PAMEx: A Compiler and Tools to Support a More Flexible Security Policy for Simpleflow

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin–La Crosse

La Crosse, Wisconsin

by

Daniel Weninger

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

May, 2023

PAMEx: A Compiler and Tools to Support a More Flexible Security Policy for Simpleflow

By Daniel Weninger

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Prof. W. Michael Petullo Examination Committee Chairman	Date
Prof. Samantha Foley Examination Committee Member	Date
Prof. Elliott Forbes Examination Committee Member	Date

Abstract

Weninger, Daniel, "PAMEx: A Compiler and Tools to Support a More Flexible Security Policy for Simpleflow" Master of Software Engineering, May 2023, (W. Michael Petullo, Ph.D.).

This manuscript describes the design and implementation of PAMEx, a custom-built Linux security tool suite that uses several aspects of modern Linux security modules to enforce its policies. The tool suite uses extended attributes to define file security information, a custom Pluggable Authentication Module (PAM) to define actions that occur when a user authenticates on a system, and a custom compiler developed with Flex and Bison. PAMEx was developed as a security enhancement for SimpleFlow, a project that uses information flows to enforce policies. The way that PAMEx provides this enhancement to SimpleFlow is by being able to classify and compartmentalize file security as hierarchical levels, and non-hierarchical labels rather than the binary policy that SimpleFlow currently has in place. As a simulation, a tool was created and given the name Oracle that, when queried, will output a response as to if a user has access to a particular file. This paper gives an overview of the design and implementation choices made during the development of PAMEx and future work that could be done to maintain and improve the project.

Acknowledgements

I would like to give my sincerest gratitude and thanks to my advisor, Dr. Petullo. Without whom I would not have been able to get as far as I had or learn as much as I did during the development of my capstone project.

Table of Contents

List of Figures

Glossary

Agile

A project management philosophy which uses an iterative approach. Each iteration is a working product whose next iteration semantics can be changed [2].

Bison

A parser that translates a BNF grammar from the tokens that a lexical analyzer makes into groupings of statements. The statements determine which functions to perform [5].

Extended Attributes

Key-value paired metadata attached to files. Extended attributes are written to in PAMEx to store the security level and labels attached to the file.

Flex

The program used in PAMEx to create a lexical analyzer or scanner. Flex creates a character stream from defined, recognizable tokens and therefore outlines legal words that can be used in a system [5].

Label

A non-hierarchical compartment defined by the privileged administrative user. A file can have zero or more labels in addition to a level to further define its security. For a user to access a file with PAMEx security labels, the user must have the same corresponding labels.

Level

A hierarchical compartment defined by the privileged administrative user. Each level in a PAMEx system has its own placement and each file on a system can be assigned a level. To access a file with a PAMEx level, a user must be assigned a level whose placement is equivalent to the file's level or higher.

Linux Security Module

A framewok integrated into the Linux kernel which provides access controls to process files and other aspects of the Linux system [7].

PAMEx

The Linux security tool suite developed to enhance the SimpleFlow project.

Level Placement

The hierarchical value of a level in PAMEx. Placements start at level zero for unrestricted access to a file and relationally go up by one. No two levels on the same system can share a placement value.

Pluggable Authentication Module (PAM)

A separated process or task which is invoked during a user's authentication process on a system. PAM modules are stacked to be invoked consecutively [3].

Scrum

A framework for the Agile method which uses a cyclical approach called a sprint. The sprint team consists of a scrum master, one or more product owners, and developers. Scrum outlines actions that occur to achieve an iterative product at the end of each sprint [12].

Sprint

One cycle in the Scrum process. The PAMEx sprint time was two weeks.

sudo-proc

The fake process file directory that PAMEx's PAM module creates. PAMEx creates this directory because the real proc directory is expecting an SELinux formatting which PAMEx does not use.

1. Introduction

1.1. Background

The objective of PAMEx is to work in conjunction with and enhance the pre-existing project SimpleFlow, an information-flow-based access control system. Even in the modern day, most access control systems simply disallow read or write access to confidential files. The SimpleFlow project was developed to go beyond most access control systems and instead allow malicious users to access confidential files up until the point of exfiltration so that SimpleFlow can monitor the user's activity. The way that SimpleFlow achieves this is by working on top of the Linux Security Module (LSM) interface and rather than simply disallowing read or write access like most access control systems do, SimpleFlow instead marks a file as either confidential or non-confidential. If a confidential file is wrongly accessed, the malicious user's actions are recorded as they pass through the system as information flows. The wrongly accessed information is then prevented from being exfiltrated by SimpleFlow's network filter. SimpleFlow begins tracking the attacker's actions as soon as they attempt to access the confidential file and therefore can capture the attacker's intentions. This leaves no room for the malicious user to come up with a fabricated excuse as to why they were trying to exfiltrate sensitive information. As an example of SimpleFlow's effectiveness, when put to practical use during the 2016 Cyber-Defense Exercise, the SimpleFlow program proved to perform well with a small overhead. While the practical exercise did not make use of all SimpleFlow's functionality, it was able to successfully use SimpleFlow's network filter which made it easy to observe any attempt at exfiltrating data. Within this exercise and other testing, SimpleFlow impressed attackers who were surprised to find their exfiltration attempts had failed [11].

Figure 1 was pulled from the paper "Studying Naive Users and the Insider Threat with SimpleFlow." to depict SimpleFlow being used to prevent an exfiltration attempt of sensitive data. The example starts with a file on the kernel marked as `secret` which contains sensitive information.

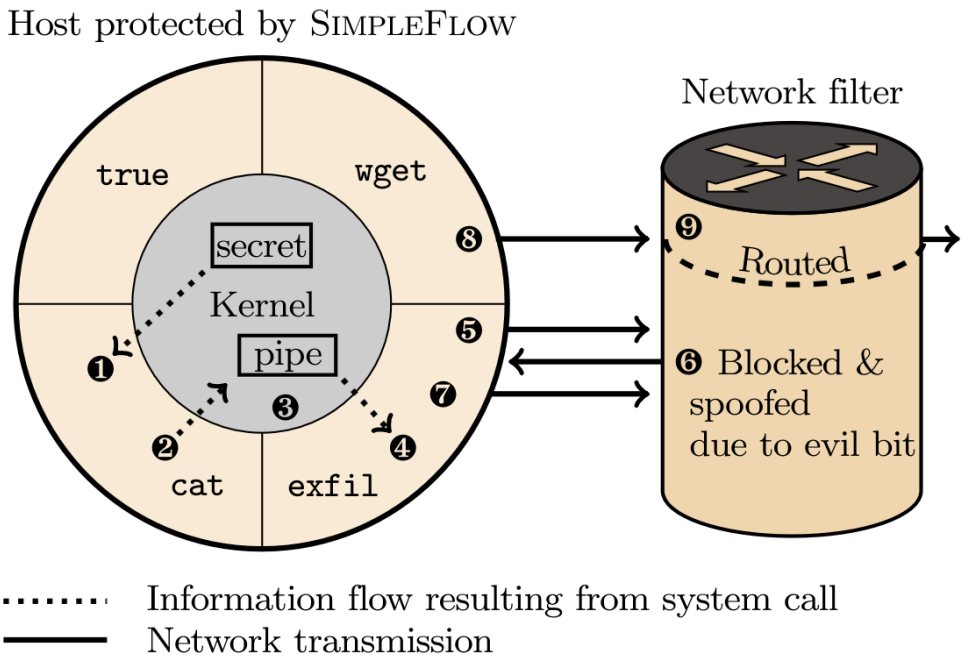

Figure 1. SimpleFlow Being Used to Prevent Exfiltration of Sensitive Data (from [11])

❶ A malicious user uses the `cat` process to open and read the confidential data in `secret`. Now any further actions processed by this user are tainted by SimpleFlow by being marked with an evil bit and are monitored by the SimpleFlow system.

❷ The malicious user attempts to pipe and exfiltrate the information in `secret`.

❸ The pipe is forked as a child from the `cat` process and therefore is also marked as malicious activity by SimpleFlow.

❹ The `exfil` command reads `secret` and becomes tainted with the evil bit.

❺ The malicious user attepts to exfiltrate the information from `secret` to a personal server. The sent packet containing `secret`'s information is marked with an evil bit to be easily recognizable by SimpleFlow as an unsolicited action.

❻ SimpleFlow's network filter notices the malicious user's attempt to exfiltrate the information from `secret`. SimpleFlow sends a fake response to the request to make the malicious user think that the process is being continued.

❼ The malicious user's exfiltration attept on `secret`'s data is ultimately blocked by SimpleFlow's network filter.

❽ The non-threatening process `wget` is performed on the system.

❾ SimpleFlow's network filter behaves normally and allows the `wget` process to continue.

The above numerical SimpleFlow example items were summarized from [11].

While effective, the research prototype of SimpleFlow is limited in that it cannot further specify the security policy of a file beyond being confidential and non-confidential [11]. A resolution for SimpleFlow's issue of a binary security system was therefore sought out and as a solution a customized Linux security tool suite was devised. The system would allow the privileged user to define hierarchical levels and non-hierarchical labels on a system and assign them to both system users and system files. This security policy was inspired by the US security classification system [10] by following the structure of having both levels and labels for security. The policy states that both system files and system users may each have one security level and zero or more security labels.

As a running example, pretend that there is a software company that is working on innovative technology that they want to be kept secretive so that all files related to the technology are on a need-to-know basis. With this proposed security policy, the company could define a system saying which staff have access to what files. For example, the privileged administrative user who oversees creating the security policy could define the hierarchy levels for the company from lowest to highest as `public`, `general_staff`, `developer`, `administrator`, and `executive_staff`. In this example, the company has also split the technology into several projects, three of which are named `alpha`, `beta`, and `charlie`. If Alice is an `administrator` for both projects `alpha` and `beta`, then, with the newly proposed security policy, she would be given the hierarchical level `administrator` and the non-hierarchical labels `alpha` and `beta`.

Continuing the above example, pretend that there is a file to outline how to begin development on project `alpha` called `alpha_dev_instructions.txt`. The privileged administrator could assign `alpha_dev_instructions.txt` the security level `developer` and the security label `alpha`. Alice, who has the security level of `administrator` which is placed higher than `developer` in the level hierarchy and who contains the security label `alpha` can access `alpha_dev_instructions.txt`. Pretend that there is also a developer in the company named Bob who is a developer for projects `beta` and `charlie`. Bob would have the security level `developer` which is a proper label to be able to access `alpha_dev_instructions.txt`, but Bob would not be given the label `alpha` so therefore would ultimately not have access to `alpha_dev_instructions.txt`.

The newly proposed security policy of levels and labels would allow multiple groupings of users such as, in our example, more `developers` who have access to project `alpha`, more `administrators` that have access to multiple projects, and files that are available to all staff members. With the newly proposed security policy, a system administrator would therefore be able to flexibly compartmentalize all the files and data that users would have access to on a system. Initially, it may seem like other previously-developed access control models may be sufficient to create a security policy that is ideal for SimpleFlow such as role-based access controls (RBAC). However, these models are not as ideal as the newly proposed PAMEx secuirty policy. As stated above, most RBAC simply disallow reading or writing to confidential files on a system, but SimpleFlow wants to instead allow that access so as to be able to monitor an attacker's actions. SimpleFlow's primary job is to record a malicious user's activity on the system and ultimately disallow exfiltrating confidential information via its network filter. With SimpleFlow having such a particular goal, only a custom security policy such as PAMEx will allow SimpleFlow to continue to monitor a system the way that it was designed to without interfering with SimpleFlow's actions. Ideally, the tool suite would be developed as a standalone Linux Security Module so as to be tailored to but not mutually exclusive for SimpleFlow though. The development of the custom

kernel module that would allow the tool suite to be standalone or ported into SimpleFlow was left to future work.

1.2. Motivation

In 2022, the author reached out to his advisor, Dr. W. Michael Petullo showing interest in the SimpleFlow project and learning more about its shortcomings including the binary security policy currently in place. Having pursued an emphasis in cybersecurity alongside his degree, the author had already had an introduction to Linux security but wanted to learn more. The motivation of PAMEx was to aid in and further enhance the development of the SimpleFlow project as well as to learn more about Linux security at a system level and how it can be added to and manipulated. Building a security policy tool system from the ground up would give a foundational experience of that aspect of security systems.

Together with his advisor, the author devised and proposed an alternative solution for the SimpleFlow security policy than the one currently in place. This policy would continue to work on top of the LSM interface as SimpleFlow already does and therefore be integrated into SimpleFlow easily. The policy's syntax and tools would be user friendly and allow for much greater malleability while having a small footprint. Another requirement was to ensure that PAMEx's integration would not need to be exclusive to SimpleFlow but could be used on many Linux based systems. Developing a project of this scale is also a perfect opportunity to research, learn about, and practice the development cycle of a project with industry standards.

2. Design

2.1. Development Process

The basic concept and outcome of the PAMEx project was well defined from the beginning, but there were several unknowns about how to achieve several goals. It was also decided early on that the project would need to be broken into several segments to achieve what was necessary but also to allow a privileged user of PAMEx to have the customizability to implement PAMEx on different systems effectively and isolate the different processes of the security system. Therefore, the project implementation would need to be flexible, but efficient and would need to be treated as several tools working together to achieve a desired end state. Once fully designed, PAMEx would become a tool suite with several components including a customized compiler, a file labeler and user database builder, a PAM module, the Oracle tool for determining a user's access to a file, and an extraneous file labeler to make retrospective file labeling easier. With all these factors in mind, it was decided that an Agile approach was the best implementation strategy for this project and that using the Scrum framework was the best way to achieve this. Agile is a model which focuses on an incremental approach to programming by developing a working product in iterations rather than all at once [2]. In addition, the Agile method via a Scrum framework is immensely popular among many industries [4] and therefore, Agile and Scrum were valuable skills to research alongside the development of the project.

Two sources were researched and followed as guidelines for the Scrum process throughout the project. The first source acted as an introduction to the Scrum process and is a blog entitled "What is Scrum?" on the official Scrum website [12]. The second source is a book entitled "The Scrum Field Guide" authored by Mitch Lacey, which acted as an ongoing guide [4]. In the first source it was found that Scrum is a cyclical process in which there are groups of individuals that work to achieve a common goal. Figure 2 was taken from the first source and depicts a typical Scrum cycle. Each cycle is referred to as a sprint and during each sprint, it is expected that four events occur: first, the product owner divides up a complex problem and puts the divided pieces into a backlog, second, the Scrum team creates an incremental product, third, the Scrum team inspects the results and adjusts for the next sprint, and fourth, the process is repeated [12].

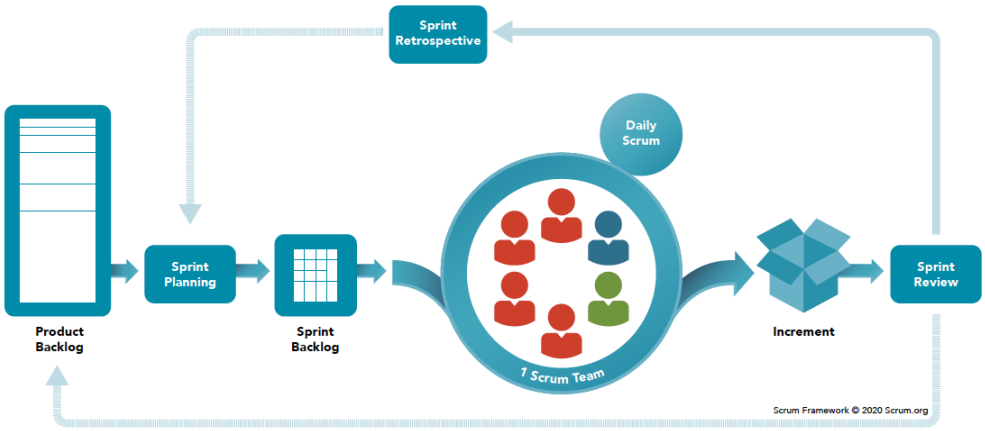

Figure 2. A Typical Scrum Cycle (from [12])

Typically, in a software project setting, the Scrum team is divided into three groups: the Scrum master, the product owner, and the developers. The Scrum master's primary job is to help the team focus on creating a high-value product increment and remove anything that may impede this process. The product owner's primary job is to prioritize the work of the developers. The developers' job is to work on completing the product increment for each sprint. The idea is that the Scrum team has all the skills needed to complete the task given, and each member is an expert at contributing to at least one of the aspects of the project [12].

At the beginning of each sprint, the product owner's job is to perform backlog grooming which means that they determine which parts of the product should be completed next and how much development should get done for that particular sprint. For each sprint, the team is expected to meet regularly for what is known as a standup where they talk about the progress that has been made on the product increment and any impediments that they have. At the end of a sprint, the team has a retrospective meeting where they talk about the previous sprint: both what went well and what could be improved [12].

The PAMEx team only consisted of two members and as such, some modifications needed to be made to the Scrum process. The Scrum role of the product owner was filled by the author's advisor and the role of the developer was filled by the author; the author also filled the role of Scrum master by leading each of the standup meetings. Each sprint was decided to be two weeks long, and during that time, the Scrum team would meet once per two weeks instead of performing a daily standup together. This decision was made primarily due to the schedule of both parties and nature of the project. The project team only consisting of two members led to more direct communication than a typical industry software product would. The Scrum team members both had times when communication needed to occur before the scheduled sprint time such as for a modification in an aspect of PAMEx's design or if the developer had a blocker during development. For these instances, often the team found it sufficient to perform communication via email but occasionally set up an online conference call.

For each biweekly meeting, the team followed Scrum procedure and would perform a standup by talking about what was completed during the sprint [4] and talk about any blockers that were present in the current incrementation. Part of the standup process included presenting research

findings on the Scrum process as well. After the standup, the team would perform a retrospective and talk about what went well and what needed to improve for the next sprint. Finally, the team would talk at a high level about what should be completed for the upcoming sprint to complete the next iteration of the project.

The team chose to track this information with the tool Jira. Within Jira, the developer, acting also as the Scrum master divided the functionality requirements of the project into Jira tickets. The tickets contained information about each piece of the project including the ticket number, the ticket type such as user story, sprint task, enhancement, or bug, a description of the ticket which often included a definition of done for each stage of the ticket, the estimated story points, the amount of development time spent on the ticket, the start date, the end date, the priority of the ticket, and the relationship that the ticket has to other tickets. Dividing the project in this way is common when using the Scrum approach so that a team can easily see what is the next highest priority in the development process to complete the next iteration of the product [4].

Throughout the project, a total of 35 Jira tickets were created with a total of 103 estimated story points in 15 sprints. That equates to an average of 6.87 points per sprint or roughly one point every two days. Figure 3 is a chart depicting the amount of total user stories remaining to complete compared to the date of the ticket's completion.

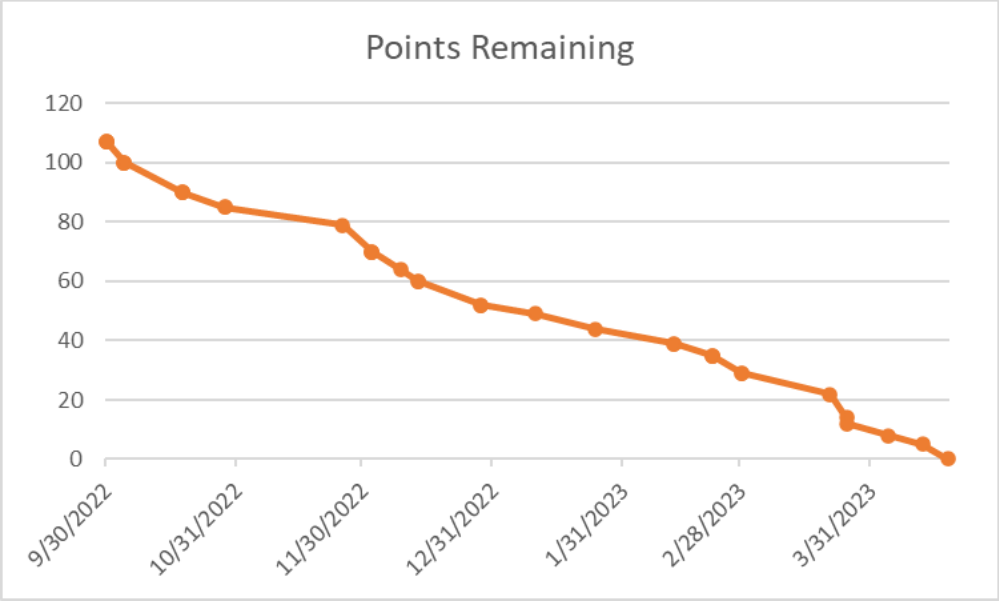

Figure 3. User Story Points Remaining Compared to Date

It is easy to tell from the diagram above that there is a steep downgrade toward the end of the project, depicting that a lot of user story points were completed near the end. This is partially because a few large user stories and tasks were designed to be completed at the end of the project. Some of these tasks included creating documentation for the PAMEx tool and writing a formal manuscript for the tool.

In a Scrum setting, after each sprint, the team should have developed and produced an incremental cohesive product. Even if the incremental product does not have all of the functionality of the completed product, it should be usable [12]. PAMEx's natural separation yet cohesiveness of

different tools made it easy to use the incremental approach, so often each sprint was dedicated to the development of a separate PAMEx tool. There was not such a natural separation for every sprint however, as some of the aspects of the project, for example the compiler, were much bigger than others. Therefore, incremental, cohesive parts of the larger pieces were developed separately. To continue with the example of the compiler, the compiler was divided into five separate sprints and therefore into five different components. The components included defining the syntax and grammar for the policy including creating a prototype EBNF diagram, creating the Flex lexer, creating the Bison parser, and two components for implementing the semantic functions that would be triggered by the grammatical parser. The separation of the semantic functions occurred naturally because the grammar already essentially had an informal separation: definition statements and assignment statements. Therefore, during the first of these two sprints, the semantic functions pertaining to the grammar that defined security levels and labels were developed, and the during the second, developing the semantic functions pertaining to the grammar that assigned the security levels and labels to users and files was focused on.

During each day of a sprint, more aspects of the Scrum process were practiced. For each day of development, the product's ticket board which included the product backlog would be viewed and updated. This served as a replacement for the daily standup that would have otherwise been performed. For each user story, a definition of done was created for each of the stages of development. The stages included the development process, the review process in which the product increment was reviewed by the product owner/advisor, and the testing process. It was found that a definition of done is an integral part of the Scrum process as it defines when each user story is completed [4][12].

Although the Scrum process had been modified, the spirit of Agile development via the Scrum method was kept and the core principles were still followed during the development of PAMEx. In doing so, the PAMEx project was able to be developed at a manageable, incremental pace to focus on each aspect of the project independently.

2.2. Technologies

All components of the PAMEx tool suite were developed in Vim or Visual Studio Code using the programming languages C, Flex, and Bison. C was chosen as the primary language for the development of PAMEx for several reasons. A major reason is that SimpleFlow, the project for which PAMEx was designed to enhance, was built using C [11] and therefore, it would be easier to integrate PAMEx into the SimpleFlow system. Secondly, PAMEx was designed to be tailored to Linux security systems. The C programming language has several libraries to be able to achieve this. While not all aspects of the Linux security system that were needed to be developed for PAMEx were fully understood at the beginning of the project, it was known that the security system tools that were being sought out to develop could be created using C [8][9][14][13]. Third, from the first design of the PAMEx system, it was known that a custom compiler would be created. This is a process that was familiar to the developer using the C language with Flex and Bison.

2.3. PAMEx Design Overview

The basic flow of PAMEx was designed as follows. First, a privileged user writes a PAMEx language file. The PAMEx parser takes this file as input and produces two text files as output: one which contains a database of the levels defined in the input file and their corresponding placements in the level hierarchy, and one which contains file and user definitions in a format that is readable by the File Labeler and User Database Builder (FLUDB). The definitions file is then read by the FLUDB line by line. From this file, the FLUDB can write extended attributes attached to the targeted files as metadata. The extended attributes will then contain the security information specified in the PAMEx language file including both levels and labels. The FLUDB tool also creates a text file that acts as a targeted users database, containing all the users defined in the PAMEx language file and their security information. It is assumed by PAMEx's PAM module that when a user logs in to a system that has PAMEx installed on it, the system will have a targeted users database file on the system for the kernel to read and interpret. Upon a user logging into such a system, the system can trigger the custom-made PAM module to write the user clearance information to the currently running process. Any action then performed by the logged-in user will be branched from the currently running process and will therefore contain the user's clearance information [8]. All that a kernel would need to do when a user tries to access a file is cross-check the currently running process' user security information with the file's security information via extended attributes to ensure that a user has the correct clearance level and labels and allow the user to export the information from files that they have access to. This flow is depicted by Figure 4 below.

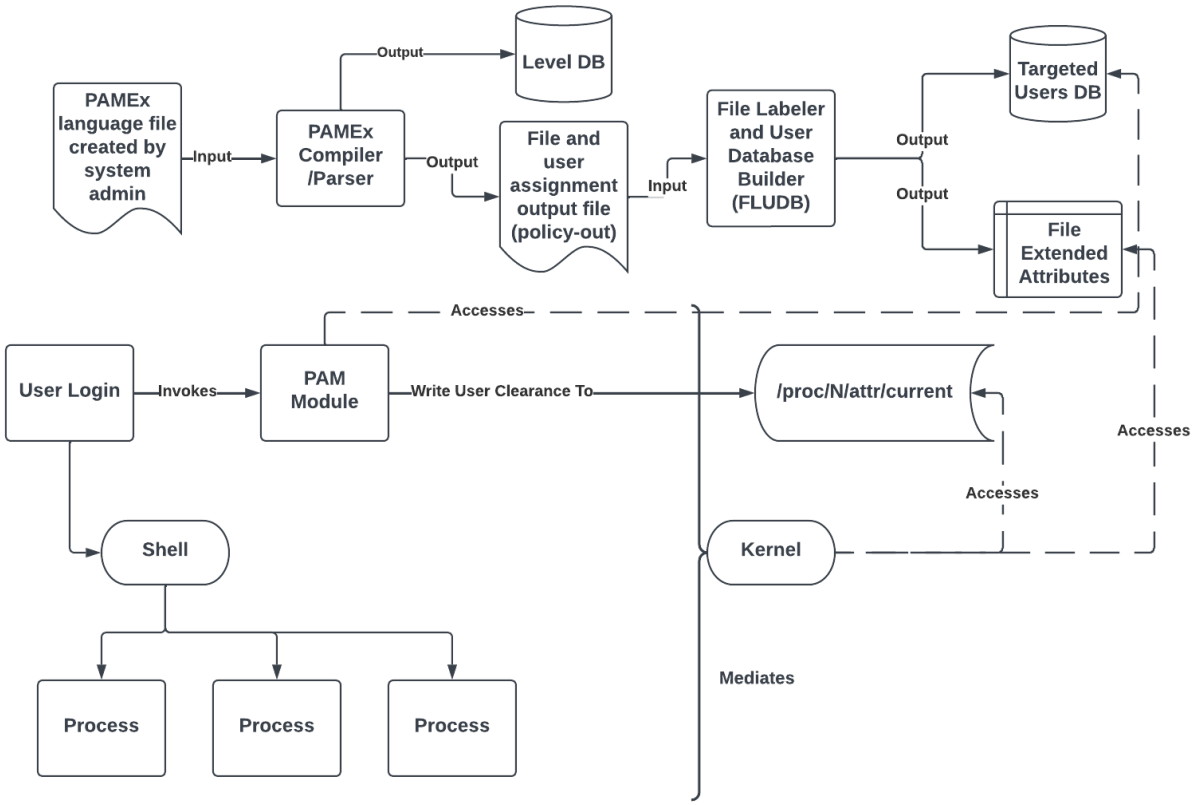

Figure 4. PAMEx Depicted at a High Level

The original design for the PAMEx project consisted of the compiler, a C tool to interpret the compiler's output, an unknown way for the file information to be stored on the system, and a PAM module to collect and persist the logged in user's information. It was known early on that enforcing a security policy defined by PAMEx would require a custom kernel module. When the project first began, it was intended to also build this kernel module, but as the development continued, it was decided that much of the security policy functionality would be in place without the kernel and PAMEx's functionality could be demonstrated without it by simulating the kernel's primary functionality with a custom-built tool called Oracle.

The implementation of the PAMEx tool suite followed a path similar to the cycle of the implementation of a security policy on a PAMEx system. It began first with the compiler by using Flex for the complier's lexer and Bison for the compiler's grammatical parser. Along with the parser, the symbol table was implemented to keep track of the different usages of variables and their data types [5]. In the first stage of the compiler's implementation, a level database was not created as a biproduct of the compiler's output because the issue that the level database would later solve of ensuring that there was only one set of levels on a system and that they were all to be checked by one source was not foreseen yet. This would change near the end of PAMEx's development during the creation of Oracle.

Following the implementation of the compiler, research was done to find that the way that many Linux security modules label files with security information is through the use of extended

attributes [13]. Extended attributes were found to be accessible through both command line processes and using a C library. The complier output interpreter tool was therefore created whose job it was to output the PAMEx policy assignment information. The file was later renamed to the File Labeler and User Database Builder or FLUDB to better describe the functionality of the tool.

The PAM module was then researched and created using the Official PAM Modules as a reference [8]. It was decided to implement PAMEx's PAM module on a virtual machine to not interfere with and possibly break the login process of a real system.

Research was done to grow an understanding of the complexity of creating a custom Linux kernel in comparison to the amount of functionality that it would be able to demonstrate [7]. As a result, the informed decision was made to not develop a custom kernel for PAMEx and to instead simulate the PAMEx tool suite using another custom-developed C tool named Oracle. With Oracle in place, PAMEx as a working tool suite could be fully demonstrated from end-to-end.

3. Design and Implementation of Tools

3.1. PAMEx Compiler Design and Implementation

At its core, PAMEx interprets and enforces custom, privileged-user-defined security information. Therefore, it was decided that it would be best to build a custom compiler to serve the purpose of defining PAMEx security information for several reasons. Namely, the compiler achieves multiple tasks at once after simply reading in an input file. The two primary tasks that the PAMEx compiler achieves are creating the output file to later be read by the File Labeler and User Database Builder (FLUDB) and second, to create the hierarchical level database to be used by PAMEx later when considering whether a given user with a given clearance has access to a given file. The level database can only be created by the compiler because the compiler's input requires the privileged, administrative user to define all the levels and their relative placements. Further tools in the PAMEx system require all the defined levels to be written to the level database so that the levels can accurately be referenced and compared to.

The PAMEx compiler was also designed for a similar purpose as many programming languages: to achieve its specific task in an efficient, flexible, and human-centric manner while clearly defining its rules [6]. With the compiler in place, the grammatical and syntactical rules are clearly defined so that there is no confusion about the abilities that PAMEx has. The PAMEx compiler consists of a lexer, a parser, semantic functions to be invoked by specific grammatical sequences found by the parser, and a symbol table to keep track of the variables and their corresponding information. The PAMEx language is not as sophisticated as many programming languages in the sense that it only has one scope, so the symbol table does not need to keep track of the scope of the variables [5]. For PAMEx's symbol table, a hash table data structure was found to be sufficient.

The PAMEx language grammar was originally designed using an EBNF diagram depicted in Figure 5 and was later revised so that it could work more easily with Flex and Bison as shown in Figure 6. The PAMEx language file was designed to have syntax and grammar that is easy to understand and feel familiar to programmers. The compiler of PAMEx is used by a privileged user who inputs a file that both defines and assigns hierarchical levels and non-hierarchical labels to files and users. This file will further be referred to as the PAMEx language file.

```
program ::= {component}
component ::= [comment] {definitions} [comment] {assignments} [comment]
comment ::= "#" (.)*
assignments ::= {assignment}
assignment := user-assign | file-assign
user-assign ::= "user-assign" level-name [label-list] "->" user";"
file-assign ::= "file-assign" level-name [label-list] "->" file";"
user ::= var
file ::= var
label-list ::= "[" labels "]"
labels ::= label-name{"," labels}
definitions ::= {definition}
definition ::= define-label | define-level
define-label ::= "label" label-name";"
define-level ::= "level" level-name operation";"
level-name ::= var
label-name ::= var
operation ::= "(" (setres | opvar) ")"
opvar ::= op var
setres ::= "set" res
var ::= (word | symbols) {number} {var}
symbols ::= ["." | "_" | "-"]+
number ::= [0-9]+
word ::= [A-Za-z]+
op ::= "<" | ">"
res ::= "restricted" | "unrestricted"
opt-ws ::= [WHITESPACE]+
```

Figure 5. The Inital EBNF Diagram Created for the PAMEx Language

```
%%

stmt_list   :   stmt_list stmt
            |   stmt {};
stmt        :   USERASSIGN level_name ASSIGN user';' { ... }
            |   USERASSIGN level_name label_list ASSIGN user';' { ... }
            |   USERASSIGN label_list ASSIGN user';' { ... }
            |   FILEASSIGN level_name ASSIGN file';' { ... }
            |   FILEASSIGN level_name label_list ASSIGN file';' { ... }
            |   FILEASSIGN label_list ASSIGN file';' { ... }
            |   LABEL label_name';' { ... }
            |   LEVEL level_name op';' { ... };
label_list  :   '['labels']' { ... };
labels      :   label_name  { ... }
            |   label_name',' labels { ... };
user        :   id { $$ = $1; };
file        :   id { $$ = $1; };
level_name  :   id { $$ = $1; };
label_name  :   id { $$ = $1; };
op          :   '('set_res')' { $$ = $2; }
            |   '('op_var')' { $$ = $2; };
op_var      :   CMP id { ... };
set_res     :   SET res { ... };
res         :   LEVELLIT { $$ = $1; };
id          :   ID { $$ = $1; };

%%
```

Figure 6. A Partial Code Snippet Taken from PAMEx's Yacc File Depicting the PAMEx Grammar. Semantic Functions have been Omitted.

3.2. PAMEx Language File Design

Syntactically, the PAMEx language file was designed with two distinct actions in mind: defining the PAMEx security information and assigning the security to users and files. The definition actions would allow the privileged user to define the hierarchical levels including the levels' names and what placement they are in the hierarchical chain in relation to other levels. Labels on the other hand, are compartments and are standalone qualifications that do not have any hierarchical value [10]. Therefore, the only information that a privileged user needs to provide when defining a new label is its name. Designing the PAMEx language file syntax and grammar to be user-friendly was an important consideration. Therefore, the definitions section's syntax and grammar are inspired by many programming languages by consisting of a keyword to be used as a type, a variable name, an operation for level definitions, and a semi-colon to mark the end of the statement. For example, in PAMEX, a level definition followed by a label definition might look like the following two lines.

```
level administrator (> developer);
label alpha;
```

For the assignment statements, the PAMEx language file depicts which level and labels are assigned to users and files. At any point, any user and any file may only be assigned one level to have a distinguished security group but may have zero or more labels to further define the security assigned. Assignment statements were designed to be minimalistic, easy to read, and also resemble that of other programming languages. The grammar for an assignment statement includes the function to be performed, the name of the level to assign to the entity, the list of labels to assign, an assignment operator, the name of the entity, and finally a semi-colon to depict the end of the statement. For our previous example with the technology company, the `administrator` Alice and `alpha_dev_instructions.txt` could be assigned to the system using PAMEx with the following lines.

```
user-assign administrator [alpha, beta, charlie] -> Alice;
file-assign developer [alpha] -> alpha_dev_instructions.txt;
```

From these assignment statements, Alice would be assigned the level `administrator` and the labels of `alpha`, `beta`, and `charlie`. `alpha_dev_instructions.txt` would be assigned the level `developer` and the label `alpha`.

The division of the definition and assignment statements in the PAMEx language file is not a hard one. Much like many programming languages, definitions and assignments are allowed to be intermixed throughout the PAMEx language file with the only grammatical requirement of a level or a label needing to be defined before it can be assigned to a user or a file. With this qualification in mind, our complete PAMEx language file example with the user Alice and `alpha_dev_instructions.txt` could look like Figure 7.

```
# This is a comment
# alpha_dev_instructions.txt and Alice example

# Definitions for alpha_dev_instructions.txt
level public (set unrestricted);
level general_staff (set restricted);
level developer (> general_staff);
label alpha;
label beta;

# Assignment of security for alpha_dev_instructions.txt
file-assign developer [alpha] -> alpha_dev_instructions.txt;

# Further definitions for Alice
level administrator (> developer);
level executive_staff (> administrator);
label charlie;

# Assignment of security for Alice
user-assign administrator [alpha, beta, charlie] -> Alice;
```

Figure 7. A Full Example of the PAMEx Language File Contents with User Alice and alpha_dev_instructions.txt

One of the major decisions that had to be made when designing the PAMEx language file was how the level placements could be defined syntactically and grammatically. In PAMEx, levels are a tiered, hierarchical system and therefore, it was decided that it would make sense for the levels to be defined in relation to one another. For example, a PAMEx language file could be written so that a level defined with the name `general_staff` is a lower placement than a level defined with the name `developer`, which is a lower placement than a level named `administrator`. For the privileged user to define levels relationally, a base level needed to be defined first – a level whose hierarchical placement is set independently rather than relationally. Therefore, the PAMEx language file allows for three different operations: the first being the operation `set` which allows a level to be set to a base level of either `unrestricted` which is hierarchical placement zero or `restricted` which is hierarchical placement one, as well as the operations $>$ and $<$ which define that a level is either one hierarchical placement greater than a previously defined level or one hierarchal placement less than a previously defined level.

Another major development decision was that no two levels would share a hierarchical placement. In other words, if there are three levels `a`, `b`, and `c` they could be defined as the following lines.

```
level a (set restricted);
level b (> a);
level c (< b);
```

Before level `c` is defined, level `b` is one hierarchical placement higher than level `a`. This is because level `b` is defined using the $>$ operation to depict that level `b` is one placement higher than

level a in the hierarchy. On the next line, when level c is defined as one placement less than level b using the < operation, level c does not take the same placement as level a, but rather the level placement of b shifts and makes room for the new level. Therefore, the level placement order in this scenario would be first, at the lowest level placement, level a, then one placement higher is level c, then, with the highest-level placement, level b.

With this relational method, the privileged user can easily and clearly define a hierarchical level structure using PAMEx, and the grammar is designed in such a way that is easy to interpret. The language file is also minimal in the sense that each character sequence in the grammar has a purpose, but it is powerful in the sense that the security can be defined to meet many systems' needs.

The PAMEx compiler is also convenient in the sense that not all users and files on the system need to be assigned PAMEx security clearance information. For any user or file that is not deliberately assigned a level, the level of the file or user is treated as a level at placement zero or unrestricted. In other words, if a file is not assigned a level, then PAMEx does not affect the access that a user has to the file. If a user is not assigned a level, they cannot access any PAMEx secured file that has a level higher than the set placement of unrestricted. This was an important decision to allow for PAMEx to work with already defined security policies on a system and to allow PAMEx to work out of the box for the privileged administrative user to only worry about defining the security policies for those users and files that it will affect.

Another notable point of flexibility with the PAMEx compiler is when assigning security information to files. Within the assignment statement, the file specified does not need to be simply the name of a file but can instead be the path to a file. This path is relative to a directory specified when using the FLUDB tool. Being able to specify the path to a file at this stage gives the privileged administrative user the ability to enforce a PAMEx security policy in multiple locations on the system.

3.3. The PAMEx Compiler Outputs Two Files: The Assignments File and a Level Database

When the compiler runs successfully, an output file is created containing all the user and file assignment information. This file is written in such a way that the computer will be able to easily read and interpret it, but is also relatively simple for a human to read because there is an evident pattern in the format. For our running example of alpha_dev_instructions.txt and user Alice, the PAMEx compiler output may look like the example in Figure 8.

```
FILE_LEVEL alpha_dev_instructions.txt developer:2
FILE_LABELS alpha_dev_instructions.txt alpha
USER_LEVEL Alice administrator:3
USER_LABELS Alice alpha
USER_LABELS Alice beta
USER_LABELS Alice charlie
```

Figure 8. An Example of a PAMEx Compiler Output File with User Alice and alpha_dev_instructions.txt

Each line depicts an assignment function that should occur. For example, the first line has the `FILE_LEVEL` keyword to depict that a file is to be assigned a level, then the path to the file, and finally, the level information including the name and its placement in the hierarchy separated by a colon delimiter. The level placement number is unnecessary for further functionality because all level information including their placement numbers are stored in the level database. The level placement numbers in the compiler output file are simply present as extra data for debugging and development purposes. When developing the FLUDB tool for instance, the level placements were used to compare and double check that the tool was gathering the appropriate level data.

In addition to the policy-out file, the PAMEx compiler outputs a level database file. The level database file was created because there is a need to compare the placement of two levels in future tools. The placement comparison was found to be most reliable if the level information persisted in one place on the system. It is essential to output the level database file at this stage because the PAMEx language file is the only place where all the levels are defined and ordered by placement. The levels are developed to be compared relationally and therefore it is necessary that every level is accounted for in some way. The levels database therefore captures a list of all the levels that have been defined in the PAMEx language file in hierarchical order. Each line of the level database file contains a level name and its placement number separated by a colon delimiter. For our running example of the technology company, the level database file could look like the example in Figure 9.

```
public:0
general_staff:1
developer:2
administrator:3
executive_staff:4
```

Figure 9. An Example of the Contents a PAMEx Level Database Outputted by the FLUDB Tool

3.4. Design and Implementation of The File Labeler and User Database Builder (FLUDB) Tool

It was determined from the beginning that a tool would need to be created to interpret the compiler's output and pass along the information given. The tool could do this by labeling the files with their respective security information and by creating a database file containing the system users' PAMEx security information. Therefore, the File Labeler and User Database Builder (FLUDB) tool was designed and created in C for PAMEx to do just that. To work flexibly, the FLUDB tool would need initial information pertaining to the locations of the input and output files. Therefore, the tool takes three input parameters from the administrative user which are as follows: the path to the compiler-created policy-out file, the directory that houses the files specified in policy-out which will be assigned the PAMEx security levels and labels, and the path for the targeted users database output file. The PAMEx FLUDB tool is limited in the way that all the files that are defined in the PAMEx language file need to be in the same directory or share a parent directory. The benefit to keeping all the PAMEx restricted files in one location are that the interpreter does not need to search through an entire file system to find the correct file, but only search for files that are within the specified parent directory.

One of the first hurdles that was faced during the development of the FLUDB tool was determining how to store the file's PAMEx security information including the file's level and various labels. To overcome this hurdle Linux security modules were studied, namely SELinux. It was found that SELinux uses a file property called extended attributes [13], so PAMEx followed suit. Extended attributes are a metadata feature of a filesystem that are built into many operating systems. They are lightweight features that allow for key-value pairs pertaining to a file's information and unlike most file metadata, extended attributes are not interpreted by the files themselves, but rather by outside programs. For those reasons extended attributes are a perfect system to store security information. Each file and directory on a Unix system has extended attributes so therefore any particular file can hold information about its own security [14].

Further inspired by SELinux, the C programming language's `xattr` library was found and utilized to achieve all extended attribute operations. Particularly, the PAMEx FLUDB tool uses the `setxattr` function to set both the key and the value of the extended attributes for both levels and labels on a file. The `setxattr` function requires root privileges and therefore the FLUDB tool needs to be run as `root` to write the files' security information [14]. Although at first seen as a

drawback, being required to run the FLUDB tool as `root` is useful in the sense of providing an extra layer of security in order for PAMEx levels and labels to be assigned to files on a system. In other words, the files cannot have security restrictions be added to them if the FLUDB tool cannot write to the file's extended attributes and the only users able to use the FLUDB tool are those with `root` access.

For the design of the extended attributes themselves, two different extended attributes were created for each file pertaining to PAMEx security information. One of the extended attributes is for the file's PAMEx level with the key `security.fsc.level` and the other is for the list of the file's PAMEx labels with the key `security.fsc.label`. The policy-out file which contains the information to write to the two extended attributes is written in such a way that the file assignment information is provided on separate lines but grouped together. Looking again at Figure 8 all the `alpha_dev_instructions.txt` assignment statements are grouped together and all of the assignment statements for Alice are grouped together.

Grouping the data in policy-out is a purposeful feature so that the file's attributes can be interpreted more efficiently. When the FLUDB tool reads a line that begins with `FILE_LEVEL`, it knows to assign or reassign a value to the extended attribute with the key `security.fsc.level` to the file specified. The value of the extended attribute is the remaining information in the same policy-out file line including the level's name and its placement. When the FLUDB tool reads a line that begins with `FILE_LABELS` it knows to append a new label to the value of the file's extended attribute with the key `security.fsc.labels`. The interpreter will read the rest of the line which will give the FLUDB tool the file to assign the label to, and then the label name. If the file is to be assigned more than one label, the following sequential lines will contain the additional labels. Figure 10 is an example of a file's extended attributes after the FLUDB file has written its security information.

```
# file: alpha_dev_instructions.txt
security.fsc.labels="alpha"
security.fsc.level="developer:2"
security.selinux="unconfined_u:object_r:user_tmp_t:s0"
```

Figure 10. An Example of alpha_dev_instructions.txt's Extended Attributes After Being Processed with PAMEx

From this example, PAMEx's security information is evident in `alpha_dev_instructions.txt`'s extended attributes. The first line depicts a definition of the file being accessed, and the second and third lines show the extended attributes written by PAMEx. The first of which are `alpha_dev_instructions.txt`'s PAMEx labels with the key of `security.fsc.labels` and the value of `alpha_dev_instructions.txt`'s list of labels assigned to it delimited by a colon. Similarly, the next line depicts `alpha_dev_instructions.txt`'s level information with the key of `security.fsc.level` and value of the level name and level placement delimited by a colon. The last line pertains to security information invoked by SELinux, and is thus orthogonal to PAMEx.

Through the process of invoking error handling, it was ensured that if an error is to occur while

invoking a function from the xattr library, the appropriate error label would be printed. This feature is especially useful when the privileged user is unaware or forgets that the FLUDB tool needs to run as root.

The user database is created by the FLUDB tool alongside the extended attribute assignments. While reading the policy-out file line by line, the FLUDB tool may come across two different user actions to perform: `USER_LEVEL` and `USER_LABELS` respectively. Much like the interpreted file lines, the user lines were also created to be standalone actions. This means that the user information can be written in any order anywhere in the file and FLUDB will still be able to correctly interpret it. In addition to the reasons stated above regarding the interpreted file lines, the dynamism of the FLUDB tool also allows for fewer bugs and greater flexibility.

Figure 11 depicts an example of an extended users database with our working example of the administrator Alice and developer Bob.

```
Alice:administrator:3:alpha:beta:charlie
Bob:developer:2:beta:charlie
```

Figure 11. The Contents of an Example Targeted Users Database Written by PAMEx

3.5. Design and Implementation of PAMEx's Pluggable Authentication Module (PAM)

The final tool that needed to be designed to implement the bare functionality of PAMEx was the PAM module. With the implementation of the FLUDB tool, PAMEx has a method of creating a targeted users database which holds all the system users' PAMEx security information and a way to write all the files' extended attributes which labels files with security information. The next step is the ability for the system to know who is logged in and for the system to have the necessary information on hand when checking the security clearance of a user. In its end state, PAMEx should be able to cross reference a user's security information with a file's extended attributes to determine if a user has the proper security clearance to access said file. For this to occur, the system must know who the logged in user is and therefore, store their PAMEx security information during a system user's authentication [3]. In our example, when the user Alice logs in to the system, the system will need to know what level of PAMEx security clearance Alice has access to and what security labels Alice has access to. Attention was turned once again to preexisting Linux Security Modules including SELinux [13]. It was determined that the best way to achieve the functionality needed was through the use of a PAM module.

PAM separates authentication and user session functionality into different modules. The modules are stacked on top of each other and performed as steps in an authentication process. The steps are sorted into two types: authentication modules and session modules. Authentication modules are performed during the authentication of a user, and session modules are performed after

the authentication of a user and during that user's session on the system. Therefore, when a user authenticates onto a system, a particular set of PAM modules are invoked depending on where and how the user authenticated [3]. While relatively simple from a high perspective, PAM modules are complicated to develop and can be dangerous to a system if invoked improperly. Many issues can occur if a user incorrectly invokes a PAM module, but most likely, the system's user login process will fail due to an error. The error often breaks the login system and further forbids a user from logging in. At this stage, development was continued on a virtual machine to ensure the safety of the system.

PAM C libraries were used to create the PAM module [9]. In the C library, PAM module actions are divided into categories such as authenticate, open session, close session, or manage account. For PAMEx, the only action that needs to be involved is the authenticate action. This is because PAMEx's PAM module creates and stores stagnant user information upon successful authentication of a system user which does not need to be modified during a user's session on the system. PAMEx's PAM module was developed as a shared object file type and placed in the `/lib64/security` directory much like the other PAM modules installed on the system so that it can be later accessed by PAM configuration files. The custom-developed module is invoked as the first action in the `system-auth` and `password-auth` PAM configuration files found in `/etc/pam.d`. Figure 12 depicts the first few PAM authentication processes of the `system-auth` file found in `/etc/pam.d` including PAMEx's PAM module `pam_pamex.so`. `system-auth` and `password-auth` are invoked every time a user graphically logs into the system, when a user uses the `su` command to log into a kernel on the system, and when `sudo` is used [3].

```
# Generated by authselect on Wed Jan 18 20:28:10 2023
# Do not modify this file manually, use authselect instead.
# Any user changes will be overwritten.
# You can stop authselect from managing your configuration by calling
# 'authselect opt-out'.
# See authselect(8) for more details.

auth    required                        pam_pamx.so /etc/userdb /tmp
auth    required                        pam_env.so
auth    required                        pam_faildelay.so delay=2000000
auth    sufficient                           pam_fprintd.so
auth    [default=1 ignore=ignore success=ok] pam_usertype.so isregular
auth    [default=1 ignore=ignore success=ok] pam_localuser.so
auth    sufficient                           pam_unix.so nullok
auth    [default=1 ignore=ignore success=ok] pam_usertype.so isregular
auth    sufficient                           pam_sss.so forward_pass
auth    required                             pam_deny.so
```

Figure 12. Code Snippet Pulled from System-Auth Depicting Implementation of PAMEx's PAM Module

Figure 13 depicts how PAM modules are invoked on a system with PAMEx via various authentication processes. When a system user authentication process occurs, the corresponding PAM

configuration files are invoked where various PAM modules are performed in succession.

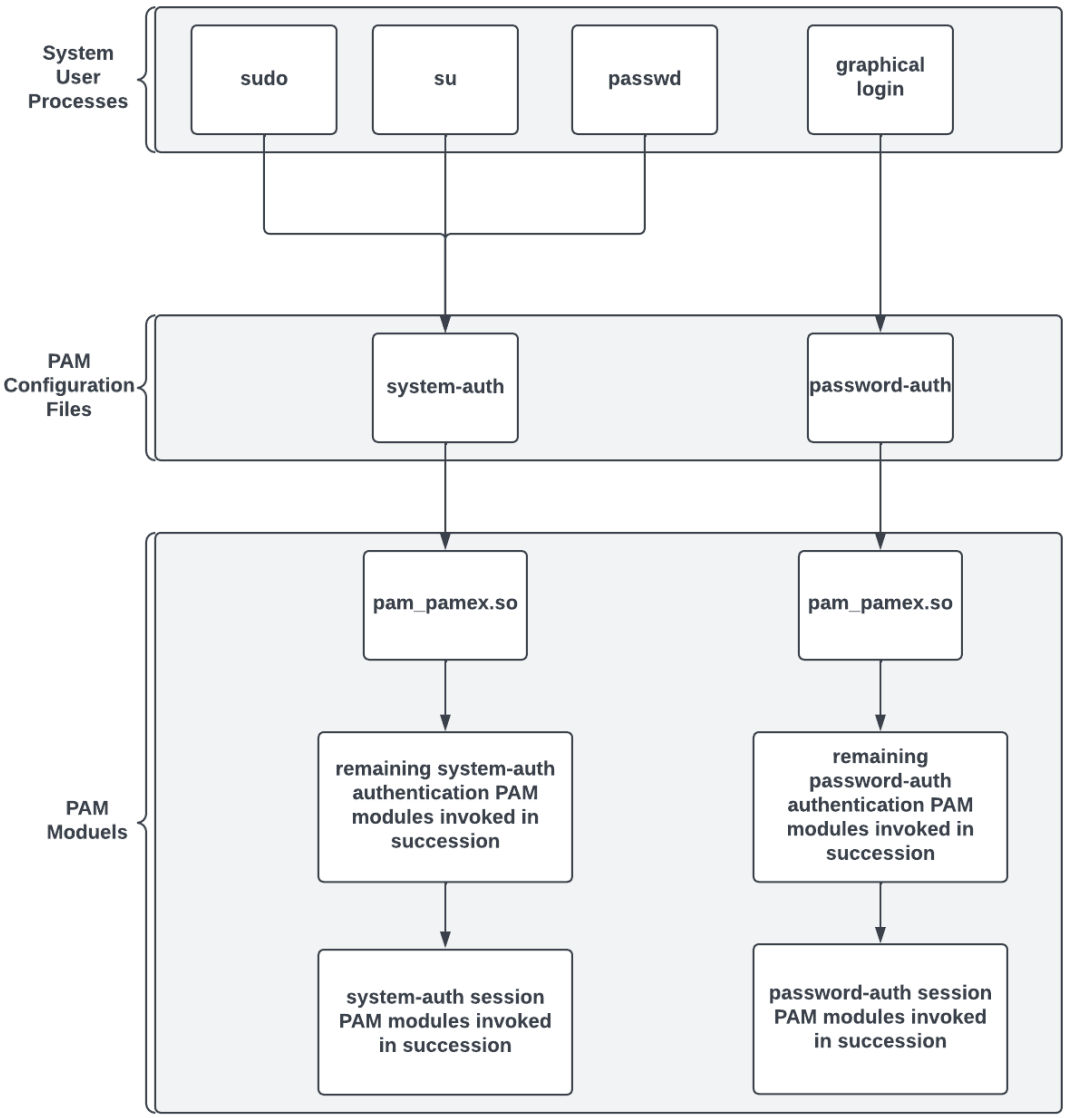

Figure 13. Overview of PAM Modules on PAMEx

Authenticating on a system using any of the above methods invokes PAMEx's custom PAM module and creates an information file that mimics the file which Linux Security Modules such as SELinux create upon user authentication. SELinux's informational file is named current and is located at `/proc/<pid>/attr` where `<pid>` is replaced with the process ID of the authentication process [13]. The process is forked as a child from the authentication process and any subsequent processes that the user invokes will be children to that process. The idea is that the signed-in user's PAMEx security information will be available through their session on the system.

A shortcoming of PAMEx is that its PAM module is unable to write to the system process file directory, `proc`. On most systems with this security method, the `proc` directory has write restrictions to disallow a user from being able to write ill-formed data, and the current kernel

requires an SELinux format. While lacking a custom kernel that would allow write access to process files, the PAMEx PAM module uses a work around by creating a fake, mimicked output process file which holds the logged in user's PAMEx security information that it would otherwise hold in the `proc` directory which the system calls pseudo-proc. The fake file can be outputted to any writable location. The following line depicts an example of what the pseudo-proc file contents might look like with our working example of the system user Alice.

```
Alice:top_secret:3:alpha:beta:charlie
```

3.6. Design and Implementation of the Extraneous File Labeler Tool

After creating the PAM module for PAMEx, it was quickly realized that there was an optimization issue that had a simple solution. The issue was that in the common scenario that a privileged, administrative user wanted to change the specification of the PAMEx security of a file after the security definitions had been assigned, the privileged user would need to re-write the PAMEx language file and run it through both the parser and the interpreter again. This two-step process could instead be handled in one after a security policy had previously been defined on a system and therefore, the Extraneous File Labeler tool was developed.

The File Labeler tool's job is to conform to the previously defined PAMEx security information but allow a privileged user to add, update, or remove a security level or label to a file. The File Labeler tool is a simple C script that takes in four arguments regarding how the privileged user is to change a file's security information. The first argument is the path to the level database that has already been created by the PAMEx parser. Any level that the privileged user redefines must come from the level database specified. The second argument is the path to the file whose security information is being changed. The third argument is a flag which specifies what process is to be done to the specified file including `-al -cl -dl` for add level, change level, and delete level respectively and `-ac -rc` for add label and remove label respectively. The last argument that the File Labeler takes in is a level or label name. In the case of removal, this name must match an already defined level or label in the file's extended attributes, but in the case of a change or addition, the name may be unique to the file so long that, in the case that a level is being added or changed, the level is contained in the level database. For example, if we wanted to change the level of `alpha_dev_instructions.txt` from `developer` to `general_staff`, a privileged user could use the File Labeler tool with the command below.

```
./file-labeler ../data/level-database
    ../files/alpha_dev_instructions.txt -cl general_staff
```

Figure 14 depicts `alpha_dev_instructions.txt`'s exteded attributes after the above command has modified `alpha_dev_instructions.txt`.

```
# file: alpha_dev_instructions.txt
security.fsc.labels="alpha"
security.fsc.level="general_staff:1"
security.selinux="unconfined_u:object_r:user_tmp_t:s0"
```

Figure 14. An Example of alpha_dev_instructions.txt's Extended Attributes After Being Modified by PAMEx's Extraneous File Labeler Tool

3.7. Design and Implementation of the Oracle Tool

A Linux kernel can enforce Linux security modules to affect how the kernel imposes its access controls. For a Linux security module, the kernel is written in such a way that allows for the module to perform its actions because of specific commands running [7]. PAMEx does not yet provide a Linux security module. It was decided to omit the development of a custom kernel because of the major challenge and it holds which is a challenge that is not needed to demonstrate the functionality of the PAMEx tool suite. The task of creating a custom kernel in addition to the PAMEx compiler and tool suite would have also greatly extended the PAMEx project's development time.

In lieu of the customized kernel, a tool called Oracle was created that allows a user to check if a logged-in system user has the PAMEx security clearance to access a particular file. Oracle is a user-prompt tool where the user submits queries about the PAMEx security of a system including the signed in user's PAMEx user security information, a file's PAMEx security information and whether the signed in user has the PAMEx security clearance needed to access a file. Oracle's primary function is to cross-check a file's extended attributes related to PAMEx with the process file that PAMEx's PAM module created containing the signed in user's information. With the use of this information, Oracle can inform the user if the signed in user has access to the specified file, and if not, why?

After a user authenticates on a system, PAMEx's PAM module creates a fake process file with user information as mentioned in Section 3.5. This is a replacement to how a custom kernel would propagate a process to be forked as a child from the authentication process. The way that Oracle works is it first prompts the user for the fake process ID that the PAMEx PAM module creates the directory for. The Oracle user inputs the fake process file ID and if it is found, Oracle then allows the user to input a command to find out more about the authenticated user or a file's PAMEx security information.

Oracle was by no means developed to be a replacement to a custom kernel. Instead, Oracle serves as an end-to-end simulation that brings together and demonstrates all the PAMEx tools and components. Oracle allows a PAMEx user to effectively demonstrate all PAMEx's functionality in a working environment. Figure 15 depicts Oracle working and demonstrating the full functionality of PAMEx.

```
Welcome to the PamEx Oracle!
Enter PID
    > 4571

Pid accepted.

Type help for a list of commands.
    > help

Pamx Oracle Commands:
help - print list of commands
user - check the name of the signed in user
userinfo - get the name and security information of signed in user
checkfileaccess - check if the signed in user can access a file
cfa - alias for the checkfileaccess command
fileinfo - get the authentication information of a file
quit - quit the Oracle

    > cfa

File path: ../tests/assignment_files/alpha_dev_instructions.txt

Access denied. User level too low.
```

Figure 15. A Working Example of the Oracle Tool Being Used

4. Testing

Test cases were created alongside the development of PAMEx to follow the Agile development model and Scrum framework. For each sprint, an iterative completed product was to be made and therefore, that iterative product was tested during each sprint and specifically how the new iteration worked with the project as a whole [4]. Unit tests were performed to ensure that for each function in each iteration, PAMEx performed how it was supposed to. These unit tests were performed manually by testing several input values for each parameter and by making sure to practice branch coverage by testing every conditional outcome. Boundary testing was also practiced during unit testing and therefore, edge cases were specifically paid attention to. It was ensured that PAMEx handled errors in the way that it was supposed to and even exited gracefully when need be [1].

One of the major decisions made while error handling was deciding for each input scenario that would result in an error, should the program exit gracefully or ignore the input and continue. In many cases when an error occurred in the program, the state was unsalvageable and therefore would be required to exit gracefully. This was especially true with the PAMEx compiler. However, in some cases, such as when the FLUDB reads in an assignment for an entity that does not exist on the system such as a missing file, the program would simply print out a statement saying that the action could not be performed, ignore the action, and move on.

Integration testing was also performed on each PAMEx program which focused more on the end-to-end process of the code rather than the individual parts [1]. For example, the PAMEx compiler focused on integration testing by creating 12 end-to-end test files. Each test file focuses on testing a particular aspect of the PAMEx language and each file builds from the last. The idea is that as a new aspect of the PAMEx language is introduced in a new test file, if the test fails, the developer will know which aspect of the language is causing the issue. Therefore, it would be easy to pinpoint how and where the error occurred. Finally, a C script was developed to test all the compiler test files in succession.

After performing unit tests in a similar manner to the PAMEx compiler, other tools had integration testing performed manually. The FLUDB tool was tested by creating several different policy-out files from the PAMEx compiler and inputting them into the tool. Once most bugs were found using unit and integration testing and the FLUDB tool was in a production-level state, the policy-out files were modified to a state that the FLUDB tool was not designed to interpret to test its error handling.

The PAMEx PAM module was trickier to test than the other tools because it is not a standalone C program. The PAM module is instead invoked whenever a user logs in to a system. As stated in Section 3.5 however, if the PAM module fails during the login process, it is likely that the entire authentication process fails whenever a user attempts to authenticate on the system in the future. The lack of being able to authenticate on a system is a major problem. Not only can no user log into the system anymore, but users also cannot authenticate when trying to perform a process as root to fix the authentication failure. Several tools were used to combat this issue. First, the PAM C libraries `security/pam_appl.h` and `security/pam_misc.h` were used to create a standalone C program that performed similar actions to a PAM module. The C program acted like a PAM simulation with built in functions like `pam_start` to begin the PAM process, `pam_authenticate` to simulate a user being authenticated, and `pam_end` to end the process [9]. Making use of these PAM libraries gave insight as to how the real PAM module should be

written. With this insight, and by perfecting the simulation, fewer bugs were likely to be created when developing the real PAM module.

Whenever testing or demonstrating the use of the PAM module, a virtual machine was used. The virtual machine has built-in functionality to snapshot a state of the machine and then reload the state later if necessary. Making use of this feature, a snapshot of the virtual machine was created every time a change was made to the PAM module but before the PAM module was fully implemented onto the virtual machine. This ensured that, should the PAM module break the virtual machine, there would be a safe state to load back to.

During the testing of the PAM module, Fedora's pamtester package was also utilized. The pamtester process allows a user to specify a particular grouping of PAM modules that they wish to test with the following command.

```
pamtester <service-name> <username> <module-name> [<flags>]
```

In the case of the PAMEx PAM module, the pamtester command might look like the following line.

```
pamtester system-auth Alice authenticate
```

The test command above reads: invoke the pamtester on the `system-auth` file in the directory `/etc/pam.d` which contains a list of PAM modules to invoke in succession. Invoke the modules with the user Alice, but only run the PAM modules with the authentication label. The pamtester would then simulate all the authentication PAM modules mentioned in the `system-auth` file running in succession with the system user Alice being authenticated. Using the pamtester in this way directly tested the PAMEx PAM module without performing a real authentication process on the system and therefore, should an error occur, the authentication system would not break [9]. The pamtester was used extensively for integration testing. Once satisfied that the PAM module was working as expected, it was then tested by performing actual user authentication on the system including the `su` command, using the `sudo` command to authenticate as root, and graphically logging in to the system. The PAM module requires two parameters which specify the location of the user database file and the output of the PAM module and were therefore tested with several correct and incorrect values as well as no values at all to ensure that errors were handled appropriately.

Oracle was created as a user prompt tool and therefore extensive integration and boundary testing was done to ensure that all user input was handled properly. For instance, the first value that Oracle prompts for is the pseudo-proc ID. If a user inputs any ID that is not found in the system, Oracle says that the ID cannot be found and asks the user to try again. A similar process is done when Oracle prompts the user for a command and when it prompts the user for a file. For each prompt, several integration and boundary tests were manually performed until Oracle was working as expected and was in a production-level state.

5. Future Work

PAMEx performs its designed job of being a custom-built security policy compiler which allows for more dynamic security policies than the default, binary policy that SimpleFlow currently enforces. PAMEx gives the privileged user the opportunity to create full hierarchical security policies on a system complete with both levels and labels. Although PAMEx is presented from end-to-end as a working security tool suite, the lack of a custom-built kernel prevents it from behaving as a fully-fledged security module. Therefore, the relationship between PAMEx and SimpleFlow as it stands is as follows: SimpleFlow is a customized kernel and network filter, tailored to monitor the actions of malicious users with a binary security policy of marking files as either confidential or non-confidential. PAMEx is a linux security tool suite that allows a privileged user to define a more flexible security policy. However, PAMEx does not have a kernel that is customized for it and therefore cannot be automatically enforced on a system as it stands. For PAMEx to work with SimpleFlow, SimpleFlow's custom kernel will need to be tailored to allow PAMEx security policies to be enforced. In doing so, a privileged administrative user will be able to further define through PAMEx how users on a SimpleFlow system will be marked as performing illicit activity and allow SimpleFlow to monitor their actions and filter their exfiltration attempts. Using the Oracle tool to simulate the security policies created by PAMEx is an excellent way to show PAMEx working from end to end but does not allow PAMEx to be used in a fully-fledged production setting like SimpleFlow for which it was designed.

The file labeler tool was developed to provide the ability to quickly change, set, or remove the PAMEx security information from a file. In future work, a tool could be developed to perform similar functionality with user security information. With the current iteration of PAMEx, if a privileged, administrative user wants to add, remove, or in some way modify a user's PAMEx security information, the administrator would need to start from the compilation process and work through each step of the PAMEx system. The development of a new user labeler tool would allow the administrator to skip some of the steps of the process should they find the need to modify the security information of system users.

The need for PAMEx to be incorporated into and working alongside SimpleFlow still stands. Porting PAMEx into SimpleFlow is a significant undertaking and one that would need to be done only after a custom kernel has been tailored for PAMEx.

6. Conclusion

The PAMEx security compiler and tool suite, designed to enhance the SimpleFlow security policy from being a binary system to being a flexible security system, has achieved an end-to-end proof of concept. Being inspired by the US security classification system [10], PAMEx can define both hierarchical levels and non-hierarchical labels while having a small footprint on a system of less than 1,700 lines of code. Although PAMEx was tailored to be integrated into SimpleFlow, it was ultimately not able to be ported into the SimpleFlow project. SimpleFlow is a custom kernel that is not tailored for PAMEx's integration yet. PAMEx on the otherhand requires custom kernel modifications in order to enforce its security policies properly. Therefore, the next step for the PAMEx project is to customize the SimpleFlow kernel to allow the integration of PAMEx.

The system was developed and integrated as several lightweight tools. Therefore, the porting of PAMEx onto SimpleFlow or any other system should be straightforward because if any of the PAMEx tools fail, it should be simple to pinpoint the issue. Likewise, any future maintenance that needs to be performed on PAMEx can be isolated to the tool that needs to be updated or maintained and therefore, should also be simple. Several end-to-end demonstrations with the aid of PAMEx's Oracle tool have proven that PAMEx works as a flexible, easily definable Linux security tool suite.

The development of PAMEx proved to be an invaluable experience for not only the further enhancement of SimpleFlow but also to learn from. During the development of PAMEx, a great deal was learned about the Linux security system and specifically how Linux security modules are invoked and function. A lot was also learned about the development of software products in general including how to effectively use the Agile process and Scrum framework. PAMEx uses the end-to-end simulation of the Oracle tool to prove the value it has for SimpleFlow and future projects.

Bibliography

[1] Jorgensen, Paul C. *Software Testing: A Craftsmans Approach*. Auerbach Publications, 2021.

[2] Kung, David C. *Object-Oriented Software Engineering: An Agile Unified Methodology*. McGraw-Hill, 2014.

[3] Pluggable Authentication Modules (PAM), https://www.redhat.com/sysadmin/pluggable-authentication-modules-pam, last accessed 15 April 2023.

[4] Lacey, Mitch. *The Scrum Field Guide: Practical Advice For Your First Year*. Addison-Wesley, 2015.

[5] Levine, John. *Flex & Bison*. O'Reilly, 2009.

[6] Levine, John R., et al. *Lex & Yacc*. O'Reilly, 1995.

[7] The Linux Kernel Documentation, https://www.kernel.org/doc/html/latest/, last accessed 15 April 2023.

[8] Linux-Pam/Linux-Pam: Linux PAM (Pluggable Authentication Modules for Linux) Project, https://github.com/linux-pam/linux-pam, last accessed 15 April 2023.

[9] Linux-PAM, "pam - Pluggable Authentication Modules for Linux", Linux-PAM Manual, Release 1.3.1, 2020.

[10] Senate Select Committee on Intelligence. "National Security Information" Senate Select Committee on Intelligence, https://www.intelligence.senate.gov/laws/national-security-information

[11] Ryan Johnson, Jessie Lass, W. Michael Petullo. "Studying Naive Users and the Insider Threat with SimpleFlow" Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats, 2016.

[12] What is Scrum?, https://www.scrum.org/learning-series/what-is-scrum, last accessed 15 April 2023.

[13] SELinux Project, https://selinuxproject.org/, last accessed 15 April 2023.

[14] T. Ts'o, "xattr - Extended attributes," Linux Programmer's Manual, xattr(7), 2021.