Securing Linux Storage with ACLs: An Open-Source Web Management Interface for Enhanced Data Protection

Proposal Presentation for GSOC 2025 Project

Aditya Patil

Introduction

Electrical Engineering Student at the National Institute of Technology Hamirpur, India
(BTech. 3rd Year)

Backend Engineer and Linux Developer

Previous Experience:
DevSecOps Intern, Athena Consulting Ltd. Dubai
Technical Growth Manager - Freelancing at Granot Inc. Tel. Aviv, Israel
Security Researcher (Contract) - Chucku LLC. Texas, USA

Avid Open Source Contributor
Working on building Stockic (Startup in Progress)

*My CV contains all you need to know about me

Understanding of the Requirements

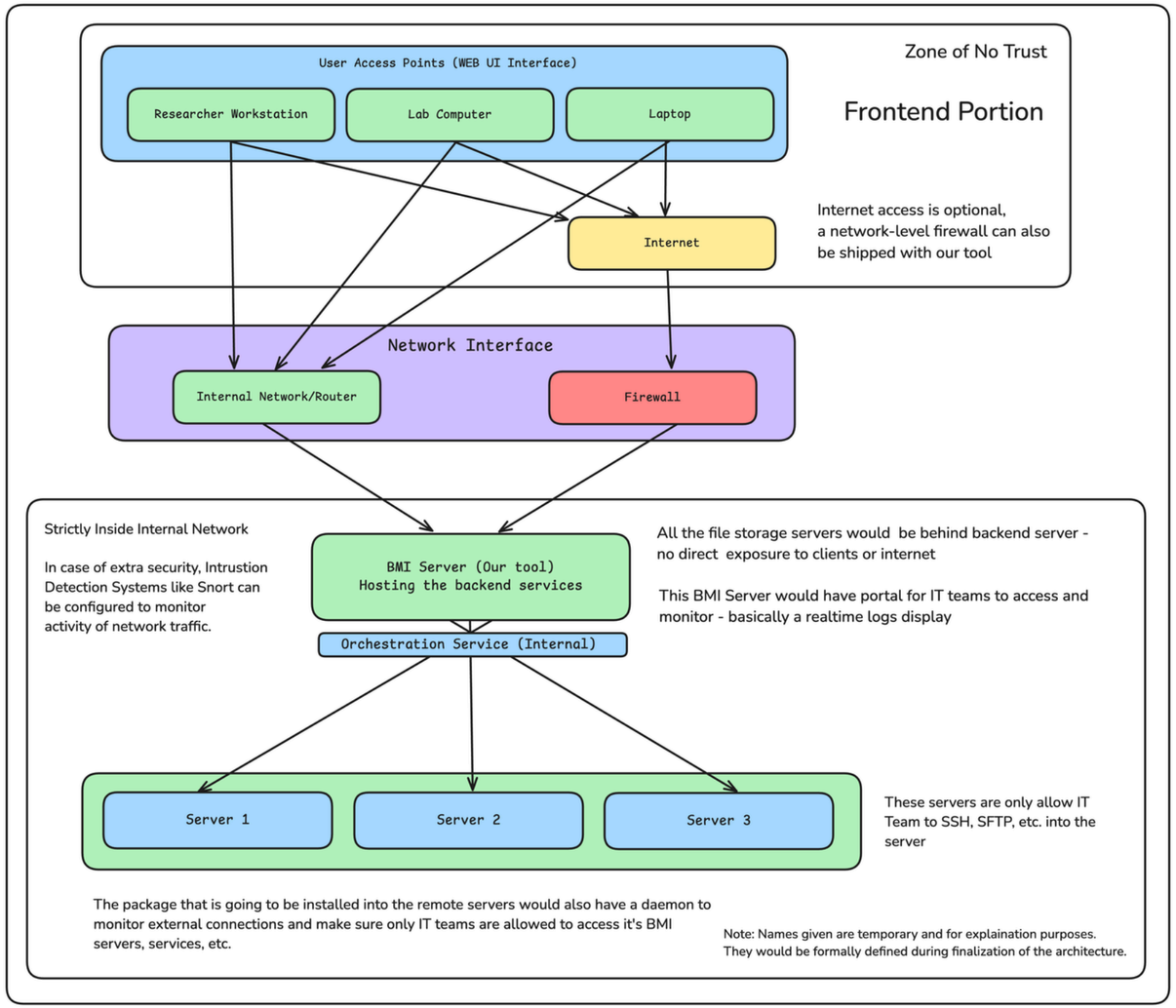
Features

The Project is going to be Open Source and, hence, must be highly flexible
Documentation must be well maintained - User Docs and Code Docs
Ideally Scalable to Infinite Linux File Servers
Simplicity to its Peak

All the files and directories must be granular in permission-granting
Only Web Access to Users
SSH/SFTP, etc. access only to IT Teams

High Level Network Architecture

Mentors Recommendation: These will be incorporated once the development of the real project starts. In the prototype, we would be focusing more on Proof of Concept and might not take these design considerations into account for faster development cycle.



Characteristics

More Focus on Security and Simplicity
High Modularity and Scalability
Replicable Builds and Highly Configurable
Highly Flexible (Since it’s Open Source)

Technicalities

Loosely Coupled Frontends and Backends
Zero-Trust Policy Over the Network
Focus on the Least Privilege Principle
More Power to IT Teams

Security Aspects of this Design

File Servers stay in the Internal Network
File Servers are not even exposed to the Frontend Devices
Controllers File Servers only accept requests from the backend, and no other
SSH/SFTP, etc., only allowed to IT Teams through the Internal Network (Configurable)

Feature Aspects of this Design

Replicable Configuration through YAML Schemas
Loosely Coupled REST API structure for Backend-Frontend

Proposed System Design for 1st Approach

Revision 1 (with incorporated changed from first meeting)

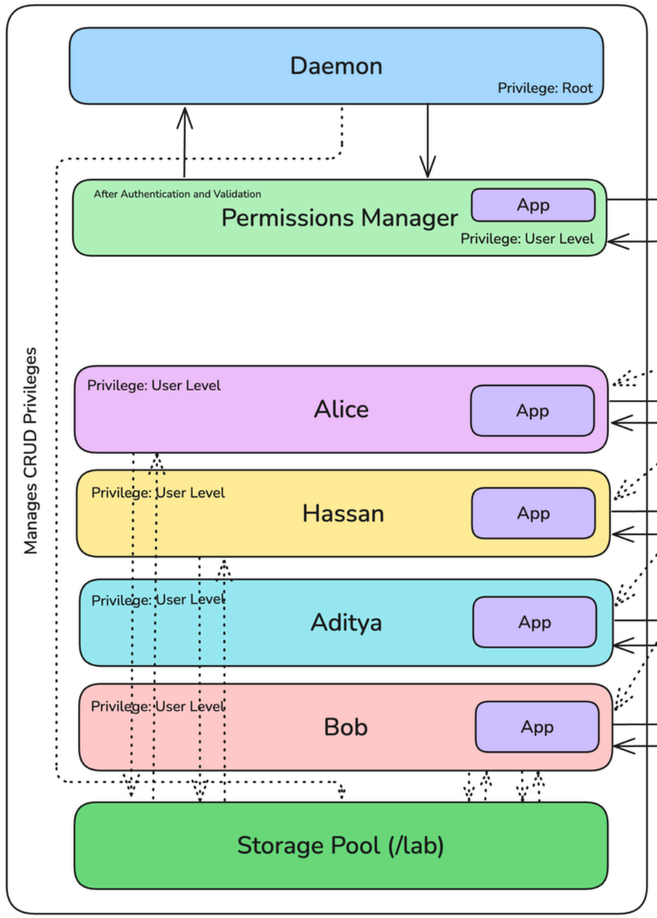
The daemon is running on Root privileges and hence, must not be exposed to even the local network - in case any exploit comes with the network service, it must not get compromised

The permissions manager can ask daemon to change privileges. This is done only if the request coming from the network is authenticated and is confirmed to be coming from IT teams

Hash Tables

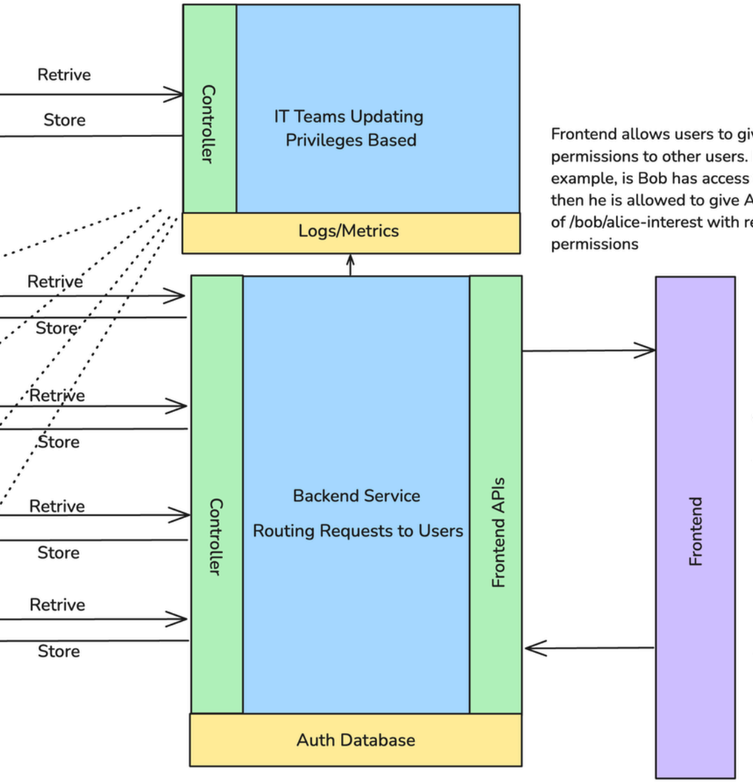
Since we need to be aware of changes been made in the directories and files in the system, we need a responsive mechanism to detect changes. For this purpose, we will maintain a hash table which detects any changes in the file system (done internally or externally). In case a change is detected, we know the some files have been changed in context of content or permissions, in which case, we will update the backend server about it which will respond to the frontend.

All the users are isolated from each other and have their own share of privileges to access the storage pool



The storage pool is the specified directory where these operations need to happen. Here, files are stored and managed. This needs to be specified in the YAML configuration during setting it up.

IT Teams can perform CRUD operation on the Storage Pool and have total control over it



Frontend allows users to give hierarchal permissions to other users. For example, is Bob has access to /bob, then he is allowed to give Alice access of /bob/alice-interest with read/write permissions

When a new user has to be created, only IT Team is able to do it.
The IT teams portal has options to create a news user and assign privileges.
When IT team creates a new user, a user on the Linux file server is created and no permissions are assigned. Further on, more privileges can be assigned.

The given architecture portrays a single file server. A similar design can be derived from this for multiple servers and the backend will do the work of orchastration.

The Linux User Management Approach for ACL

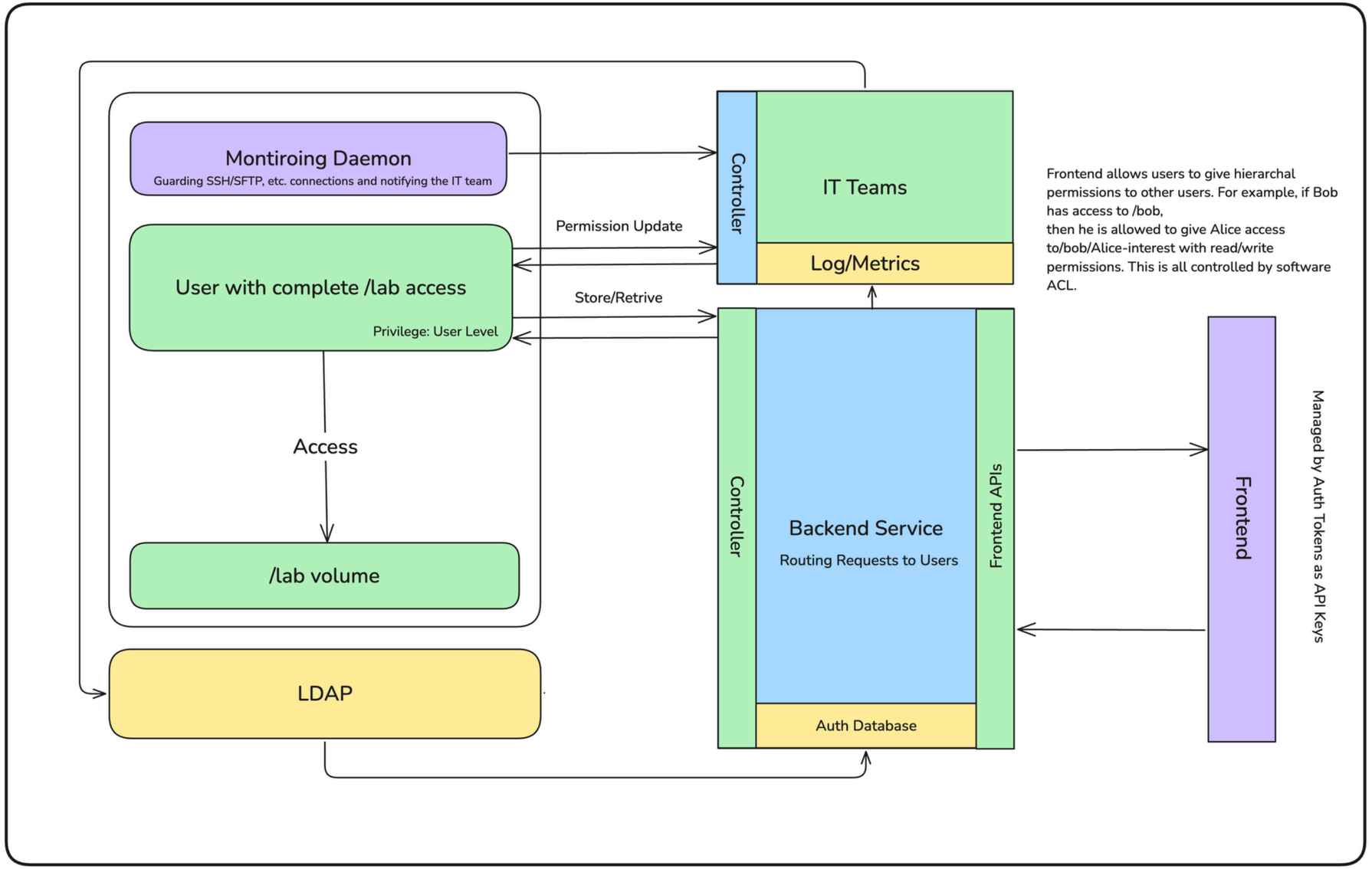
The Current Idea was based on using Linux's Own User control which can be used as ACL on the File Server

Pros: It's hard and secure, well maintained by Linux developers
More lower-level protection from unauthorized access
Since users are not created frequently, this can be ideal for maximizing security

In this case, the problem is boiled down to an optimization problem, which can be tailored as per requirements (like manually defining operation or using system APIs)

Revision 1 (with incorporated changed from first meeting)

Proposed System Design for 2nd Approach



The Software Level ACL

In this case, we just have one user on file server running an application that has complete access to all the files.

Access is provided based on the privileges reflected in the database. This approach is simple to build but has lesser isolation of files and directories.

LDAP servers and other auth mechanisms can be used here

Proposed Tech Stack - All are Questions

Golang for the backend server, APIs, daemons, and data-intensive applications like logs streaming

Reason: Performance, Simplicity, Developer Productivity, Safety, Stability, Powerful Standard Libraries, Support from Google, Best in Class Concurrency Support [My Most Proficient Language]

Python for Data Analytics and IT Operation Backends

Reason: Simplicity, Huge Support from Analytics GUIs and Data Processing Libraries [My First Language and Oldest Used]

C or Rust for Lower Level Operations

Reason: While interacting with filesystems, we might need lower-level access for managing permissions and for security purposes, optimizations

Proposed Tech Stack - All are Questions

Vanilla Javascript for Frontend Development

Reason: Since we need to keep our frontends lightweight, we would prefer using Vanilla JS or, in case of higher requirements, NextJS due to its performance and developer base

SQL and NoSQL Databases (self-hosted for user management)

Reason: This really depends on how we are using it and how fast we need it to be. This can be decided later since I am comfortable with any of them, and each has their own pros and cons

Docker for Deployment

Reason: Since we need to bundle up our packages and provide cross-platform support over various Linux distributions, Docker would help a lot. More focus can be given on configurable and automated deployments

Docker would be the second installation priority. Emory prefers deployment with tarballs with source code for security reasons.

Documentation Stratergy

Documentation

Reason: Sphinx Docs is a popular document framework that is simple, elegant, lightweight, and perfect for our application. Development and Deployment times are very short. Furo is my favorite theme.

Here, we will have two sections: User Docs and Code Docs

Product Docs - This is for non-technical users who want to learn how to use this tool from the frontend side if they have any issues. This is like the manual researchers use if they are having an issue. This also contains deployment instructions for IT Teams.

Code Docs - This is for IT Teams who want to use APIs provided in the Backend and want to develop our product.

Core Considerations after 1 Revision

We need users with privileges to grant other users privileges as per their requirements and not disturb the IT Team.

We already have LDAP installed in the network - so authentication based on it would be preferred.

The second system design is preferable for faster development and completion of the requirements.

No need for a ticketing system.

Installation over source code is preferred with tarball releases - docker would be a later priority.

No need for a security monitoring daemon.

Aditya’s Requests: I need the mid-level system design of current Emory’s implementation of the service. Would also like to know more constraints and specific requirements based on the proposed designs.