Automatic Policy Generation for BPFContain

Carleton University - COMP 4905 - Final Report Draft

Jake Jazokas – 101083496

Supervised by: Anil Somayaji

April 11th, 2022

Abstract

Given the relative difficulty of generating security policies for BPFContain. This project aims to automatically generate these policies using a tracing program to log events occurring across a system. The generated log file is then used by a translation program to extract relevant information and transform the data into a valid security policy for BPFContain. The combination of these two programs results in an efficient system for the automatic generation of security policies for a given program.

Acknowledgments

I would like to thank Professor Anil Somayaji for assisting me with the initial idea for the project. I would also like to credit William Findlay for the implementation of BPFContain (http://people.scs.carleton.ca/~soma/pubs/students/william-findlay-mcs.pdf), which was used throughout this project to enforce the generated security policies.

Table of Contents

List of Figures/List of Tables

1. Introduction

The main objective of this honors project is to research and implement an application that will automatically generate security policies for the container security daemon BPFContain, created by William Findlay [1]. The purpose of these security policies is to limit the behaviour and interactions that a process can make within a containerized system. The focus of these policies will be to provide protection against unusual program behaviour. eBPF is used to give the application access to the calls being made in kernel space. This will allow us to view what applications are doing and will give our application the information it needs to automatically generate security policies. Once the security policies are generated by the implementation, they will be sent to BPFContain, which will then execute and enforce them.

1.1 Problem

Although BPFContain is a powerful tool, the process of manually creating in-depth security policies is a time-consuming task and can be prone to human error. That is why this project aims to automatically generate these policies, as it will save time and eliminate the chance of human error.

1.2 Goals

The goal of this project is to create a new application that leverages the use of both BPFContain and eBPF to automatically generate security policies that will protect against abnormal program behaviour. This application will contain two parts, a tracing program, and a translation program. The goal of the tracing program is to capture and log events, which are generated by the traced program whenever it interacts with the filesystem, devices, or other processes. The goal of the translation program is to interpret the events logged by the tracing program and transform them into YAML security policies that can be parsed and enforced by BPFContain. The tracing program is implemented using bpftrace, which is a high-level tracing language for eBPF on Linux, and the translation program is implemented as a Python script.

1.3 Motivation

The motivation behind this project originated from my passion for both automation and system security. Wanting to further my knowledge and experience in these fields, my supervisor helped shape the idea for this project. This project aims to contribute to BPFContain, by simplifying the generation of security policies.

1.4 Objectives

The initial objective of this project was to implement support for the generation of security policies that outline both allowed and restricted interactions between a process and files within a filesystem. However, BPFContain can enforce more than just filesystem rules, so the next objective was to implement support for the other areas of enforcement. These additional areas of enforcement include sending signals, inter process communication, and process capabilities.

2. Background

This section will provide all background knowledge needed to understand both the approach of the implementation and the results. It also aims to provide a summary of existing works that are related to this project.

2.1 Background Information

The goal of this section is to provide an understanding of the concepts that comprise process security on Linux based operating systems (*OS*). It will also explain the purpose and function of eBPF, in relation to this project.

2.1.1 The Linux Kernel and eBPF

The Linux kernel is the main component of any Linux based operating system. Its purpose is to function as a low-level interface between the physical hardware of a device and the software running on it [2]. The main functions of the Linux kernel are to provide memory and process management, access to hardware devices, system security, and system calls [2]. System calls are used by processes to request the kernel to perform some function.

The Extended Berkeley Packet Filter (*eBPF*) allows for the extension of kernel capabilities to privileged user space processes [3]. eBPF provides to ability to dynamically monitor a system by hooking onto specific kernel or userspace functions. These hooks are executed alongside the original functions and provide the ability to trace the execution, inputs, and outputs of the hooked function. eBPF is used in this project as it provides the ability to trace events generated by a specific process, across the entire system.

2.1.2 Virtual Filesystem

The virtual filesystem (*VFS*) is a software layer in the Linux kernel that provides an interface for userspace programs to interact with the underlying filesystem abstraction [4]. VFS functions within the kernel trigger when system calls pertaining to a filesystem are executed. Kernel functions are also able to call VFS functions directly, this aids in the segregation of kernel and user level events, as the kernel does not need to execute system calls to interact with a filesystem. System calls are designed for execution at the user level; thus, they should not be executed by any kernel level functions. VFS functions are hooked by this project to provide insight into operations occurring within the filesystem.

2.1.3 bpftrace

The bpftrace tool provides a high-level language, that interfaces with eBPF to provide tracing capabilities within a Linux system [5]. This tool is used extensively within the tracing component of this project, as it provides the capability to easily create hooks for a variety of dynamic kernel and user functions.

2.1.4 Kernel Probes, User Probes and Tracepoints

Kernel probes (*kprobes*) are used to trace any function calls within the kernel [6]. The eBPF program interacting with kernel probes, only has access to read the traced information, and cannot modify it. There also exists another kind of kernel probe, `kretprobe`, which allows for tracing of function returns instead of calls. Whereas user level probes such as `uprobes` and `uretprobes` give the same functionality as the kernel variant, except the events and tracing occur at the user level [7]. Tracepoints can be used to hook specific functions. Each time the function specified in the tracepoint is executed, a function associated with the traced function is also executed [8]. Tracepoints are a functionality of the Linux kernel, but they can be used to trace the execution of user level functionality such as system calls. This project uses a variety of probes and tracepoints to gather information about the execution of a specific program.

2.1.5 BPFContain

BPFContain [1] is the base application that will be used to run the security policies generated by this project. BPFContain [1] reads and executes YAML formatted security policy files, which define the resources that a program is allowed or restricted from accessing. BPFContain is used to confine process execution, so that a process only has a minimal set of privileges that are required for it to operate [1]. Confinement is essential for the security principle of least-privilege, as processes should not have privileges that they don't immediately need.

2.2 Related Work

This section will cover other tools related to Linux security policies and tracing. Tools such as Dtrace and SystemTap served as inspiration to bpftrace, and thus, most of their functionalities have been also included in bpftrace.

2.2.1 Audit2Allow

Audit2Allow is a Linux tool that generates SELinux policy based on rules, which are inferred from logs of denied operations for a given program [9]. SELinux is a security architecture that supports the implementation of access control security policies, this functionality is like that of BPFContain, but with a more limited scope. The Audit2Allow tool inspired improvements to the tracing program in this project, as originally all traced operations were assumed to have been successful. By looking at operations that were denied, the tracing program can generate specific restrictions for the program being traced. These restrictions explicitly outline areas of a system that a program should not be able to interact with.

2.2.2 IPCDump

IPCDump [10] is a Linux tool used to trace inter process communication (*IPC*). It relies on BPF to probe specific areas of the kernel using kprobes and tracepoints [10]. To interact with BPF, IPCDump uses the go library gobpf [11]. After using this tool and going through the source code, I was able to determine which probes were used to trace specific events within the kernel. This provided me with insight into which probes to use in the tracing program, to capture events related to inter process communication.

2.2.3 Dtrace

Dtrace is an analysis tool that allows for kernel and user level tracing. Dtrace was the predecessor to eBPF [3] and bpftrace [5] and provided kernel level tracing before eBPF was implemented in the Linux kernel [12].

2.2.4 SystemTap

SystemTap [13] is like bpftrace in that it used kernel and user level probes to trace system events. However, SystemTap [13] instructions compile into C code. This compiled code is then loaded as a module in the Linux kernel. This adds additional security risks as it is not verified that the program can be trusted and will not crash the kernel. Whereas bpftrace [5] scripts compile into BPF code, which ensures that the program is safe to run in the kernel [14].

3. Approach

This section will cover the design of the structures within the tracing and translation programs that make up this project. It will also cover the attempts made along the way in creating the final implementation, as well as a breakdown of the final implementation itself.

3.1 Design

There are many components that make up the overall design of this project. However, they can be broken down and visualized as elements of the tracing and translation programs. The general design of the overarching project has information flow from the tracing program to the translation program through a log file containing the traced events. The translation program then uses this data to create a BPFContain security policy.

3.1.1 Tracing Program

The design of the underlying structure of probes and tracepoints used in the tracing program increased in complexity throughout the course of the project as new features were added. The current design of the tracing program is shown below in Fig.1 and Fig.2. In these diagrams, blue nodes represent the high-level concept of what the structure is attempting to trace, orange nodes represent the type of operation captured by the probes, yellow nodes represent the probed kernel functions, and green nodes represent tracepoints. The flow of data follows the direction of the arrows and begins when the probes or tracepoints are executed and return a value. These return values are relevant as they determine if the operation was allowed or denied. This data is then interpreted as a specific operation and printed to a log file.

Fig.1 displays the structure of three distinct traced features, signals, capability checks, and inter process communication. These features can be captured using a small number of probes and tracepoints, compared to operations on filesystems.

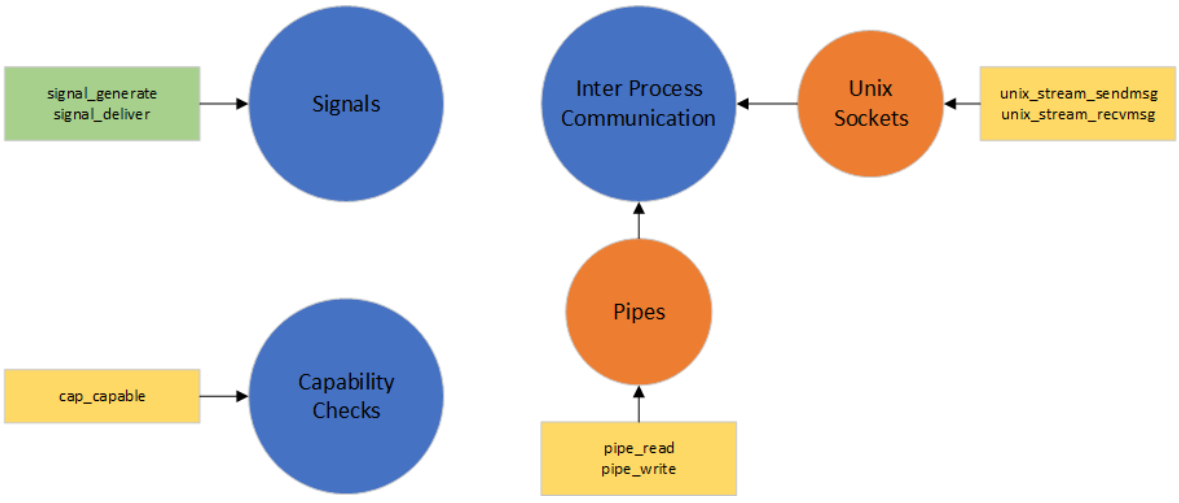

Figure 1: Design of signal, capability, and IPC tracing

Fig.2 displays the complex structure surrounding the implementation of the ability to trace operations within a filesystem. Some probes such as `vfs_rmdir` require multiple permissions to function properly, in this case these permissions are write and execute. The probe for `vfs_open` can produce a permission for each file operation depending on the flags it has set. Thus, extra processing must be included within the hook for `vfs_open` that determines the permissions needed for any given flag. Operations on devices are also traced by the structure in Fig.2, these operations are differentiated from file operations in the log, as they include an extra string detailing the device information.

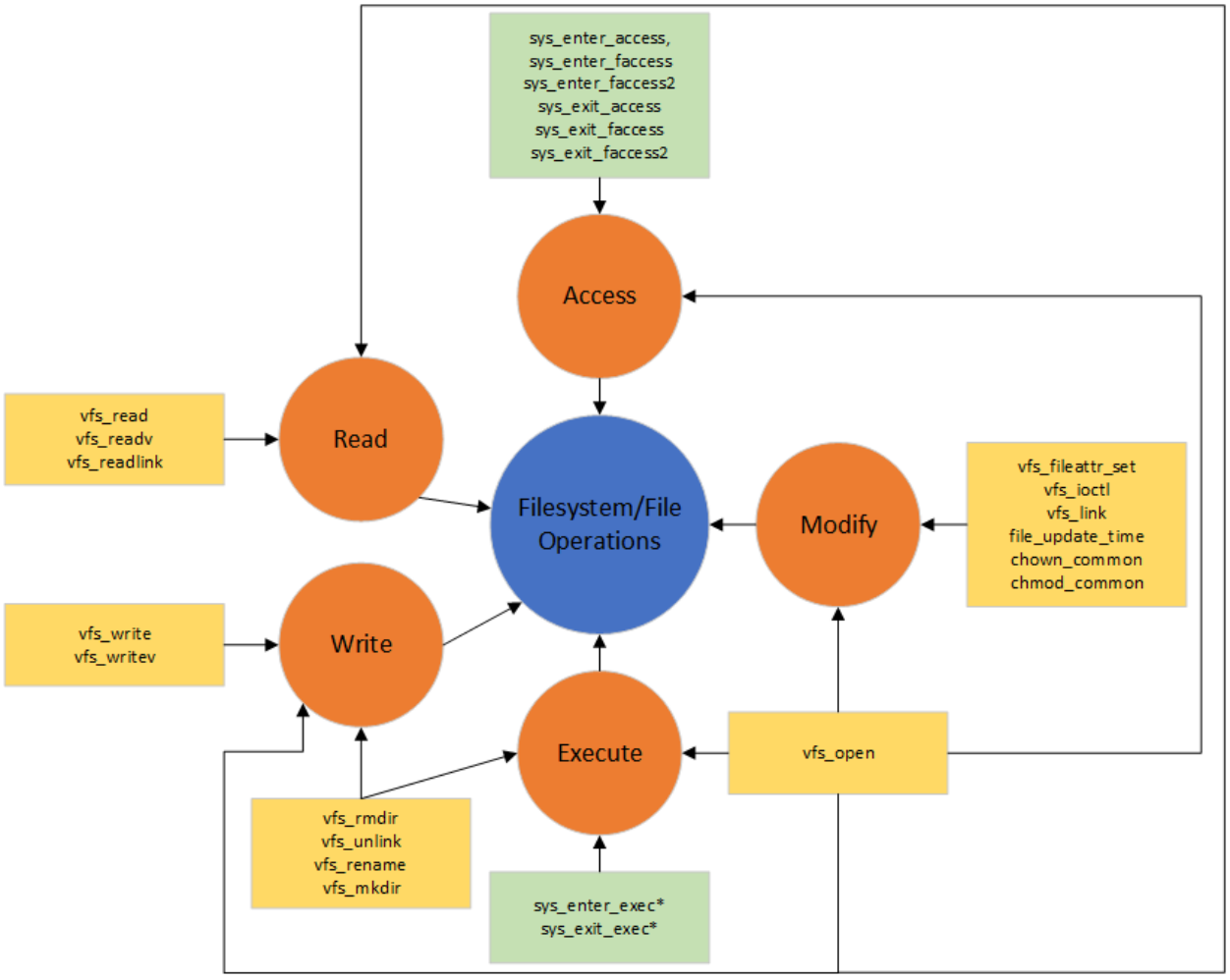

Figure 2: Design of filesystem, file, and device operations

3.1.2 Translation Program

The translation program is made up of two classes, `TraceFile()` and `TraceToPolicy()`. The `TraceFile()` class is used to read the file generated by the tracing program into several arrays, where each array is associated with a different type of trace, ranging from `VFS` operations to Unix Sockets. The `TraceToPolicy()` class takes in a `TraceFile()` object and uses it in a series of functions that gather information and produce output related to a specific part of a BPFContain policy. At the end of the `TraceToPolicy()` class, the policy fragments are combined to create a full, YAML formatted, security policy. This policy is written to a file, which can then be used by BPFContain.

The design of the translation program can be seen below in Fig.3. In the diagram, blue nodes represent a Python class, yellow nodes represent the input, orange nodes are objects that the class has access to, and green nodes represent an output. Orange nodes are needed as there is no output for the `TraceFile()` class, it is only used as an object which is passed to the `TraceToPolicy()` class.

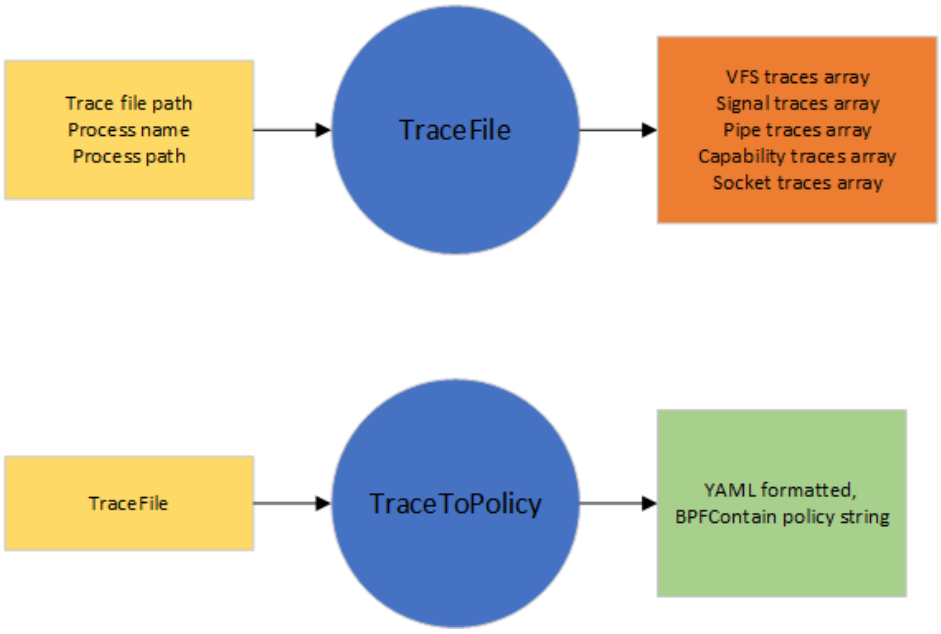

Figure 3: Design of the translation program

3.2 Attempts

Both the tracing and translation programs have gone through many changes, over several iterations. This section attempts to highlight why each change was made and how the performance of the project improved after each change.

3.2.1 BCC

My first attempt at tracing the system was using the BCC framework [15]. This framework provides a python front end that interacts with BPF and provides the ability to do kernel level tracing. To use BCC [15], I had to build it from source, as the version available on the ubuntu package manager is outdated. After dealing with numerous issues from dependencies when building BCC [15], I decided to switch to using another tool, bpftrace [5], which could be easily installed via. The package manager DNF [16] on Fedora v35.

3.2.2 Kprobes to Kretfuncs

Initially, all tracing was done using `kprobes`. This provided the ability to gather information about the input parameters as the probes were activated. However, this lacked the ability to trace the return values of the hooked functions. Changing the `kprobes` to `kretfuncs` gave the tracing program access to the return values of the hooked functions. These return values are used to decide if a traced operation should be allowed or denied.

3.2.3 Read, Write, Modify, Execute, Access

In the first iteration of this project there were only checks for read and write permissions on files and filesystems. This evolved to include the modify, execute, and access permissions as these access flags are supported by BPFContain, and thus had to be tracked.

3.2.4 Specific to General probes

The initial iterations of the tracing program traced `kill()` system calls using `sys_enter_kill`, to trace signals that were sent by the traced process. This was problematic as signals can be generated, sent, and received by methods other than the `kill()` system call. To capture signals generated by any means, the `signal_generate` and `signal_deliver` tracepoints were used. These allowed for a broader range of signals to be captured as these tracepoints are executed each time any signal is generated or delivered to a process.

Later iterations of the tracing program probed the `vfs_fchmod` and `vfs_fchown` kernel functions to gather information about file ownership or permission changes. This proved to not be effective as there were several separate ways to change file ownership or permissions that did not cause the `vfs_fchmod` and `vfs_fchown` kernel functions to activate. However, I noticed in the kernel source code [17] that these `VFS` functions, along with other file ownership and permission change functions, were all just calling a common function. These common functions are `chown_common` and `chmod_common`, and probing them allows for every file ownership or permission change to be traced, as they are the underlying functions called by most ownership and permission change functions

3.3 Implementation

This section will outline how the current implementation of this project functions. It will provide insight into how solutions were researched and implemented.

3.3.1 Tracing Program

I used bpftrace to implement the tracing program [5]. bpftrace provides a high-level language that can be used to interact with the BPF system [5]. Although a current version of bpftrace can be easily installed on Fedora, the kernel must be built with specific flags that allow

access to BPF, kprobes, uprobes, and tracing of system calls, tracepoints and kernel functions [18].

Using bpftrace [5], I attempted to trace filesystem operations by hooking the relevant kernel functions. These function hooks get the inputs and return values from read, write, execute, access, and modify operations within the virtual filesystem (*VFS*). The design of the structure used to contain all the different hooks can be seen in Fig.2. These hooks are activated any time the traced program attempts to perform any file operation on any file or filesystem, which includes special files, devices, and directories. The variation in input parameters between functions can be seen below in the function definitions from the Linux header file `fs.h`.

```
extern int vfs_readlink(struct dentry *, char __user *, int);
extern ssize_t vfs_read(struct file *, char __user *, size_t, loff_t *);
```

Knowing the input parameters to each of these functions allowed me to recognize that the file object being referenced is not always a file struct and could be another structure such as a dentry. However, no matter the structure used to contain information about a file, we can still access the underlying information, using a variety of pointers to traverse each structure. For example, the first argument of `vfs_read`, the file pointer struct, could be used to get information on the file that is being accessed, by extracting the name of the file by traversing the file pointer. First, we get the `path` struct `f_path` contained within the file pointer. This `path` struct then gives us access to the `dentry` pointer, which is a specific part of the overall path to the given file. Finally, we can access the `qstr` struct `d_name` to get the name of this specific path component. Since a dentry struct is an underlying object within a file struct, it can be used in a similar fashion to extract the desired file information.

The next challenge was to extract the full path of any file that was interacted with by the tracing program. This is done similarly to how the file name is resolved, except in this case we

want to iterate through the dentry pointer d_parent, until we hit the root directory. Each of these d_parent objects contain the name of the parent directory for a given dentry. Since unbounded loops are not allowed to be run in the kernel, we assume a max depth of ten folders. This limit has not been reached in testing thus far but could be increased if needed in the future. A possible solution to this would be to provide an extra input parameter to the tracing program that would set the max nested depth across all hooks. A code snippet is shown below, detailing how a dentry object is traversed to find the full path.

```
$parentDir = ((struct file *)arg0)->f_path.dentry->d_parent;
@fullPath[0] = str($parentDir->d_name.name);
...
while($count < 10){
$parentDir = $parentDir->d_parent;
```

All parent directories are stored within the fullPath array, which we then iterate through, to log the full path to the file. Filesystem access is denoted with VFS: at the beginning of the log line, the rest of the line contains information on the operation performed on the file, the process ID, and the return value. If the file receiving the operation is a device file, an extra string containing the device information is appended to the end of the log line. An example of this output can be seen below, VFS functions with a return value less than 0 are regarded as denied operations.

```
VFS: Path: /usr/bin/sed, Program: bash, Tid: 12314, Mode: Access, Return: 0
VFS: Path: /, Program: bash, Tid: 12354, Mode: Write, Return: -32
```

After all the filesystem operations could be traced, I moved on to trace signal events. This is done using the two tracepoints, signal_generate and signal_deliver. These are activated whenever a signal is generated by or delivered to a process. The signal_generate tracepoint stores the process being sent the signal as well as the signal type and return value. When the signal_deliver tracepoint is activated, the same information can be retrieved, however, the combination of both events can be used to compare if the generated signal was created successfully but not delivered, resulting in a failed operation. We are then able to log

the information gathered, such as the process that sent and received the signal, the signal number, and the final return value which denotes the success of the whole operation. Lines in the log containing traced signal events begin with the string `Signal:` and can be seen in the snippet below.

```
Signal: From: bash, To: bash, SigNum: 17, Return: 0
Signal: From: bash, To: strace, SigNum: 17, Return: 1
```

After signals were fully implemented, the next events that I attempted to trace were operations on pipes. This is the starting step for generating rules for inter process communication (*IPC*). The implementation took inspiration from IPCDump [10] and uses the `pipe_read` and `pipe_write` function hooks to log information. The function definitions for these two operations, found in the Linux source file `pipe.c` are as follows:

```
static ssize_t pipe_read(struct kiocb *iocb, struct iov_iter *to);
static ssize_t pipe_write(struct kiocb *iocb, struct iov_iter *from);
```

Information is extracted, whenever these function hooks are activated, from the first argument, the `kiocb` struct pointer, which contains a `file` pointer struct that can be traversed to get the underlying path using the same method outlined for `VFS` functions. Pipes are denoted with `Pipe:` at the beginning of the log line, which also contains information on the process name that is reading to or writing from the pipe, the full path to the pipe, and the return value which can be used to determine if the operation was allowed or denied. An example of this output can be seen below.

```
Pipe: Path: /home/jakejazokas/Desktop/namedPipe, Program: bash, Mode: Write, Return: 8
```

After pipes were fully implemented, I proceeded on to the next category of events relevant for creating IPC rules, Unix Sockets. This is done using the two kernel function hooks, `unix_stream_sendmsg` and `unix_stream_recvmsg`. Both functions store their relevant

information with the msg argument, this can be traversed using pointers to get from the `iocb` struct to the `dentry` struct, which we can then use to extract what process the socket stream is being sent to or received from. Sockets are denoted with `Socket:` at the beginning of the log line, which also contains information on the process receiving the signal, the process sending the signal, and the return value to determine if it was a valid operation.

Finally, the last events I attempted to trace were capabilities. Capabilities are how Linux distributes specific privileges to processes. I took inspiration from the bpftrace script `capable.bt` [19], which uses a `kprobe` on the `cap_capable` kernel function that activates whenever a security check is made within the kernel. This was modified to instead hook the kernel function using a `kretprobe` instead of using a `kprobe`, as it allows for the return value to be verified. The function definition, found in the Linux header file `security.h`, is as follows:

```
extern int cap_capable(const struct cred *cred, struct user_namespace *ns,
                       int cap, unsigned int opts);
```

The numerical representation of the capability can be extracted using the data passed to the int `cap` argument. This numerical argument is transformed into the string representation of the given capability. If the unsigned int opts has a value of 0, we know that the given process has the given capability, this can be additionally checked by verifying the return value of the function is not less than 0. Capabilities are denoted with `CAP:` at the beginning of the log line and contains information on the program whose capabilities are being checked, the capability being checked, the audit which is the numerical representation of the permitted set of operations, and the return value representing if the capability check passed or failed. An example of this log output can be seen below.

```
Capability: Program: bash, CapInt: 2, CapName: CAP_DAC_OVERRIDE, Options: 0, Return: -1
Capability: Program: bash, CapInt: 1, CapName: CAP_DAC_READ_SEARCH, Options: 0, Return: -1
```

The final implementation of the tracing program can trace every event mentioned in this section. The tracing program is written in the bpftrace language and is executed using the bpftrace command. The tracing program takes in an optional argument that allows for the tracing of a program with a specific name. This is done by limiting each probe to only log data when the executing process matches the input argument. In bpftrace this is done by adding the following filter to each probe:

```
/comm == str($1)/
```

Where `str($1)` is the name of the desired program to trace. The variable `$1` represents the first argument passed into the bpftrace script through the command line. Each probe or tracepoint distinguishes itself in the log by specifying the type of the trace at the beginning of each line. Currently, the trace program can log all events except for events related to networking. If there is time remaining in the semester, I would like to add support for tracing and translating networking events.

3.3.2 Translation Program

The current implementation of the translation program is written in Python. It uses a variety of regex strings to match and extract different information within the log, this information is then translated into a YAML security policy for BPFContain. The program currently supports generating allow and deny rules for all events mentioned in section 3.3.1. The first class in the translation program is the `TraceFile()` class. The `TraceFile()` class is used to open the trace log and split it into sections based on the type of activity. Currently, it splits the data into VFS traces, Signal traces, Capability traces, Pipe traces, and Unix Socket traces.

The `TraceFile()` object is then passed into the `TraceToPolicy()` class, which generates the output YAML file through several functions. The first function being, `generate_policy_start()`. This function generates the start of the output string that will be written to the YAML file. It includes the process name and the command path used to execute the process. Currently, this information is set in the `TraceFile()` object so it can be retrieved from the `TraceToPolicy()` class .

Next, the `TraceToPolicy()` class calls the function `generate_policy_allow()`, which calls all the functions associated with creating `allow` rules for the policy. The first of these functions is `generate_device_access()`. This function reads through the array of all VFS traces and grants access to null and/or terminal devices, based on the paths that the process interacted with while it was being traced by ensuring that the return value is nonnegative. The next function called by `generate_policy_allow()` is `generate_read_write_execute_modify_access()`. This function uses the regex strings below, to match and separate filesystem operations.

```
# Regex Strings - Group 0 is the path, Group 1 is the return
read_regex = r"(?:Path: )([^,]*)(?:, Program: "+traceFile.procName+r", Tid:
[\d]*, Mode: Read, Return: )([-0-9]+)"
write_regex = r"(?:Path: )([^,]*)(?:, Program: "+traceFile.procName+r", Tid:
[\d]*, Mode: Write, Return: )([-0-9]+)"
access_regex = r"(?:Path: )([^,]*)(?:, Program: "+traceFile.procName+r", Tid:
[\d]*, Mode: Access, Return: )([-0-9]+)"
execute_regex = r"(?:Path: )([^,]*)(?:, Program: "+traceFile.procName+r",
Tid: [\d]*, Mode: Execute, Return: )([-0-9]+)"
modify_regex =  r"(?:Path: )([^,]*)(?:, Program: "+traceFile.procName+r",
Tid: [\d]*, Mode: Modify, Return: )([-0-9]+)"
```

These separated event arrays are then iterated through, and each path is added as a key to a dictionary. This dictionary stores the key value pair `path->[accessArray]`, where the value for each key is an array of access permissions (*read, write, etc.*). We then iterate through this dictionary to create file and filesystem `allow` rules for the traced program, by checking if the return value is greater than or equal to 0. If the `allowBool` flag is set to false, we can instead use this function to generate restrictions, as it will check if the return value is instead

less than 0. Currently, this function can create rules for any file within the main filesystem. In BPFContain security policies, filesystems are denoted by `fs` and files are denoted as `file`.

The next function called by `generate_policy_allow()` is `generate_signals()`. This function uses the regex strings below, to match and separate all the information from the array of traced signal events.

```
pattern_string = r"(?:From: )([^,]*)(?:, To: )([^,]*)(?:, SigNum:
)([^,]*)(?:, Return: )([-0-9]+)"
```

Then, we iterate through the regex matches and check if our traced program sends any signals to itself or other programs. This is done by comparing if the traced program is the same as the program found in capture group 0, ensuring that the traced program sent the signal. Capture group 3 is used to check if the probed return value is greater than or equal to 0, which indicates that the signal event was successful. When we find a valid signal we add the process name as a key, to a dictionary. This dictionary stores the key value pair `processName->[signals]`, where the `processName` is the process receiving the signal and the value for each key is an array of signals that can be sent. We then iterate through this dictionary to create signal `allow` rules for signals that the traced program can send.

The next function called by `generate_policy_allow()` is `generate_capabilities()`. This function uses the regex strings below, to match and separate all the information from the array of traced capability check events.

```
pattern_string = r"(?:Capability: Program: "+procName+r", CapInt:
)([^,]*)(?:.*Return: )([-0-9]+)"
```

Then, we iterate through the regex matches and check if the return found in capture group 1 is greater than or equal to 0. This indicates the capability check was successful and we

add the process name as a key, to a dictionary. This dictionary stores the key value pair `processName->[capabilities]`, where `processName` represents a process, whose capabilities are being checked. The value for each key is an array of capabilities that a given process has. We then iterate through this dictionary to create a capability allow rule for capabilities that the traced program has.

The last function called by `generate_policy_allow()` is `generate_ipc()`. This function uses two regex strings that extract both Pipe and Unix Socket events. These events are iterated through and added to arrays based on if the result is greater than zero. Then both valid pipe and socket rules are used in combination to determine if a process is communicating with any other processes. The output security policy fragment will only contain data if the `generate_ipc()` function is able to determine that inter process communication is occurring.

Finally, after the completion of the `generate_policy_allow()` function, the `generate_policy_restrict()` function is called. This function calls all the same methods as `generate_policy_allow()`, however they are all passed the value `allowBool=False`. This changes the structure of each function to instead check for results that are less than 0, indicating a failed operation.

Once all the functions in the `TraceToPolicy()` class have been executed, it returns a formatted string containing a combination of all the generated rules for the program being traced. This string can then be written to a YAML file, which will populate it with the generated security policy.

4. Results

Currently, the whole implementation can automatically generate BPFContain security policies for every feature that is able to be controlled by BPFContain, except for networking events and process tainting. This section will outline any future work that could be done to improve the project as well as information about the results that have been gathered so far.

4.1 Future Improvements

If there is time remaining in the semester, I would like to implement some functionality for the tracing and translation of networking events. BPFContain policy for networking is a simple set of rules that outline if the process can act as a client or server and if it can send or receive networked data [1]. Networking could be traced by probing `TCP` and `UDP`, `connect` and `accept` functions. However, more research is still needed to determine the best approach. If any part of this feature is completed in time, it will be included with the final version of this report.

Regarding process tainting, an implementation could define a set of operations that are deemed to be unsafe. When any of these operations is seen in the trace, they would be added to an array that tracks all the taint traces. Tainting outlines the same rules as the restrict and allow sections, however, when one of these rules is executed, it will cause the running process to become tainted. Once a process is tainted, its security policy becomes stricter [1]. These rules should restrict when a program tries to perform an operation that would make it less trusted [1]. A basic networking taint rule could be determined by if a policy with no allowed network access, tries to access a network. However, filesystem and inter process communication rules may be harder to automate and will require further research, as a custom taint tracking system will need to be developed. Current research in the field of taint tracking includes papers with implementations such as CONFLUX [20] and TaintEraser [21].

4.2 Tracing Program Results

Using the program `bash` with a test workload of using device files, creating, and removing directories, creating files, reading from, and writing to files, renaming, and moving files, changing file ownership and permissions, interacting with pipes, opening subprocesses using command line pipes, causing capability checks, and opening new bash shells. The operations within the workload, test for both operations that should be allowed and ones that should be denied. Ensuring that the generated policy will have both the allow and restrict sections. The final version of this report will contain more extensive results for a variety of different programs to demonstrate the capture results of all supported events. It will also include the results from running the generated security policy with BPFContain.

The trimmed output of the tracing program can be seen in section 7.1 of the appendix. The output has been trimmed to show only lines that are relevant to the translation program when generating security policies. The full version of the trimmed log file in section 7.1 of the appendix can be given as an argument to the translation program, which will generate the final security policy shown in section 7.2 of the appendix.

5. Conclusion

In conclusion, the functionality of the final implementation of this project has surpassed the initial objective of only tracing and translating filesystem operations. Overall, I believe the project has demonstrated the capabilities of eBPF and bpftrace are sufficient for use in the automatic generation of BPFContain security policies. In the future, networking features and process tainting could be added. However, even without these functionalities the project can automatically generate meaningful BPFContain security policies, which can aid in the reduction of time and effort required when manually creating these policies.

6. References

[1] W. P. Findlay, "A Practical, Lightweight, and Flexible Confinement Framework in eBPF," Carleton University, Ottawa, 2021.

[2] Red Hat, "What is the Linux kernel?," 27 February 2019. [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel.

[3] eBPF, "eBPF Documentation - What is eBPF?," [Online]. Available: https://ebpf.io/what-is-ebpf.

[4] R. Gooch, "Overview of the Linux Virtual File System," [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/vfs.html.

[5] iovisor, "GitHub - bpftrace," [Online]. Available: https://github.com/iovisor/bpftrace.

[6] J. Keniston, P. S. Panchamukhi and M. Hiramatsu, "Kernel Probes (Kprobes)," [Online]. Available: https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[7] S. Dronamraju, "Uprobe-tracer: Uprobe-based Event Tracing," [Online]. Available: https://www.kernel.org/doc/html/latest/trace/uprobetracer.html.

[8] M. Desnoyers, "Using the Linux Kernel Tracepoints," [Online]. Available: https://www.kernel.org/doc/html/latest/trace/tracepoints.html.

[9] "audit2allow(1) - Linux man page," [Online]. Available: https://linux.die.net/man/1/audit2allow.

[10] guardicore, "IPCDump," [Online]. Available: https://github.com/guardicore/IPCDump.

[11] iovisor, "gobpf," [Online]. Available: https://github.com/iovisor/gobpf.

[12] dtrace.org, "About DTrace," [Online]. Available: http://dtrace.org/blogs/about/.

[13] SystemTap, "SystemTap wiki," [Online]. Available: https://sourceware.org/systemtap/wiki.

[14] E. Rocca, "Comparing SystemTap and bpftrace," 13 April 2021. [Online]. Available: https://lwn.net/Articles/852112/#:~:text=The%20important%20design%20distinction%20between.

[15] iovisor, "BPF Compiler Collection (BCC)," [Online]. Available: https://github.com/iovisor/bcc.

[16] Fedora Project, "Using the DNF software package manager," [Online]. Available: https://docs.fedoraproject.org/en-US/quick-docs/dnf/.

[17] "fs/open.c - Linux Source Code," [Online]. Available: https://elixir.bootlin.com/linux/latest/source/fs/open.c#L565.

[18] iovisor, "bpftrace Install," [Online]. Available: https://github.com/iovisor/bpftrace/blob/master/INSTALL.md.

[19] "GitHub - bpftrace/tools/capable.bt," [Online]. Available: https://github.com/iovisor/bpftrace/blob/master/tools/capable.bt.

[20] K. Hough and J. Bell, "A Practical Approach for Dynamic Taint Tracking with Control-flow Relationships.," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 31.2, pp. 1-43, 2021.

[21] D. Zhu, J. Jung, D. Song, T. Kohno and D. Wetherall, "TaintEraser: Protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review,* vol. 45.1, pp. 142-154, 2011.

7. Appendix

7.1 Example Trimmed Trace File Output

```
Attaching 34 probes...
Tracing Filesystem Function and Returns ... Hit Ctrl-C to end.
VFS: Path: /etc/terminfo/x/xterm-256color, Program: bash, Tid: 12449, Mode: Access, Return: -2
VFS: Path: /etc/bash_completion.d/redefine_filedir, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /usr/lib64/gconv/gconv-modules.cache, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/bash_completion.d/lilv, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/bashrc, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 512
VFS: Path: /etc/passwd, Program: bash, Tid: 12449, Mode: Read, Return: 2648
VFS: Path: /etc/rc.d/init.d, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/bash_completion.d/authselect-completion.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Write, Return: 30
VFS: Path: /usr/lib64/libc.so.6, Program: bash, Tid: 12449, Mode: Read, Return: 832
VFS: Path: /usr/bin/flatpak, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/colorzgrep.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 9
VFS: Path: /etc/profile.d/vte.sh, Program: bash, Tid: 12449, Mode: Read, Return: 2166
VFS: Path: /etc/passwd, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/bash_completion.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 0
VFS: Path: /usr/lib64/libtinfo.so.6.2, Program: bash, Tid: 12449, Mode: Read, Return: 832
VFS: Path: /etc/profile.d/colorxzgrep.sh, Program: bash, Tid: 12449, Mode: Read, Return: 183
VFS: Path: /etc/profile.d/flatpak.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/bash_completion.d/redefine_filedir, Program: bash, Tid: 12449, Mode: Read, Return: 2167
VFS: Path: /etc/bash_completion.d/gluster, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 15
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 25
VFS: Path: /etc/authselect/nsswitch.conf, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/bash_completion.d/python-argcomplete, Program: bash, Tid: 12449, Mode: Read, Return: 4108
VFS: Path: /usr/lib64/libtinfo.so.6.2, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/PackageKit.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/init.d/livesys-late, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/inputrc, Program: bash, Tid: 12449, Mode: Read, Return: 943
VFS: Path: /etc/bash_completion.d, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /usr/share/bash-completion/bash_completion, Program: bash, Tid: 12449, Mode: Read, Return: 76322
VFS: Path: /etc/profile.d/colorls.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/gawk.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/gawk.sh, Program: bash, Tid: 12449, Mode: Read, Return: 757
VFS: Path: /etc/profile.d/nano-default-editor.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/bash_completion.d/python-argcomplete, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/less.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/colorls.sh, Program: bash, Tid: 12449, Mode: Read, Return: 1431
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 5
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Write, Return: 44
VFS: Path: /etc/bash_completion.d/dog, Program: bash, Tid: 12449, Mode: Read, Return: 24232
VFS: Path: /etc/profile.d/vte.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 12
VFS: Path: /etc/profile.d/which2.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /usr/lib64/libc.so.6, Program: bash, Tid: 12449, Mode: Read, Return: 68
VFS: Path: /usr/share/terminfo/x/xterm-256color, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/less.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/authselect/nsswitch.conf, Program: bash, Tid: 12449, Mode: Read, Return: 0
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Write, Return: 85
VFS: Path: /etc/profile.d/flatpak.sh, Program: bash, Tid: 12449, Mode: Read, Return: 831
VFS: Path: /etc/init.d/README, Program: bash, Tid: 12449, Mode: Access, Return: -13
VFS: Path: /etc/bash_completion.d/gluster, Program: bash, Tid: 12449, Mode: Read, Return: 11525
VFS: Path: /etc/bash_completion.d/abrt.bash_completion, Program: bash, Tid: 12449, Mode: Read, Return: 43
VFS: Path: /etc/rc.d/init.d/README, Program: bash, Tid: 12449, Mode: Access, Return: -13
VFS: Path: /etc/profile.d/PackageKit.sh, Program: bash, Tid: 12449, Mode: Read, Return: 1322
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 26
VFS: Path: /usr/share/terminfo/x/xterm-256color, Program: bash, Tid: 12449, Mode: Read, Return: 3814
VFS: Path: /etc/bash_completion.d/python-argcomplete, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/colorls.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/colorzgrep.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /usr/lib64/libc.so.6, Program: bash, Tid: 12449, Mode: Read, Return: 784
VFS: Path: /etc/bash_completion.d/dog, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/rc.d/init.d/livesys, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/bash_completion.d/gluster, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /usr/lib64/libc.so.6, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /usr/lib64/libc.so.6, Program: bash, Tid: 12449, Mode: Read, Return: 80
VFS: Path: /etc/inputrc, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Write, Return: 9
VFS: Path: /usr/share/terminfo/x/xterm-256color, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/lang.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
```

```
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Write, Return: 5
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 1
VFS: Path: /usr/bin/bash, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /home/jakejazokas/.config/bash_completion, Program: bash, Tid: 12449, Mode: Access, Return: -2
VFS: Path: /etc/bash_completion.d/authselect-completion.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /usr/bin/bash, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/gawk.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/PackageKit.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /home/jakejazokas/.bash_history, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 416
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Write, Return: 8
VFS: Path: /etc/profile.d/colorxzgrep.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Write, Return: 1
VFS: Path: /etc/bash_completion.d/abrt.bash_completion, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/init.d/functions, Program: bash, Tid: 12449, Mode: Access, Return: -13
VFS: Path: /etc/profile.d/colorxzgrep.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/bash_completion.sh, Program: bash, Tid: 12449, Mode: Read, Return: 726
VFS: Path: /null, Program: bash, Tid: 12449, Mode: Write, Return: 17
VFS: Path: /etc/bash_completion.d/lilv, Program: bash, Tid: 12449, Mode: Read, Return: 1931
VFS: Path: /etc/profile.d/bash_completion.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/colorgrep.sh, Program: bash, Tid: 12449, Mode: Read, Return: 201
VFS: Path: /etc/profile.d/debuginfod.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/bash_completion.d/authselect-completion.sh, Program: bash, Tid: 12449, Mode: Read, Return: 6481
VFS: Path: /etc/bash_completion.d/dog, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/lang.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/ld.so.cache, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Write, Return: 15
VFS: Path: /usr/bin/sed, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/bash_completion.d/abrt.bash_completion, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /usr/bin/pidof, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/debuginfod.sh, Program: bash, Tid: 12449, Mode: Read, Return: 180
VFS: Path: /etc/profile.d/colorgrep.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/vte.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /usr/share/bash-completion/bash_completion, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/ld.so.preload, Program: bash, Tid: 12449, Mode: Access, Return: -2
VFS: Path: /etc/profile.d/less.sh, Program: bash, Tid: 12449, Mode: Read, Return: 253
VFS: Path: /usr/lib64/ld-linux-x86-64.so.2, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /usr/lib/locale/locale-archive, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /home/jakejazokas/.bashrc, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /2, Program: bash, Tid: 12449, Mode: Read, Return: 1
VFS: Path: /etc/bash_completion.d/redefine_filedir, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/init.d/livesys, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/which2.sh, Program: bash, Tid: 12449, Mode: Read, Return: 478
VFS: Path: /etc/profile.d/lang.sh, Program: bash, Tid: 12449, Mode: Read, Return: 3187
VFS: Path: /etc/profile.d/colorgrep.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/nano-default-editor.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/rc.d/init.d/livesys-late, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /usr/bin/bash, Program: bash, Tid: 12449, Mode: Execute, Return: 0
VFS: Path: /usr/share/bash-completion/bash_completion, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /tty, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/rc.d/init.d/functions, Program: bash, Tid: 12449, Mode: Access, Return: -13
VFS: Path: /etc/bashrc, Program: bash, Tid: 12449, Mode: Read, Return: 2917
VFS: Path: /etc/bash_completion.d, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /etc/profile.d/colorzgrep.sh, Program: bash, Tid: 12449, Mode: Read, Return: 220
VFS: Path: /etc/profile.d/nano-default-editor.sh, Program: bash, Tid: 12449, Mode: Read, Return: 120
VFS: Path: /etc/bash_completion.d/lilv, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /home/jakejazokas/.bash_completion, Program: bash, Tid: 12449, Mode: Access, Return: -2
VFS: Path: /null, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /etc/profile.d/debuginfod.sh, Program: bash, Tid: 12449, Mode: Access, Return: 0
VFS: Path: /usr/share/terminfo/x/xterm-256color, Program: bash, Tid: 12449, Mode: Read, Return: 0
VFS: Path: /, Program: bash, Tid: 12449, Mode: Read, Return: 119
VFS: Path: /etc/profile.d/flatpak.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
VFS: Path: /home/jakejazokas/.bash_history, Program: bash, Tid: 12449, Mode: Read, Return: 15061
VFS: Path: /etc/authselect/nsswitch.conf, Program: bash, Tid: 12449, Mode: Read, Return: 3038
VFS: Path: /home/jakejazokas/.bashrc, Program: bash, Tid: 12449, Mode: Read, Return: 492
VFS: Path: /etc/profile.d/which2.sh, Program: bash, Tid: 12449, Mode: Open, Return: 0
Pipe: Path: /, Program: bash, Mode: Write, Return: -32
Pipe: Path: /home/jakejazokas/Desktop/namedPipe, Program: bash, Mode: Write, Return: 8Signal: From: bash, To:
bash, SigNum: 13, Return: 0
Signal: From: bash, To: bash, SigNum: 17, Return: 0
Signal: From: bash, To: sudo, SigNum: 17, Return: 0
Signal: From: bash, To: bash, SigNum: 17, Return: 2
Capability: Program: bash, CapInt: 1, CapName: CAP_DAC_READ_SEARCH, Options: 0, Return: -1
Capability: Program: bash, CapInt: 2, CapName: CAP_DAC_OVERRIDE, Options: 0, Return: -1
```

7.2 Automatically Generated Policy for Bash

```
name: bash
cmd: /usr/bin/bash

defaultTaint: false

allow:
  - device: terminal
  - device: null

  - file: {path: /usr/share/terminfo/x/xterm-256color, access: ra}
  - file: {path: /root/.bash_history, access: rwm}
  - file: {path: /etc/bash_completion.d/lilv, access: ra}
  - file: {path: /home/jakejazokas/.bashrc, access: r}
  - file: {path: /etc/bash_completion.d/authselect-completion.sh, access: ra}
  - file: {path: /etc/passwd, access: r}
  - file: {path: /usr/lib64/libc.so.6, access: r}
  - file: {path: /etc/bash_completion.d/dog, access: ra}
  - file: {path: /etc/inputrc, access: r}
  - file: {path: /etc/profile.d/gawk.sh, access: ra}
  - file: {path: /etc/profile.d/PackageKit.sh, access: ra}
  - file: {path: /root/.bashrc, access: r}
  - file: {path: /etc/profile.d/vte.sh, access: ra}
  - file: {path: /etc/bash_completion.d/redefine_filedir, access: ra}
  - file: {path: /usr/lib64/libtinfo.so.6.2, access: r}
  - file: {path: /etc/profile.d/colorxzgrep.sh, access: ra}
  - file: {path: /etc/profile.d/bash_completion.sh, access: ra}
  - file: {path: /etc/profile.d/colorls.sh, access: ra}
  - file: {path: /etc/profile.d/less.sh, access: ra}
  - file: {path: /etc/bash_completion.d/python-argcomplete, access: ra}
  - file: {path: /etc/authselect/nsswitch.conf, access: r}
  - file: {path: /usr/share/bash-completion/bash_completion, access: ra}
  - file: {path: /etc/bashrc, access: r}
  - file: {path: /etc/bash_completion.d/gluster, access: ra}
  - file: {path: /etc/profile.d/colorzgrep.sh, access: ra}
  - file: {path: /etc/profile.d/flatpak.sh, access: ra}
  - file: {path: /etc/bash_completion.d/abrt.bash_completion, access: ra}
  - file: {path: /etc/profile.d/colorgrep.sh, access: ra}
  - file: {path: /etc/profile.d/debuginfod.sh, access: ra}
  - file: {path: /etc/profile.d/which2.sh, access: ra}
  - file: {path: /etc/profile.d/nano-default-editor.sh, access: ra}
  - file: {path: /etc/profile.d/lang.sh, access: ra}
  - file: {path: /home/jakejazokas/.bash_history, access: r}
  - file: {path: /home/jakejazokas/Desktop/temp.txt, access: w}
```

```
  - file: {path: /usr/bin/cat, access: a}
  - file: {path: /usr/bin/flatpak, access: a}
  - file: {path: /etc/bash_completion.d, access: a}
  - file: {path: /usr/bin/sed, access: a}
  - file: {path: /etc/init.d/livesys-late, access: a}
  - file: {path: /usr/bin/bash, access: ax}
  - file: {path: /etc/init.d/livesys, access: a}
  - file: {path: /etc/rc.d/init.d/livesys-late, access: a}
  - file: {path: /etc/rc.d/init.d/livesys, access: a}
  - file: {path: /usr/bin/lesspipe.sh, access: a}
  - file: {path: /usr/sbin/pidof, access: a}
  - file: {path: /usr/bin/register-python-argcomplete, access: a}
  - file: {path: /usr/bin/pidof, access: a}
  - file: {path: /usr/bin/grep, access: a}

  - fs: {pathname: /, access: rwm}

  - signal: {to: bash, signals: [sigPipe, sigChld]}
  - signal: {to: sudo, signals: [sigChld]}

  - ipc: namedPipe

restrictions:

  - file: {path: /etc/ld.so.preload, access: a}
  - file: {path: /etc/terminfo/x/xterm-256color, access: a}
  - file: {path: /root/.bash_completion, access: a}
  - file: {path: /etc/rc.d/init.d/functions, access: a}
  - file: {path: /etc/init.d/README, access: a}
  - file: {path: /etc/rc.d/init.d/README, access: a}
  - file: {path: /home/jakejazokas/.config/bash_completion, access: a}
  - file: {path: /home/jakejazokas/.bash_completion, access: a}
  - file: {path: /etc/init.d/functions, access: a}
  - file: {path: /root/.config/bash_completion, access: a}

  - fs: {pathname: /, access: w}

  - capability: [dacOverride, dacReadSearch]
```