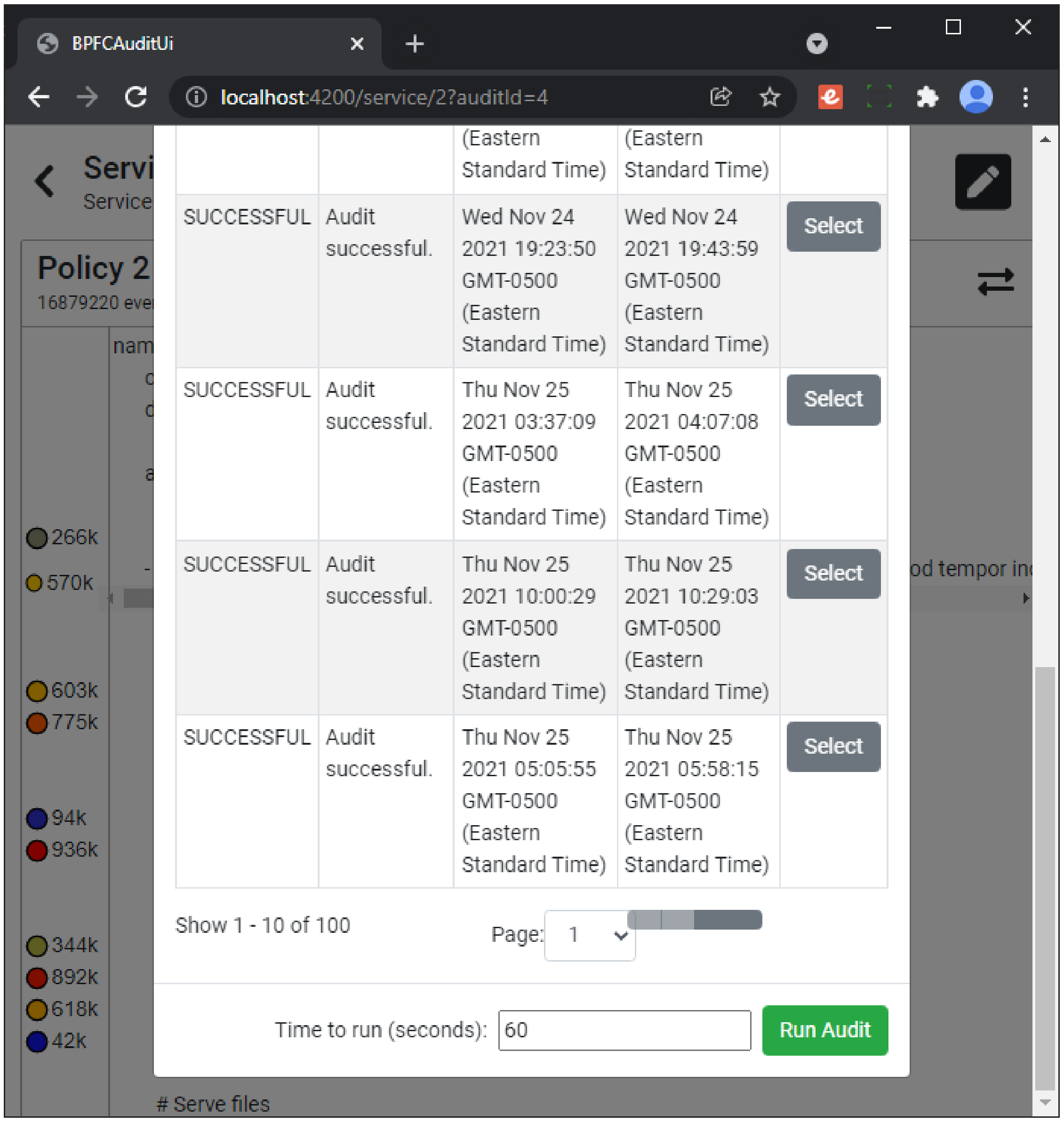
Status	Completion Message	Start Time	End Time	Select
SUCCESSFUL	Audit successful.	Wed Nov 24 2021 23:08:18 GMT-0500 (Eastern Standard Time)	Wed Nov 24 2021 23:18:54 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 06:38:28 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 07:25:16 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 00:07:01 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 00:39:23 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 05:16:21 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 05:53:07 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 05:01:41 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 05:59:53 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Wed Nov 24 2021 18:50:09 GMT-0500 (Eastern Standard Time)	Wed Nov 24 2021 19:16:15 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Wed Nov 24 2021 19:23:50 GMT-0500 (Eastern Standard Time)	Wed Nov 24 2021 19:43:59 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 03:37:09 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 04:07:08 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 10:00:29 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 10:29:03 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 05:05:55 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 05:58:15 GMT-0500 (Eastern Standard Time)	Select

Show 1 - 10 of 100   Page: 1

Time to run (seconds): 60   Run Audit

Figure 3.7: The audit management modal rendering 10 of 100 audits at a resolution of 1920x1080.

At very small resolutions, the page selection button overlaps with the pagination navigation button as shown in Figure 3.8. Other than this, the contents of the modal can be scrolled using the vertical scrollbar and the paged entries of the table can be traversed as expected.



		(Eastern Standard Time)	(Eastern Standard Time)	
SUCCESSFUL	Audit successful.	Wed Nov 24 2021 19:23:50 GMT-0500 (Eastern Standard Time)	Wed Nov 24 2021 19:43:59 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 03:37:09 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 04:07:08 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 10:00:29 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 10:29:03 GMT-0500 (Eastern Standard Time)	Select
SUCCESSFUL	Audit successful.	Thu Nov 25 2021 05:05:55 GMT-0500 (Eastern Standard Time)	Thu Nov 25 2021 05:58:15 GMT-0500 (Eastern Standard Time)	Select

Show 1 - 10 of 100

Page: 1

Time to run (seconds): 60 Run Audit

Figure 3.8: A responsiveness issue that occurs for the audit management modal at very small resolutions.

4 Discussion

We have now covered the design, implementation, and some of the results that BPFCAudit has attained with respect to some of its requirements. Considering this, we now turn our discussion towards how well BPFCAudit supports its end goals of policy development and policy validation and what limitations currently exist. Specifically, we will further discuss our results, briefly present some similar policy development frameworks, explore paths forward on addressing any limitations and determine how well BPFCAudit currently satisfies its requirements.

4.1 Satisfaction of Requirements and Associated Limitations

In Section 1.3, we introduced a set of functional requirements and a set of non-functional requirements that BPFCAudit should respect to be a viable framework for policy development and policy validation. We now explore how well BPFCAudit's current implementation satisfies these requirements, and what limitations, if any, are hurting the level of satisfaction for a given requirement.

4.1.1 Policy Analysis Presentation (F1)

It is believed that the BPFCAudit frontend presents rule violations and allows in a manner that supports policy analysis and policy verification (F1). Audit events that are either denied or allowed by the default policy are presented in a pane separate from the policy which clearly distinguishes them. This hopefully signals to the user that they need to refine their policy to add rules that explicitly permit the actions represented by these audit events. Usability testing is required to determine if this presentation is effective in practice. It should be noted that how events are presented in the "Other Events" window (e.g., in Figure 2.8) is likely inaccurate to how they would be presented when integrated with a live backend. Currently in BPFContain, the default policy is hard coded into the enforcement engine and there is no way to map the rules from the default policy into BPFContain's policy language [11]. While it is possible that this could change in the future, right now some of the "raw" data associated with the audit event would be presented and the user would have to translate this data into a rule in BPFContain's policy language, presenting usability concerns.

Another limitation associated with the policy analysis interface is that some of the backend APIs that are required by this interface are unimplemented. All figures demonstrating the policy analysis page depict mock data. As of right now, only the Audit Management modal is fully functional when attempting to integrate the frontend with the backend when visiting the policy analysis page. One API required to get the policy analysis interface fully functional is a PATCH API that can be used to update the policy that is associated with a given service. The relationship between services and their policies is also not currently a part of the backend data model. To allow the user to specify this relationship, one likely approach could be presenting the available policies based on a configurable directory that contains all known policies, from which the user could select one. An API for this would be needed as well and is currently modeled by the frontend's mock APIs. A sensible default policy directory could be BPFContain's own default directory for policies (/var/lib/bpfcontain/policy), though root privileges will be needed to read entries. The policy analysis API would also need to be implemented. The design of this API is well detailed in Section 2.2.3.

4.1.2 Audit Event Data Collection (F2)

BPFCAudit's ability to collect live audit event data from BPFContain containers as presented in Section 2.2.2 is mostly functional. That is, BPFCAudit is able to ingest audit event data from BPFContain as quickly as it can produce it as shown in Section 3.1.2 and the persisted audit event data does not lose any details that would be likely to matter in a resulting policy analysis. A feature that is unimplemented in BPFContain which also requires subsequent integration from BPFCAudit is the ability to filter audit event data. Currently, BPFCAudit enforces that only one audit runs at a given time. As explained in Section 2.2.2, this is due to the fact that BPFContain audit filters are unimplemented and it must be assumed that only one BPFContain container is running and producing audit events at a given time to guarantee that these events are associated with the proper service. Section 2.2.2 details a filter that BPFContain could accept which would guarantee this association between audit events and a given service.

A quality-of-life feature that could be implemented for the audit API is the automatic instantiation of a BPFContain container when an audit is triggered for a given service. By using a wrapper around the BPFContain run command, a contained instance of a program could automatically be created before audit data is collected. This way, the user would not have to manually orchestrate the containerization of their service before running an audit. For use cases that require certain flags to be passed to a program upon startup, this could be added as a parameter to the audit API.

4.1.3 Audit Data Model and Persistence (F3)

BPFCAudit's implementation was mostly able to meet F3. That is, BPFCAudit audits successfully "time-box" audit event data such that they are only collected during this time. BPFCAudit is also able to persist audits and their associated audit event data so that they can be later retrieved if the application is shut down. This supports policy validation as users can build a history of audit event data and continuously develop their policy in accordance with the development of their application. The only portion of F3 that was not met was the ability to tag audits. The intention of audit tagging is so that users can understand the motivation behind a given audit, or which version of their application a given audit represents. All that is needed for this is a PATCH API for audits which permits for updating a "tag" field on a given audit, and an associated button and modal on the frontend, specifically in the audit management modal (Figure 2.7).

4.1.4 Efficiency (NF1)

The performance (NF1) of BPFCAudit's backend was characterized in relation to BPFContain (Section 3.1). It was found that BPFCAudit can process audit events as quickly as BPFContain can produce them. However, BPFContain does not currently send all audit events that it produces over the WebSocket that it publishes. Because of this, the resulting audit data is only as accurate as what BPFContain publishes (i.e., “dropped” events do not appear in the resulting policy analysis). BPFContain’s interface for publishing audit events is not yet finalized, and significant performance improvements can still be made [11]. Because of this, BPFCAudit’s ability to process audit events will need to be re-tested once BPFContain can more quickly publish them.

Basic rendering performance tests were conducted for portions of BPFCAudit’s frontend (Section 3.2.2). As of writing, the ability of the frontend to scale and quickly render large amounts of data is well within acceptable bounds. All these tests were however conducted with mock implementations of BPFCAudit’s APIs. Since the time it takes for API requests to complete generally represents a significant portion of a web application’s loading time, it is expected that page load times will take longer once these APIs are available and the frontend is fully integrated with them. Thus, more work will be required to determine if the application is performant once these APIs are implemented.

4.1.5 Deployment (NF2)

A major hurdle to the adoptability of BPFCAudit is its deployment (NF2). Currently, the steps required to get BPFCAudit’s frontend and backend, and the PostgreSQL datastore running and integrated together are partially documented on both the backend’s and frontend’s GitHub. The steps involved are numerous, and deployment has not been tested on any machine other than an Ubuntu VM. Ideally, BPFCAudit could be deployed with a “one-and-done” command, allowing the user to begin using the application without having to work through a complicated setup and configuration. This could be done using Docker and Docker Compose. While official Docker images exist for PostgreSQL, docker images would have to be created and published for the frontend and backend [35]. Docker Compose could then be used to deploy all these containers in one command [36]. By using Docker Compose, BPFCAudit could be easily deployed in a consistent fashion for any platform that supports Docker.

4.1.6 Extensibility (NF3)

Due to time constraints, many best practices that would have taken more time and careful deliberation to implement were forgone in the implementation of BPFCAudit's backend. This is due to the backend being more of a "feasibility" test, in that certain processes such as audit event data collection had to be implemented and tested before BPFCAudit could be considered a viable application. I did not want to spend time developing BPFCAudit's frontend if the plans for BPFCAudit's backend did not work in practice. Because of this lack of best practices, the extensibility (NF3) of the framework has suffered and is currently tied heavily with BPFContain. That is, trying to integrate with another policy-based containerization framework will first require a major refactor to avoid code repetition and ensure a maintainable application. It is not expected that significant challenges would be posed by integrating with a new container framework in BPFCAudit's frontend, as all application logic is driven by BPFCAudit's backend. If multiple containerization frameworks are supported in the future, BPFCAudit's data models may need to be "annotated" with the name of the framework that they are associated with. What data is presented to the user could then be determined by an application-level configuration specifying the currently in-use framework.

4.1.7 Usability (NF4)

Some usability (NF4) problems of the frontend were highlighted by the responsiveness issues found in Section 3.2.3. It is not expected that these responsiveness issues would be difficult to address as many similar issues were found and resolved during development The audit management modal also has issues displaying its icons as demonstrated by Figure 2.7. The result is a confusing interface with buttons that do not serve a clear purpose. Sorting arrows also do not appear on the column headers in the modal's table for displaying audits. More work is required to determine the correct approach to get these icons to render properly. With the current implementation of the policy heatmap page, the user would have to make changes to the policy directly on their system as opposed to just editing it directly in BPFCAudit. This is inconvenient as the user would have to continuously switch between the BPFCAudit and an editor to change their policies when one would likely expect that they could edit the policy directly in the policy pane.

There are also significant usability problems posed by the unimplemented overview of services page (Section 2.3.4). The page's implementation does not represent its original design and provides no information other than what services exist on the system. It is also highly unclear what the purpose of the page is when no services exist on the system, as the only elements present would be the text indicating that all the services are listed and a button for creating new services. At the very least, this case of no services should be handled, and additional instruction should be provided to users. Another usability issue that currently exists in the frontend is the lack of ability to delete services and audits. Audits or services that no longer serve a purpose to the user would thus remain unless the user directly modifies the database.

To get a solid idea of how usable BPFCAudit is in practice, usability testing is required as is described in Section 3.2 with respect to the policy heatmap page. We could extend this proposed usability testing to encompass the entirety of BPFCAudit's user interface once it is fully implemented. After this study is conducted, we can determine whether changes are required to make BPFCAudit's GUI easier to work with to improve its adoptability.

4.1.8 Reliability (NF5)

Throughout all my manual testing, no unexpected terminations of audits occurred. However, no guarantees of the proper termination of audits have been established. Since the nature of audits is highly asynchronous, more work is needed to determine the overall reliability (NF5) of this process. Because no rigorous testing has been done, it is possible that race conditions exist in the audit workflow or that certain exceptions could be handled more gracefully. If an exception occurs during the audit workflow currently, the entire audit is halted and marked as failed along with the reason for failure. It is possible that certain exceptions could be safely logged and "ignored" rather than "throwing away" the entire audit. It may also be advantageous to introduce a retry policy in the case that the WebSocket connection to BPFContain is dropped or never established.

A major issue with the maintainability and overall reliability of BPFCAudit is the lack of automated tests in both the frontend and the backend. A good starting point would be to add unit tests to both components. By adding unit tests, basic verification of isolated portions of BPFCAudit could be done and provide some much-needed documentation as well [37]. Furthermore, integration tests which verify that BPFCAudit's individual components work together could be written as well [38]. It is likely that this style of testing would take considerable effort to create, however. This is because integration testing is mostly precluded by BPFCAudit's current deployment limitations (Section 4.1.5) and scripts would need to be created to properly create a test environment if these issues were not first addressed.

4.2 Other Policy Development Tools

At a high level, BPFCAudit is a GUI which visualizes a security policy and its efficacy in relation to empirical data. The application is highly specialized in that it currently supports policy development only for BPFContain and in this sense it is likely the first application of its kind. However other interfaces have been developed for similar policy analysis problems for other access control frameworks. These interfaces are worth considering as BPFCAudit's GUI could be compared to them as usability benchmarks. If BPFCAudit's GUI is not a novel approach for supporting the development of security policies, it may also be worth considering adopting an existing policy analysis interface as it is likely to be more mature. It could also be worth integrating BPFCAudit's components into these existing interfaces, resulting in a consolidated interface for the analysis of security policies for various security frameworks.

SELinux implements mandatory access control (MAC) to control what system resources are available to a given user role, program, or service [39]. Somewhat similarly to BPFContain, targeted SELinux policies can be written which specify the exact resources and actions that a program requires to function and enforce this, but it can also specify system-wide security policies. SELinux's approach to enforcement is however much more complex than BPFContain's eBPF-based approach and results in significantly more complex policies which are difficult to audit [40]. Because of this complexity, several tools and novel approaches have been developed to assist in the analysis and creation of SELinux policies. One such tool PVA (policy visualization analysis) allows for the graphical querying of policies to identify possible violations of a security goal [41]. In contrast to BPFCAudit which uses audit logs to identify unused rules and insufficient privileges for a given container, PVA allows users to specify a set of security goals and automatically identifies rules in a SELinux policy that conflict with these goals. PVA can then visualize these violations in a “semantic substrate” and an adjacency matrix. While PVA aims to assist policy development for policies that encompass entire systems rather than individual containers, its approach of specifying security goals could be worth considering for the development of BPFContain policies. BPFCAudit’s approach is however beneficial in the case that a user has little knowledge of the application they are confining since it drives policy development using data from a given container, rather than requiring the user to have deep knowledge of an application’s resource needs.

Another tool used for the goal of security policy development is AppArmor’s aa-logprof. Like BPFContain, AppArmor enforces access to resources on a per-application basis using a security policy [42]. Unlike BPFCAudit aa-logprof does not provide a GUI, but it can audit an application’s behavior under a given policy like BPFCAudit. aa-logprof also allows for the interactive creation of rules with a policy that is set to a complaining mode, which is very similar to BPFContain’s concept of complaining policies. aa-logprof is interacted with using a command prompt but provides a significant advantage over BPFCAudit in that it can automatically generate rules based on encountered events. Generating BPFContain rules from audit event data would be a significant improvement to BPFCAudit’s usability as opposed to the current approach of requiring users to write rules based on the presented audit events.

4.3 The Future and Adoptability of BPFCAudit

This project’s primary goal was to create an application that can assist in the development and verification of BPFContain policies. In support of this goal, we defined some functional requirements in Section 1.3 that BPFCAudit should satisfy. In summary, these functional requirements stated that BPFCAudit should:

1) Display events that coincide with BPFContain rules in a user interface in a fashion that supports policy development (F1).
2) Collect audit event data from live BPFContain containers and use this data to drive policy analysis and development (F2).
3) Persist collected audit event data into audits which ensure this data pertains to a specific time to build a historical profile of an application's behavior (F3).

We explored to what extent we were able to implement these functional requirements in BPFCAudit and what specific limitations remain in Section 4.1. Overall, it is believed that BPFCAudit was able to mostly able to deliver on these functional requirements although adoptability is greatly hampered by some dissatisfaction of the application's non-functional requirements, particularly its deployment (NF2). Large work items remain in terms of implementation and integration of some APIs as well. Time also must be spent to sufficiently test and document the existing application before tackling other limitations. Thus, we believe that BPFCAudit is currently a good foundation for supporting its end goal of policy development and verification, specifically in relation to BPFContain.

In terms of BPFCAudit's positioning in relation to supporting additional policy-based containerization frameworks, the first step is addressing BPFCAudit's existing limitations (Section 4.1) and extensibility concerns (NF3). Beyond this, the Lockc project whose goals are very similar in nature to BPFContain's is an in-development policy-based containerization framework that recently adopted BPFContain's policy language [43]. Once fully implemented, it is possible that BPFCAudit could be applied the Lockc framework to support policy development and validation.

It should however be noted that BPFCAudit poses a considerable burden on these policy-based containerization frameworks in order to collect the data necessary to support policy development. Specifically, BPFCAudit mandates that the framework must accurately and efficiently publish an event each time a rule being enforced is used to allow or deny some action. BPFContain's interface for publishing audit events is partially explored in Section 2.2.2. BPFCAudit also requires that the framework publishes an interface for deserializing a policy into rules (and that these rules have consistent hashes). It is not expected that this is a significant burden as the framework must already be able to convert a policy into rules to implement enforcement. BPFCAudit only requires that the given framework publishes this interface. BPFContain's proposed interface for converting a policy into rules is partly explored in Section 2.2.3. Thus, framework authors must first consider if providing additional support for policy development is necessary. If so, they must then consider if it is worth including these interfaces in their project to support BPFCAudit or similar solutions, or if they can produce a tool that supports policy development in less effort than building these interfaces.

BPFCAudit is very likely the first policy analysis tool developed for BPFContain policies. However other policy analysis tools for other security frameworks have been developed, some of which were explored briefly in Section 4.2. One of these tools takes a similar approach to BPFCAudit in that it drives policy development based on empirical data collected from applications, but unlike BPFCAudit it is capable of automatically generating rules and is a CLI. Another tool allows users to specify security goals and the resulting analysis identifies problems with the user's policy in a GUI. To properly consider the merits of these alternative policy analysis tools in relation to BPFCAudit, usability testing is required. After this testing is conducted, BPFCAudit could be modified to support policy development and verification in a more user-friendly fashion.

5 Conclusion

In this honours project report, I have introduced the challenges associated with developing BPFContain policies (Section 1). I have also presented a proposed solution to these challenges with the design and implementation of BPFCAudit (Section 2), which provides an interface for analyzing BPFContain policies based on data collected from running BPFContain containers. Some testing has revealed that BPFCAudit can effectively collect all events that BPFContain publishes and that the BPFCAudit frontend is performant and mostly responsive while using mock data (Section 3). Finally, we explored how well BPFCAudit's implementation satisfies its functional and non-functional requirements, limitations associated with each requirement, some alternative policy development tools, and the future of BPFCAudit (Section 4).

While the limitations of BPFCAudit's current implementation make it unusable in practice, BPFCAudit succeeds in being a strong framework for supporting policy development and verification. If BPFCAudit's limitations such as its difficulty of deployment were addressed, it could be used to develop BPFContain policies for many applications. By supporting policy development and verification, BPFCAudit could greatly increase the adoptability of BPFContain by reducing the complexity associated with writing policies. Furthermore, BPFCAudit could be extended to support policy-based containerization frameworks other than BPFContain. BPFCAudit could also provide interface specifications that these frameworks could adhere to to reduce the burden BPFCAudit would have adapting to each. However other policy-based containerization frameworks must consider if it is easier to write their own policy development tool rather than providing the interfaces that BPFCAudit requires to function.

6 References

[1] W. Findlay, "A Practical, Lightweight, and Flexible Confinement Framework in eBPF," August 2021. [Online]. Available: https://www.cisl.carleton.ca/~will/written/techreport/mcs-thesis.pdf.

[2] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu and W. Zhou, "A Comparative Study of Containers and Virtual Machines in Big Data Environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD),* 2018.

[3] P. C. V. Oorschot, Computer Security and the Internet - Tools and Jewels, Springer, 2020.

[4] IBM, "Three-Tier Architecture," 28 October 2020. [Online]. Available: https://www.ibm.com/cloud/learn/three-tier-architecture.

[5] json:api editors, "json:api," 2021. [Online]. Available: https://jsonapi.org/.

[6] M. Morley, "JSON-RPC," 2013. [Online]. Available: https://www.jsonrpc.org/specification.

[7] VMWare, "Spring," 2021. [Online]. Available: https://spring.io/projects/spring-boot#overview.

[8] Spring Initializr Contributors, "initializr," November 2021. [Online]. Available: https://github.com/spring-io/initializr.

[9] The PostgreSQL Global Development Group, "About PostgreSQL," 2021. [Online]. Available: https://www.postgresql.org/about/.

[10] javaTpoint, "Spring Data JPA," 2021. [Online]. Available: https://www.javatpoint.com/spring-boot-starter-data-jpa.

[11] W. Findlay, *Private Communication,* 2021.

[12] Oracle, "Interface ScheduledExecutorService," 2020. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html.

[13] baeldung, "WebSockets with AsyncHttpClient," 14 August 2019. [Online]. Available: https://www.baeldung.com/async-http-client-websockets.

[14] Oracle, "Class ConcurrentHashMap," 2021. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html.

[15] Oracle, "Class LongAdder," 2021. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html.

[16] Figma, "Design Features," [Online]. Available: https://www.figma.com/design/.

[17] IEEE, "Responsive Web Design Guidelines for IEEE Sites," 2021. [Online]. Available: https://brand-experience.ieee.org/guidelines/digital/mobileapp-and-responsive-design-guidelines/responsive-design/.

[18] B. Gaille, "15 Desktop vs Web Application Pros and Cons," 9 March 2019. [Online]. Available: https://brandongaille.com/15-desktop-vs-web-application-pros-and-cons/.

[19] M. Pekarsky, "Does your web app need a front-end framework?," 3 February 2020. [Online]. Available: https://stackoverflow.blog/2020/02/03/is-it-time-for-a-front-end-framework/.

[20] Tilde Inc., "Ember Overview," 2021. [Online]. Available: https://emberjs.com/.

[21] Tilde Inc., "Ember Data Introduction," 2021. [Online]. Available: https://guides.emberjs.com/release/models/.

[22] Tilde Inc., "Ember Inspector Introduction," 2021. [Online]. Available: https://guides.emberjs.com/release/ember-inspector/.

[23] Microsoft, "Typescript for the New Programmer," 16 November 2021. [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html.

[24] Sass team, "Sass Basics," 2021. [Online]. Available: https://sass-lang.com/guide.

[25] Mirage, "Mirage Introduction," [Online]. Available: https://miragejs.com/docs/getting-started/introduction/.

[26] Onechiporenko, "Ember Models Table Demo," 2021. [Online]. Available: http://onechiporenko.github.io/ember-models-table/v.3/bs4/#/examples/common-table.

[27] baeldung, "Difference Between save() and saveAndFlush() in Spring Data JPA," 29 July 2020. [Online]. Available: https://www.baeldung.com/spring-data-jpa-save-saveandflush#save.

[28] J. Jenkov, "Java Reflection Tutorial," 25 September 2018. [Online]. Available: http://tutorials.jenkov.com/java-reflection/index.html.

[29] IEEE, "Usability Testing Toolkit for IEEE Web Publishers," 2021. [Online]. Available: https://brand-experience.ieee.org/guidelines/digital/other-guidelines/usability-testing/.

[30] Tilde Inc., "Ember Guides," 2021. [Online]. Available: https://guides.emberjs.com/release/ember-inspector/render-performance/.

[31] Tilde Inc., "cli.emberjs," 2021. [Online]. Available: https://cli.emberjs.com/release/basic-use/cli-commands/.

[32] "Window Resizer," 2016. [Online]. Available: https://coolx10.com/window-resizer/.

[33] W. Findlay, "BPFcontain-rs," 17 September 2021. [Online]. Available: https://github.com/willfindlay/bpfcontain-rs.

[34] Google Search Central, "Site Performance For Webmasters," 2 May 2010. [Online]. Available: https://www.youtube.com/watch?v=OpMfx_Zie2g&t=567s.

[35] PostgreSQL Contributors, "postgres," November 2021. [Online]. Available: https://hub.docker.com/_/postgres.

[36] Docker Inc., "Overview of Docker Compose," 2021. [Online]. Available: https://docs.docker.com/compose/.

[37] Smartbear, "What is Unit Testing," 2021. [Online]. Available: https://smartbear.com/learn/automated-testing/what-is-unit-testing/.

[38] Smartbear, "What is Integration Testing," 2021. [Online]. Available: https://smartbear.com/learn/automated-testing/what-is-integration-testing/.

[39] CentOS wiki contributors, "SELinux," 20 March 2020. [Online]. Available: https://wiki.centos.org/HowTos/SELinux.

[40] A. Eaman, B. Sistany and A. Felty, "Review of Existing Analysis Tools for SELinux Security Policies: Challenges and a Proposed Solution," 2017. [Online]. Available: https://www.site.uottawa.ca/~afelty/dist/mcetech17.pdf.

[41] W. Xu, M. Shehab and G.-J. Ahn, "Visualization-based policy analysis for SELinux: framework and user study," 2012.

[42] Archlinux Wiki Contributors, "AppArmor," 7 November 2021. [Online]. Available: https://wiki.archlinux.org/title/AppArmor.

[43] Lockc contributors, "Lockc," October 2021. [Online]. Available: https://github.com/rancher-sandbox/lockc.

7 Appendices

Appendix A. BPFContain Policy Used for Audit Performance Testing

```yaml
# A unique name for the policy file
name: perfTest

# Spawn tainted
defaultTaint: true

# Spawn in complaining mode
complain: true

# A set of restrictions (rights to explicitly revoke)
restrictions:
  - fs: {path: /home/cody/Desktop/myFile, access: r}
```

Listing A-1: The BPFContain Policy used for Audit Performance Testing (perfTest.yml)

Appendix B. Multithreaded C Program that Rapidly Opens a File Used for Audit Performance Testing

```
#include <stdlib.h>
#include <argp.h>
#include <pthread.h>
#include <unistd.h> // sleep

// Adapted from argp example #3
const char *argp_program_version = "";
const char *argp_program_bug_address = "";

static char doc[] = "A program for rapidly opening a file in read mode.";
static char args_doc[] = "";

/* The options we understand. */
static struct argp_option options[] = {
  {"single",   'l', 0,         0,  "Only read a single file, do not spawn threads
that open repetitively"},
  {"file",     'f', "file",    0,  "File to open in read mode" },
  {"threads",  't', "threads", 0,  "Number of threads to open the file for"},
  {"seconds",  's', "seconds", 0,  "Amount of time in seconds to open the file"},
  { 0 }
};

/* Used by main to communicate with parse_opt. */
struct arguments
{
  int single, threads, secondsToRun;
  char *file;
};

/* Parse a single option. */
static error_t parse_opt (int key, char *arg, struct argp_state *state)
{
  /* Get the input argument from argp_parse, which we
     know is a pointer to our arguments structure. */
  struct arguments *arguments = state->input;

  switch (key)
    {
```

```
    case 'l':
      arguments->single = 1;
      break;
    case 't':
      arguments->threads = atoi(arg);
      break;
    case 'f':
      arguments->file = arg;
      break;
    case 's':
      arguments->secondsToRun = atoi(arg);
      break;

    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}

// argp parser
static struct argp argp = { options, parse_opt, args_doc, doc };

// Open the file a given file. Close the file descriptor if for some reason we
are able to get one.
int openFile(char* file_name)
{
   FILE *fp;
   fp = fopen(file_name, "r");

   if (fp != NULL) {
     fclose(fp);
   }

   return 0;
}

struct thread_data {
  char* file_name;
  int ret;
};

// Flag used to indicate that spawned threads should exit and we should collect
their counts.
int quitFlag = 0;
```

```
// Function that our threads will run, rapidly opens a file and counts how many
times this is done
// so long as the quit flag has not been set.
void *threadOp(void *threadArg)
{
  struct thread_data *args;
  args = (struct thread_data *) threadArg;
  int counter = 0;

  while (1)
  {
    if (quitFlag)
    {
      args->ret = counter;
      pthread_exit(NULL);
    }
    openFile(args->file_name);
    counter += 1;
  }

  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  struct arguments arguments;
  char ch;
  char* file_name;
  FILE *fp;

  // Arg defaults
  arguments.single = 0;
  arguments.threads = 1;
  arguments.file = "/home/cody/Desktop/myFile";
  arguments.secondsToRun = 30;

  // Parse arguments
  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  pthread_t threads[arguments.threads];
  int* threadResults[arguments.threads];
  struct thread_data thread_data[arguments.threads];
  file_name = arguments.file;
  // File opening control.

  // Open the file once and exit.
  if (arguments.single)
  {
    openFile(file_name);
    return 0;
  }

  // Open the file repeatedly with 't' threads.
  // Based on https://hpc-tutorials.llnl.gov/posix/creating_and_terminating/
  int pthread_ret;
  for (int i=0; i < arguments.threads; i++)
  {
    thread_data[i].file_name = file_name;
    thread_data[i].ret = 0;
    pthread_ret = pthread_create(&threads[i], NULL, threadOp, (void *)
&thread_data[i]);

    if (pthread_ret) {
      printf("ERROR; return code from pthread_create() is %d\n", pthread_ret);
      exit(-1);
    }
  }

  // Wait until we have opened the file for as long as the user wants to, then
kill all threads.
  sleep(arguments.secondsToRun);
  quitFlag = 1;

  // Ensure threads have been terminated.
  for (int i=0; i < arguments.threads; i++)
  {
    pthread_join(threads[i], NULL);
  }

  // Count how many times we opened the file by summing the reported counts from
each thread.
  int result = 0;
  for (int i=0; i < arguments.threads; i++)
  {
    result += thread_data[i].ret;
  }
  printf("Opened the file %d times in %d seconds.\n", result,
arguments.secondsToRun);

  return 0;
}
```

Listing B-1: A tunable C program that rapidly opens a file using multiple threads. Used for triggering BPFContain audit events rapidly in combination with the policy listed by Appendix A.

Appendix C. Bash Script Used to Orchestrate Audit Performance Tests

```bash
#!/usr/bin/env bash
set -e # Exit if any command fails

# Parameters for tuning testing
TIME_TO_WAIT_FOR_AUDIT_TO_SETTLE_IN_SECONDS=3
HOW_LONG_TO_RUN_AUDIT_FOR_IN_SECONDS=120
# Add audit settle time to audit runtime as this time period has no events
occuring
AUDIT_RUNTIME_IN_SECONDS=$(( $TIME_TO_WAIT_FOR_AUDIT_TO_SETTLE_IN_SECONDS +
$HOW_LONG_TO_RUN_AUDIT_FOR_IN_SECONDS ))
TIME_TO_PRODUCE_EVENTS_FOR=15
THREADS=2

# 1. Restart BPFContain daemon to reset global audit stats counter - so we should
run with root privileges

# If stop fails (the daemon was not already running), ignore the failure result
bpfcontain daemon stop || true

# Need to wait a moment as BPFContain uses a pid lock
sleep 1s

bpfcontain daemon start

# Wait for BPFContain to start up
sleep 2s

# 2. Audit creation request

# Audit creation request parameters
AUDIT_END_TIME_ISO_8601=$(date -u --date "+${AUDIT_RUNTIME_IN_SECONDS} seconds"
+"%Y-%m-%dT%H:%M:%SZ")
SERVICE_ID_FOR_AUDIT=46940
```

```
# Perform audit creation request
printf "Audit creation request response:\n"
curl -f --location --request POST 'http://localhost:7060/bpfca/api/v1/audits' \
--header 'Content-Type: application/vnd.api+json' \
--header 'Accept: application/vnd.api+json' \
--data-raw "{
   \"data\":{
      \"type\": \"audits\",
      \"attributes\":{
         \"endTime\": \"${AUDIT_END_TIME_ISO_8601}\",
         \"serviceId\": 46940
      }
   }
}"

# Wait for the audit creation to settle
sleep 3s

# 3. Contain and run our spammy program
printf "\n\nStarted to produce audit events at $(date)\n"

bpfcontain run perfTest.yml -- "/home/cody/Desktop/AuditEventSpam/eventSpam -t
${THREADS} -s ${TIME_TO_PRODUCE_EVENTS_FOR}"

printf "Finished producing audit events at $(date)\n"
```

Listing C-1: Bash script used to orchestrate audit performance testing.