

Practicum Project Report

BCIT COMP8045 – Half Practicum

Andrew Burian
A00852121

28-04-2016

SKAFE: Scriptable Kernel Auditing For Endpoints

Table of Contents

Introduction.....	3
A. Background.....	4
B. Project Statement.....	5
C. Alternate Solutions.....	6
Auditd.....	6
Syslog.....	7
OSSEC.....	8
D. Chosen Solution.....	9
E. Development Details.....	11
Deliverables.....	11
Technical Challenges.....	11
Program Structure & Data Flow.....	12
Agent.....	12
Server.....	14
The Rule Engine.....	15
Operation Manual.....	18
Server.....	18
Compiling & Installing.....	18
Configuration.....	18
Running.....	21
Agent.....	22
Compiling & Installing.....	22
Configuration.....	22
Running.....	23
F. Testing and Results.....	24
Test Design.....	24
Testing Tools.....	24
List of Unit Tests.....	26
Agent Tests.....	27
Server Tests.....	28
System Test.....	31
G. Implementation Details.....	33
For Testing.....	33
For Performance.....	33
For Flexibility.....	34
For Future Development.....	34
H. Research Done.....	36
Kernel Auditing.....	36
Concurrent Systems.....	36
Golang Features.....	36
Linux Daemons and Systems Design.....	37
I. Future Work.....	38
Documentation.....	38
Installer Package.....	38

Script Languages.....	38
Script Builtin Features.....	39
Automated Integration Testing.....	39
Performance Benchmarks.....	39
J. Timeline.....	40
Stage Breakdown.....	40
Stage 0 – App Core.....	40
Stage 1 – Basic Rule Engine.....	40
Stage 2 – Application Durability.....	41
Stage 3 – Script Rules.....	41
Overall Timeline.....	42
Conclusion.....	43
Appendix.....	44
List of Terms and Program Names.....	44

Introduction

Auditing systems can be technically challenging and difficult, but good visibility into endpoint systems is essential for both security awareness, and potentially for compliance reasons in a corporate environment. This project aims to simplify and unify kernel auditing across multiple systems, which can then be richly managed and processed at one central location.

Current solutions for kernel auditing make use of projects such as the kernel audit daemon *auditd* which delivers kernel audit events in log form, and then ship these logs to be collected centrally for further processing. The logs are then processed with a SIEM (Security Information and Event Management) tool such as *Splunk*, or *ArcSight* to perform analysis and correlation on these log events. This provides adequate security for most compliance needs, but often does not contain rich enough data to perform meaningful analysis without human operators having to manually investigate retroactively.

SKAFE will aim to provide more complete data on events, and allow scripts to be run on the events in real time, allowing much more analysis to be done automatically on events without requiring human intervention. SKAFE will have two components: the agent, and the server. One server will be able to manage many agents, and each agent will report to only one server.

The agent, similar to *auditd*, will connect to the kernel's audit subsystem using the Linux NETLINK interface, and receive system audit events directly from the kernel. In userspace, the agent will add as much context as possible to these events, such as resolving uid's to usernames, pid's to process names, providing hash sums of executed binaries, and even the name of the kernel container the event is coming from, making SKAFE default container-aware. This rich event is then shipped in full to the server, over a mutually authenticated TLS link to ensure data security and completeness. If the server is unavailable or overburdened, the agent will compress and cache the events locally until the server returns to full operation.

On the server, the events will be received, and have one more round of information added to them, including the agent name they were received from, and any IP to DNS name resolving. The full event is then processed through the server's rule engine. The rule engine consists of passing the event through a decision tree of nodes, where each node is a single rule. The decision tree is completely user-defined, and very customizable. Rule's can either be simple matching rules such as “process.name = /bin/sudo”, or they may be entire scripts written in Ruby which can perform any function the user can code in, such as checking the hash against an online database. The scripts will also have access to some SKAFE defined functions, such as to get a complete copy of the binary in question, which is then fetched from the endpoint by the agent.

This utility will therefore allow complex and in-depth analysis and decision making to happen efficiently and automatically in one central and easily manageable point, easing the burden on human security analysts, and providing more complete insight on events happening on a network.

A. Background

One of the most essential things a security solution needs to include is the use of system and kernel level auditing for all the endpoints on a network. Since the Linux kernel version 2.6.6, the kernel has had an audit subsystem built into it which is capable of presenting nearly all the happenings on a system to a user-space application. The most common tool for interacting with this interface is the well established *auditd* project, which can be configured to request certain types of events from the kernel audit system, and logs them for later inspection.

When developing a security solution for multiple endpoints, it becomes necessary to centralize this information. If the endpoint is using *auditd*, logs can be shipped either with a log collection agent such as *OSSEC*, or sent through the unified logging agent *rsyslog* to a remote logging server. Now centralized, these logs must be parsed to provide any useful context. Again, *auditd* provides this functionality with the tools *ausearch*, and *aureport* which act as sort of a *grep* for audit logs. With these tools, to build an enterprise level security solution that uses audit, the events must travel through at least two different programs, the auditor and the log shipper, and then be parsed using another toolkit still to generate useful analysis, which requires human interaction. The usefulness of audit logs becomes relegated to being used as a paper-trail from which to reconstruct events after the fact.

Audit logs tend to also lack user-friendly information. Parent processes for process execution is simply given as a pid, and not the binary path for example. Although the audit logs will show if a binary is created in the /tmp folder, executed, and then removed, it does not give any indication as to what may have been in that binary. All this further analysis, if desired, must be done by hand, and may require manually logging into the endpoint in question. If too much time has elapsed since the event, the forensic evidence may already be lost. In any case, this also creates the requirement for security analysts who have the time and ability to perform all this leg work, assuming that they are even alerted to a potential problem at all. For a large environment of potentially hundreds of busy endpoints, this is often an unreasonable burden on the staff, and these manual tasks need to be replaced with automated intelligence.

Solutions do exist for managing the flow of logs and events into indexes and correlating separate events into larger understandings and altering on problems. Solutions like *Splunk* or *ELK (Elasticsearch Logstash Kibana)* can ingest large amounts of data and make it easier for operators to query, giving them better operational awareness. More complete SIEM (Security Information and Event Management) tools such as HP's *Arcsight* take this a step further. It can be programmed to match indicators and correlate events into specific alerts, and manage investigations between several analysts. These tools, however, operate at a step removed from the actual endpoints themselves. They sit on the end of a pipe which feeds them the event information, and then they act on it. They still lack ability to perform some deeper interaction themselves.

B. Project Statement

This project will aim to deliver a powerful and flexible tool for easily maximizing use of the difficult and often overlooked kernel audit events. It will feature easy configuration for rapid development, and a easily customized and integrated design by allowing user scripts to make up the core logic of the application. The application must perform well enough in terms of speed and resource footprint to conceivably be deployed on busy production machines without affecting their performance.

C. Alternate Solutions

Currently, there are dozens if not hundreds of commercial and open source projects aimed at providing different types of systems and audit monitoring, as well as log collection and event processing. Examining some of these provide the foundation, and sometimes integration that SKAFE will be built off of, and to work alongside of.

Auditd

Auditd is by far the most well known and widely used kernel auditing tool used anywhere. It is not wrong to call it the standard for collecting and manipulating kernel audit logs. It is divided into three components, two of which will actually be used to integrate it with SKAFE.

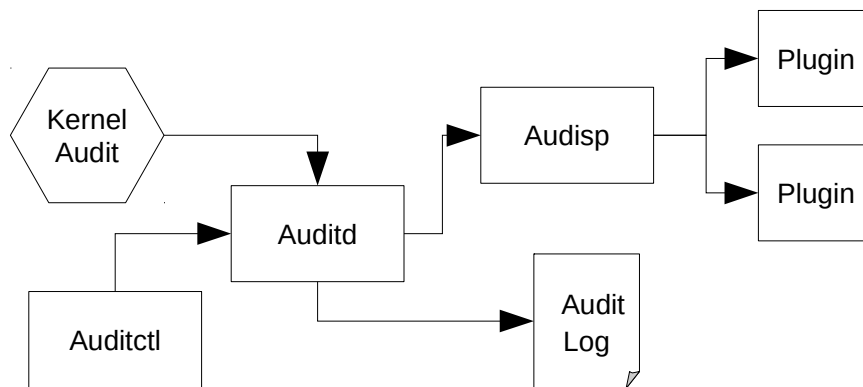


Illustration 1: Auditd component diagram

First, there is the *auditd* daemon itself. The daemon registers itself with the kernel as the system's user-space audit recipient, and manages the connection to the kernel over the Netlink interface. The helper program *auditctl* commands the daemon to register or deregister certain watches with the kernel. Once an event is received by *auditd* from the kernel, it is sent two places: the audit log configured in the daemon's settings (typically `/var/log/audit/audit.log`), and to the second component of the system called *audisp*.

Audisp is the audit event multiplexer. Started as a child process of *auditd* itself, *audisp* can be configured to launch any number of child processes of its own and pass along copies of any event it receives from *auditd*. This simple multiplexer allows *auditd* to be very extensible, and has been included in the design of SKAFE as a deployment option.

The *auditd* project also includes C libraries and helper programs for parsing and exploring the stream of audit events that pass through *auditd* and its child plugins.

Syslog

Similar projects to *syslog* include the more modern *rsyslog*, and *syslog-ng*, all of which accomplish similar goals within the context of this project.

Syslog, or more generally the system logger, is an application which can receive and process logs from any system service or daemon, and handle them according to a user definable set of rules. Every system under Linux has the ability to connect to the system logger and send log events to it for handling. Each log sent to the system logger contains a message, a priority, and a facility which is used to distinguish between different types of logs.

Below is the complete list of logging levels and the most common *syslog* facilities available.

Levels		Facilities	
Name	Description	Name	Description
EMERG	System is unusable	AUTH	Authorization messages
ALERT	Immediate action required	AUTH_PRIV	Private messages (may contain sensitive data)
CRIT	Critical conditions	CRON	Messages from the cron daemon
ERROR	Error conditions	DAEMON	System daemons without their own facility
WARNING	Warning conditions	FTP	FTP daemon messages
NOTICE	Normal, but significant conditions	KERN	Kernel messages
INFO	Informational messages	MAIL	Mail system messages
DEBUG	Debug messages	USER	Generic user-level messages

With this central logging facility, Linux security systems tend to use *syslog* to manage events from several distinct systems at once. Newer versions like *rsyslog* can operate in a client-server arrangement, and ship logs over the network to a central *syslog* accumulator. This remote logging practice is widely used to amalgamate logs from many systems running many applications into one single place for analysis and processing.

Syslog is essentially ubiquitous in Linux systems, with the minor exception of newer *systemd* based systems starting to phase out its use in favour of *systemd-journald*. It is safe to assume however, that any auditing solution that produces log or alerts should be able to interface with the system logger to be part of a larger system at play.

OSSEC

OSSEC is an open source host based intrusion detection system, that has been widely used to perform security monitoring on numerous systems from one central server.



Illustration 2: OSSEC System Architecture (from ossec.github.io)

OSSEC works by placing an agent on every system that is going to be monitored. Each agent is configured with the location of various system and application log files that are available on its host system, and begins monitoring the files for additions. Each time an application or system performs any action that generates a log, the log message is written to the appropriate log file, and then the agent monitoring the file becomes alerted to the change, and copies the entry out of the file itself. In this way the agent gathers every event of every application off the system in real time.

Once the agents have a log, the message is sent to the server as-is, with only a small amount of additional information added such as the time the log entry was created, and the host name of the machine the agent was on. The bulk of the processing happens on the OSSEC server. The server receives each log entry, and consults an XML configuration containing a list of rules and patterns to match against the entry. These rules can then trigger other rules, or even active responses that are pushed back to the agent to be run on the system in question.

OSSEC can be set to archive every log entry it receives, or selectively log some entries as well as any alerts generated by the matching rules.

D. Chosen Solution

The solution for this use case was the construction of the custom auditing tool: SKAFE. This project draws on the design and processes of the other project, aims to work with and alongside some common tools, and improves on the flexibility, performance, and ease of use compared to others.

SKAFE focuses solely on kernel audit events. These events are perhaps some of the most valuable pieces of telemetry coming from any Linux system, but in their default form are cumbersome to read by a human, parse programatically, or correlate with other events. They are also limited in that they often report kernel style information rather than the more useful userspace equivalents. For example, users are always represented by their IDs and files by inode numbers, rather than the more helpful usernames and file paths. As such they have proven difficult for more general log collecting and auditing systems to make use of without dedicating substantial resources to first cleaning up the logs, a special treatment which may not even be possible depending on the system design.

The design of SKAFE at a high level is conceptually similar to that of *OSSEC*. Building on the methods *OSSEC* has proven to work, SKAFE features an agent-server model, where a multitude of agents all contact a single central server. Most of the intensive processing is done on the server to keep the resource footprint of the agents to a minimum, and allow dedicated systems to run the server.

The ability for an administrator to create completely custom matching rules for received events is also borrowed from the success of *OSSEC*, but with some changes to enhance usability. Rather than the XML format and multitude of options, a simpler INI configuration file is simpler to maintain and update. There is also an increased focus on rule chaining. Each rule in SKAFE explicitly has a “watch” parameter indicating the event that must be triggered prior to this rule itself being evaluated. Regular expression matching has two supported types: the POSIX standard regular expression patterns, and the more common Perl patterns.

SKAFE includes no active response capabilities itself. It will expose its alerts and logs in such a way that other applications may easily hook into them and perform this role, but SKAFE itself will not support it. This decision is based largely in the philosophy that in a production environment, the fewer applications that may execute arbitrary commands at their own will the better. SKAFE will have its hands full delivering the most powerful audit analysis engine possible, and is more suited to provided telemetry than it is to acting on it.

With the evident success of the *auditd* project, SKAFE is not designed to be in competition with *auditd*, but at the same time, with SKAFE's requirement to be deployed in any situation, it must also not be dependant on the presence of *auditd*. SKAFE is therefore able to function in two modes. The first being independent from *auditd*, assigning itself as the system audit daemon and connect directly to the kernel to receive audit events and parse them. The second mode will assume that *auditd* is running, and so cannot register itself as the auditor without disrupting *auditd*. It instead takes advantage of *auditd*'s

plugin system, and register itself as a child process to be spawned by *audisp*. Events are then read from the standard input, but processing will otherwise continue as normal.

To support a variety of different ways security systems may want to use SKAFE's output, an equally varied number of output options must be present. Log files are the standard means of a daemon reporting on its activity, and SKAFE is no exception, so provisions are made for server logs, event logs, and alert logs to all be captured to separate or unified log files. As mentioned previously, the Linux system logger is also an incredibly powerful and common way of daemon's communicating activity, so SKAFE is also able to redirect any or all of its three log streams to the system logger. Finally, to accommodate both systems where SKAFE may be run in the foreground as a terminal application and systemd systems which capture standard outputs of programs for the journal, the application also supports logging directly to either stdout or stderr.

A powerful addition to SKAFE that sets it apart from other solutions is its ability to have user scripted rules in addition to the simple regular expression matching rules. SKAFE, as part of the central rule tree, will allow users to specify that an event triggering this rule should be passed as input to a function defined by the user in a common scripting language. What happens then is entirely up to the discretion of the creator of the script. These scripts will be run in unrestricted interpreters as native applications themselves and should be able to perform anything from more complex analysis on the event itself, to contacting arbitrary external resources for additional input. This freedom is what gives SKAFE its edge over other solutions, and while some systems support scripting in addition to other base functions, scripting is SKAFE's base function and will more readily allow complete control to the system administrators.

The script engine itself is also built in such a way that future development could add functions into the script interpreters that interface back to SKAFE and perform actions that take advantage of the trust relationship the server already has with the agents, rather than establishing new connections in the script itself.

SKAFE is written in the Golang programming language. This language was selected for a number of reasons, primarily among them are its ability to create and manage lightweight “green threads” or application level threads that greatly simplify working with the amount of concurrency present in SKAFE. Also, Golang's inter-thread communication model and built in primitive types like the buffered channel are ideal for the sort of system communication that SKAFE is doing internally. Golang's testing framework is the final large reason for the language selection. As is evident in the Testing section, Golang's unit testing abilities are very powerful.

E. Development Details

Deliverables

SKAFE is intended to fit into a complex niche in the overall security system of any network. It should both be a powerful standalone tool, and also work nicely along side existing systems, all the while being very high performance and error tolerant.

Also taken into consideration is that this application will almost certainly never be completely “finished” let alone finished in the relatively short 400 hours of this project. Security is forever a moving target, and as new systems and processes come and go, SKAFE will need to continue to adapt to fit into whatever situation it can conceivably be thrown into.

These following deliverable represent the goals to be achieved at the end of the initial 400 hours of development, not those of a completely finished project.

- SKAFE Agent capable of being loaded as an *auditd* plugin, enrich events, and securely transmit to the server
- SKAFE Server capable of receiving events from a multitude of agents, and running them on a user defined rule tree to trigger alerts.

A more detailed list of deliverables is available in the table breakdown of the [Timeline](#) section.

Specific rational behind the features of SKAFE are covered in the [Chosen Solution](#) section.

Technical Challenges

The main technical challenges come from the requirements to make this system useful at a large scale. On a busy system, dozens of audit events could be being produced every second. Multiply that by tens of endpoints, and performance optimization becomes the most critical feature of the whole project.

- Extremely tight memory management
The decision to use Golang has the drawback that Golang is a garbage collected language, which is costly when lots of allocating and freeing is being done. Overcoming this by object reuse, and careful allocation will be a challenge.
- Maximum possible concurrency
Golang offers lightweight thread abstractions, but they still need to be implemented properly. To allow a server to scale to hundreds of endpoints, we can't rely on hardware getting faster, but we can add more processors. Making good use of lots of cores will not be straightforward.
- No-loss policy
For the system to be usable in a corporate environment where compliance requirements may be

critical, the system must be fail safe, and never lose any events under any circumstance.

- **Highly configurable**

This project should not make too many assumptions about the situations or use-cases it is going to be deployed in. It should be able to work well in a large corporate environment, or a small lab. Consequently, little should be hard coded, and most should be easily and quickly configurable.

- **Integration with *auditd***

In line with the above requirement to be highly configurable, we cannot assume that this solution will be deployed in isolation, and in fact is more likely to be deployed along side existing solutions that depend on *auditd*. This application will have to be able to be launched as an audisp daemon and not interfere with the workings of *auditd*.

Program Structure & Data Flow

SKAFE was designed with modularity in mind, the reasons for which are summarized in the [Implementation Details](#) section. For the remainder of this document, the SKAFE program will be talked about in these terms. SKAFE refers to the entire system, comprised of one or multiple agents and a single server. The server and agents are each referred to as their own application. Within each of the applications, there are multiple systems which represent the working core of the project, and each system typically has its own thread.

Listed below in the Agent and Server sections are all of the internal systems in the order in which data flows through them. The data between systems is always a key-value map representing the audit event.

Agent

The SKAFE agent is composed of six systems. Two of which, the auditor and the *audisp* systems, are mutually exclusive, as only one of them will be running at any point.

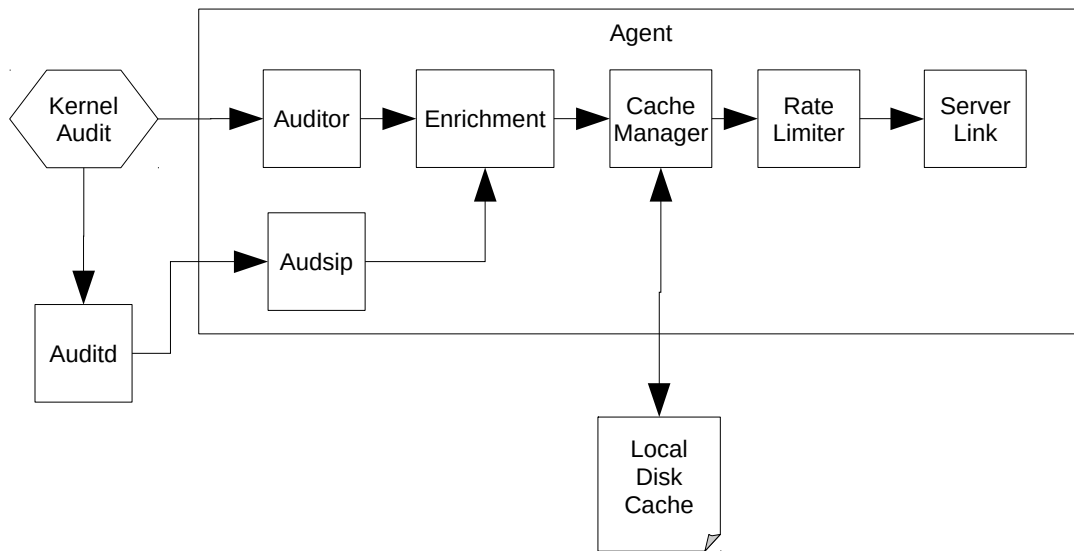


Illustration 3: SKAFE agent systems diagram

- **Auditor**
The auditor system registers the application with the kernel as the system audit daemon, and opens the Netlink socket on which to receive kernel audit events from. Multiple audit events are compressed into a single event if they correspond to the same kernel event.
- **Auditsip**
If the agent is going to be run as an *auditd* plugin, this system is run instead of the auditor system. Rather than opening a netlink socket, this system reads event data from the application's standard input, as the *auditd* multiplexer auditsip will run this application as a child process, and send event data to it over standard input. Otherwise does all the same processing as the auditor system.
- **Enrichment**
This system adds additional data to each event received from the input systems. The enrichment system performs user id resolution, full command reconstruction, binary hashing, and gathers any other information that may be pertinent to the event in question.
- **Cache Manager**
As the name suggests, this system exists to buffer the flow of events between the input and output systems. Audit events may come in bursts heavy enough to overwhelm a network connection. The cache manager can be built to perform in memory buffering using Go's buffered channels, or by writing to a local disc file for longer storage.
- **Rate Limiter**
To prevent the network connection from being overwhelmed, and to smooth out bursts of events, the rate limiter system introduces a forced delay between events sent to the server. This

system is built using Go's native timing interfaces that are already being used for thread management, and so are very efficient means of delaying execution.

- **Server Link**

This is the agent's output system. It is responsible for establishing a connection to the SKAFE server, ensuring a secure TLS connection if requested, and transmitting events in binary format over the network. If connection to the server is lost, it is the job of this system to continuously attempt to reestablish connection after a short timeout.

Server

The SKAFE server is composed of just four systems, all of which are always running.

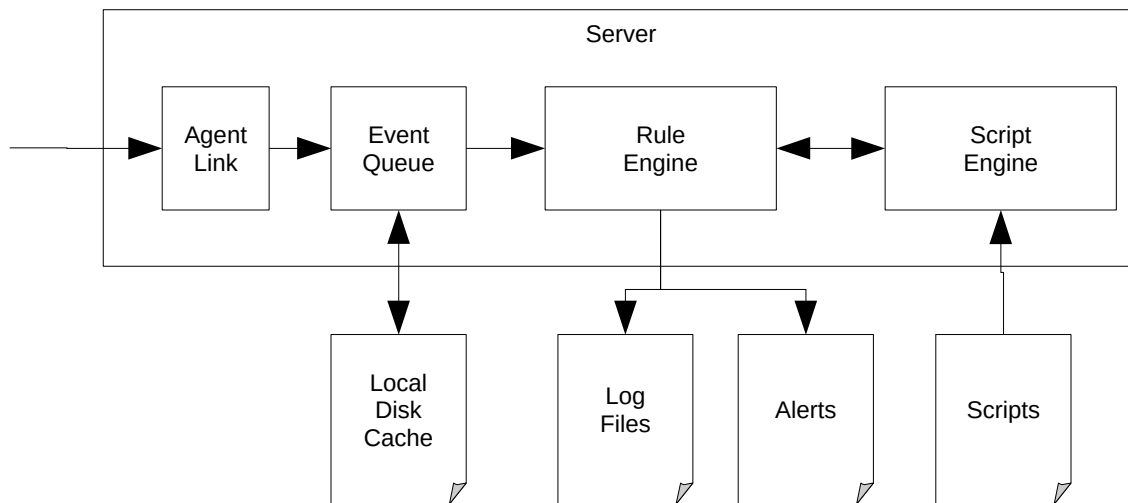


Illustration 4: Server systems diagram

- **Agent Link**

This system is the counterpart to the Agent's Server Link system. It has much the same responsibilities in ensuring an encrypted connection if requested, but with the addition of handling multiple client connections simultaneously.

- **Event Queue**

A near exact duplicate of the cache manager system on the Agent, the event queue system must act as a buffer between the potentially rapid stream of events arriving from a number of agents, and the relatively slow and intensive operation of the rule engine.

- **Rule Engine**

The rule tree is managed in this system. Here, multiple worker threads will run, retrieving events from the queue system and recursively running them through the decision tree. Events are passed of to the script engine when script rules are encountered.

- Script Engine

This system maintains the pool of script interpreter child processes. When an event is received from the rule engine system along with the script to run, the script engine selects an available interpreter and passes the event into processing. It retrieves the result when finished and passes it back to the rule engine.

The Rule Engine

The rule engine is the core of the entire SKAFE system, and merits more specific explanation unto itself. The rule engine is a set of software objects each representing a rule node. Together these nodes make up a single decision tree.

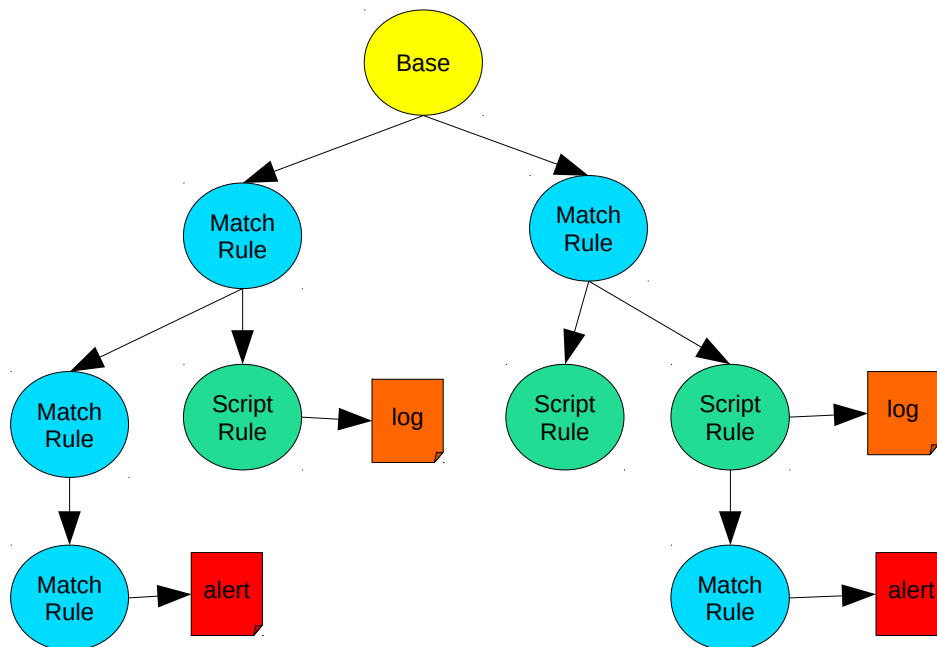


Illustration 5: An example decision tree

The decision tree forms the core of the rule engine. Each rule, either simple or script, makes a single node in the tree. Node may either subscribe to other nodes, or to the base event node. Nodes may also trigger either a log or an alert event, regardless if they are a leaf node in the tree or not. This means it is possible for a single event to trigger multiple alerts, but for different reasons.

An event enters the tree and follows a basic life cycle. Starting at the base node, it follows a DFS tree traversal. At each node, the rule or script is evaluated with the event. If the event matches the parameters, or if the script returns true, any log or alert events are triggered and the event continues down to any other nodes subscribed to that node.


```

1 // The recursive function that makes up each rule node
2 func RuleNode(event) {
3     // if the rule is a match rule
4     if ruletype == MATCH_RULE {
5         // for each specified regex
6         for match := range regexMatches {
7             // check if it applies to this event
8             if !match(event) {
9                 // if not, fail
10                return
11            }
12        }
13        // otherwise, if it's a script rule
14    } else if ruletype == SCRIPT_RULE {
15        // run the script
16        result := runScript(event)
17        // check if the script returns false
18        if ! result {
19            // fail if it does
20            return
21        }
22    }
23
24    // arriving here means the rule passed
25
26    // perform any desired log/events/alerts
27    executeTriggers(event)
28
29    // run the rules that are watching this rule
30    for node := range this.watchers {
31        RunNode(node)
32    }
33 }

```

Illustration 6: Golang style pseudocode for a rule node

The above Golang code shows a pseudo version of how each rule node works (for actual implementation, see ruleEngine.go). Each node checks if it's a match style node, or a script style node, and does the appropriate checks. For script nodes, this is simply to execute the script, and capture the boolean result. For match nodes, each specified match is individually checked, and if no single match fails, the node passes. After the checks any log events (log, alert, or both) are run, which creates the content of the log files produced by SKAFE.

It is critical that the nodes themselves remain stateless, as it is a requirement that multiple worker threads be able to run the same node at once. So the nodes themselves are very simple and contain no volatile data once they have been instantiated. The regular expressions used for matching rules are pre-compiled, and are then thread-safe. Script rules are more complex to maintain thread safety, but this was successfully handled with the separation of the script engine from the rule engine (refer to Illustration 4 above). All workers in the script engine have access via the same type of thread-safe buffered channel used in the rest of the application which connects them to the script engine. When a

script rule is needed to be run, the rule engine worker requests a script worker be made available for it. The script engine either selects an already running and idle worker, makes a new worker on the fly, or waits until a worker has finished and is returned to the idle pool and then delivers a handle to the worker to the rule engine worker. The rule engine worker can then use the script engine worker to perform whatever script action is required, retrieve the result, and then return the script engine worker to the idle pool for another to make use of.

To ensure logging to a file is also thread-safe, an additional layer of locking was added to the logger objects that wrap up the file handles. All rule engine workers share the same handles to these logger objects, so within each logger a mutex lock prevents itself from attempting to write to the file simultaneously. This is a potentially unnecessary precaution as the writing should be protected at the OS level at least to the size of the internal write buffer, but as this is not absolute the mutex lock was also implemented.

Match nodes suffer the slight drawback where they are only capable of doing logical AND conditions on the matches. That is to say if three matches are specified for a node, the requirement is implicitly that all the keys are present and they all match true. A single rule node cannot check for one match OR another match. However, this is overcome by the simplicity of creating multiple rule nodes, and the fact that rule nodes may have no effect directly, but only trigger another node. Instead of a single node checking for condition A OR B, two nodes will be created for A and B respectively, and then have the same outcomes as each other. However if this is still not sufficient, and a substantially more complex piece of logic must be implemented, the recommended solution is to simply place whatever logic is needed into a script and call that. A script may be used for all sorts of interaction, but it may also simply perform a slightly more complex matching rule. The rule engine will continue to work the same either way.

Operation Manual

Detailed here is the documentation needed to configure and run the application. This section is to be compiled into a series of man pages (see [Future Work](#)).

The agent and server are designed to fit into a number of systems, so init setup guides are not included.

Server

Compiling & Installing

Compiling the server requires the Golang compiler version 1.5 or higher. Lower versions may work but have not been tested. The server has only a few external dependencies, all of which are available with the project source code, or through the 'go get' tool to retrieve them from Golang's official repositories.

Once the source has been prepared, compile simply with the 'go build' command.

The resulting binary does not need to be in any particular location to run, and should be placed wherever the system administrator deems most suitable. Typically this will be /usr/bin/ or /usr/local/bin.

Configuration

All configuration files are in INI format, meaning options are set as key value pairs in the form “key = value”. Lines that start with a pound sign (“#”) are treated as comments.

The main server configuration file is located at /etc/skafe/skafe-server.conf, though this may be overridden with the config flag at runtime. This configuration file modifies all the behaviours of the server that are not related to rules or scripts.

Server Config		
<u>Key</u>	<u>Values</u>	<u>Description</u>
port	Int 1-65534	Specifies the port the server will listen on for new connections from agents
tls	True/false	Enables or disables TLS connections Defaults to false if not present
tlscert	File path	Absolute path for the server's TLS public cert Required if tls = true
tlskey	File path	Absolute path for the server's TLS private key Required if tls = true
tlsca	File path	Absolute path for a CA chain that is authoritative for signing certs from clients. If this key is present, server will strictly check client's signatures

Server Config		
Key	Values	Description
serverlog	File path stdout stderr syslog	Specifies where the server should log information about its internal operations, including client connects and debug info
eventlog	File Path stdout stderr syslog	Specifies where to log events that match a rule whose trigger is “log” or “both”
alertlog	File Path stdout stderr syslog	Specifies where to log events that match a rule whose trigger is “alert” or “both”
rulesdir	File Path	The absolute path of the folder which will contain the .rules files for the rules engine

```

1 # Configuration for the SKAFE server
2 #
3
4 # Listen port for the server
5 port = 6969
6
7 # Use TLS encryption [true|false]
8 #tls = true
9
10 # TLS cert and key pair to use
11 tlscert = demoserver.pem
12 tlskey = demoserver.key
13 tlsca = democa.pem
14
15 # internal server events are logged here
16 #serverlog = /var/log/skafe/skafe-server.log
17 serverlog = stdout
18
19 # events that trigger a log are logged here
20 #eventlog = /var/log/skafe/events.log
21 eventlog = stdout
22
23 # events that trigger an alert are logged here
24 #alertlog = /var/log/skafe/alerts.log
25 alertlog = stderr
26
27
28 # folder containing *.rules files
29 #rulesdir = /etc/skafe/rules
30 rulesdir = ./config/rules

```

Illustration 7: Example skafe-server.conf file

The next configuration file important to the server are the files containing rule information to make up the decision tree. Each rule requires a named heading, which in INI files is done with [square brackets], under which these options are available. Rules may also inherit from parent rules using the dot

notation. For example: rule [A.B] will implicitly have all the settings present in rule [A], but duplicates in [A.B] will override the previously specified values.

Rules Config		
<u>Key</u>	<u>Values</u>	<u>Description</u>
watch	Rule Name	The rule to subscribe to. If this rule should receive all events, set the watch to “base” If this key is not present, this rule will be skipped
action	match script	Specify whether this is a match rule or a script rule
trigger	log alert both	Specify if matching this rule should result in a log event, and alert event, or both
regextype	posix perl	The flavor of regex to use in match_xxx entries
match_xxx	Regular expression	Match events containing the key “xxx” who's value matches the provided regex ex: match_foo = ^bar\$ will match an event that contains event[“foo”] == “bar”
script	Script name	Specify which script should be run for this rule Required if action = script

```

1 [MatchRule]
2 action = match
3 regextype = posix
4 trigger = log
5
6 [ScriptRule]
7 action = script
8
9 [MatchRule.test]
10 watch = base
11 match_key = execve
12
13 [MatchRule.test2]
14 watch = base
15 match_key = file
16
17 [MatchRule.test3]
18 watch = MatchRule.test
19 match_key = exec[a-z]+
20
21 [MatchRule.test4]
22 watch = MatchRule.test
23 match_key = ex
24 match_data = llama
25
26 [catchall]
27 action = match
28 watch = base
29 trigger = log

```

Illustration 8: Example .rules config file

More than one rules file may be present in the server. To have a rule file loaded, it must be located in the folder specified in the server configuration file, and it must have a file name ending in '.rules'.

Running

The server does not explicitly require root access to run, and may be run as its own user or a lower privileged user for security. Whichever user the server is run as needs access to the configuration files and log files that will be used during operation.

As with any Golang program, flags may be prefaced with -single or --double dashes and do not require an equals (“=”) sign before the argument

Server Arguments	
Flag	Description
-config <file>	The location of the skafe-server config (default: /etc/skafe/skafe-server.conf)

Agent

Compiling & Installing

Compiling the agent requires the Golang compiler version 1.5 or higher. Lower versions may work but have not been tested. The agent has only a few external dependencies, all of which are available with the project source code, or through the 'go get' tool to retrieve them from Golang's official repositories.

Once the source has been prepared, compile simply with the 'go build' command.

The resulting binary does not need to be in any particular location to run, and should be placed wherever the system administrator deems most suitable. Typically this will be /usr/bin/ or /usr/local/bin.

Configuration

All configuration files are in INI format, meaning options are set as key value pairs in the form “key = value”. Lines that start with a pound sign (“#”) are treated as comments.

The agent has a single configuration file, by default located in /etc/skafe/skafe-agent.conf, but which can be overridden with command line argument at run time.

Agent Config		
Key	Values	Description
port	Int 1-65534	Specifies the port the client will attempt to reach the server on.
server	Host (ip or name)	Network address of the server Required
tls	True/false	Enables or disables TLS connections Defaults to false if not present
tlscert	File path	Absolute path for the agent's TLS public cert Required if tls = true
tlskey	File path	Absolute path for the agent's TLS private key Required if tls = true
tlscac	File path	Absolute path for a CA chain that is authoritative for signing certs from the server. If this key is present, server will strictly check server's signatures
log	File path stdout stderr syslog	Specifies where the agent should log information about its internal operations, including server connects and debug info

```
1 # correct config
2 server = skafe-server.local
3 port = 9999
4 log = stdout
5 []
```

Illustration 9: Example simple agent config

Running

The agent must be run as root for a number of its functions to work properly. Running as an unprivileged user will cause the agent to exit with an error message.

As with any Golang program, flags may be prefaced with -single or --double dashes and do not require an equals (“=”) sign before the argument

Agent Arguments	
Flag	Description
-config <file>	The location of the skafe agent config (default: /etc/skafe/skafe-agent.conf)
-auditor [true/false]	If enabled, set the agent to register itself as the system auditor and connect to the kernel to receive audit events (default: false)

F. Testing and Results

Test Design

Testing SKAFE was a challenge unto itself, but ultimately the program's structure helped substantially in breaking up the tests into manageable components.

Firstly, SKAFE overall was broken up into its two network components: the agent and the server. Each of these systems could be tested largely independently, with only a few tests to ensure proper communication was being established over the network.

Within each of the SKAFE components, the concurrency design allowed them to be further broken up into individual systems. These systems only ever communicated to one another via a single type of inter-system message, sent over Go's channel abstraction. For this reason, each system could be also tested in isolation, with tests confirming that the correct messages were sent and received sufficient to prove their ability to work with the other systems.

Finally, within each system the program was broken up into individual and independently operating functions to as much an extent as possible . At this level of isolation it was now possible to create automated unit tests for every possible input/output. These unit tests were set to run automatically for every new build, preventing any unexpected regression or bug introduction.

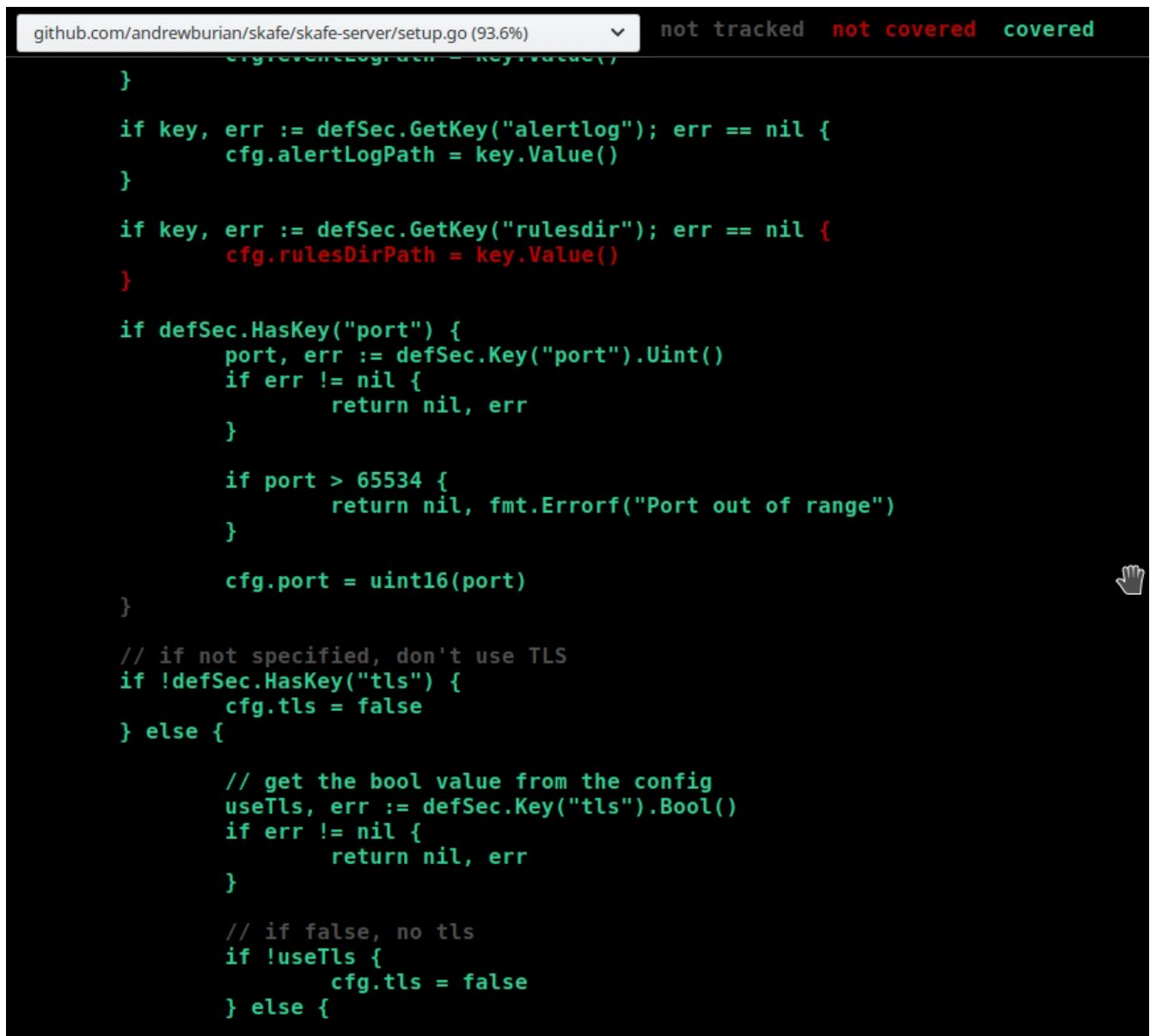
Every system in SKAFE was coded in its own file, and each system source file (ex: sys.go) had a corresponding unit test file (ex: sys_test.go) which explicitly was responsible for testing that particular system.

Testing Tools

One key advantage in the use of Golang for this project was access to Go's unit testing framework. This framework is extremely powerful, and tightly coupled to the language itself.

The testing tool 'go test' compiles the regular application, but additionally compiles any file with the suffix “_test.go”, files which are normally excluded from compilation, into the binary. This means any test code has direct access to code and systems that may have been hidden from external tests, such as anonymous functions, or private data.

Perhaps the single most useful feature though, is Go's ability to provide test coverage reports once tests are run. Golang compiles in runtime checks so that after the unit test suite has been run, the tool can report line for line which code paths were executed or not during the tests. This enables the developer to instantly spot edge cases that have not been covered by a unit test.



```
github.com/andrewburian/skafe/skafe-server/setup.go (93.6%)  not tracked  not covered  covered
}
    cfg.alertLogPath = key.Value()
}

    if key, err := defSec.GetKey("rulesdir"); err == nil {
        cfg.rulesDirPath = key.Value()
    }

    if defSec.HasKey("port") {
        port, err := defSec.Key("port").Uint()
        if err != nil {
            return nil, err
        }

        if port > 65534 {
            return nil, fmt.Errorf("Port out of range")
        }

        cfg.port = uint16(port)
    }

    // if not specified, don't use TLS
    if !defSec.HasKey("tls") {
        cfg.tls = false
    } else {

        // get the bool value from the config
        useTls, err := defSec.Key("tls").Bool()
        if err != nil {
            return nil, err
        }

        // if false, no tls
        if !useTls {
            cfg.tls = false
        } else {
```

Illustration 10: Example of test coverage output, with an intentional omission

During the development of the project and the tests, every system had its functions covered to the maximum extent possible with unit tests, and verified using this tool.

List of Unit Tests

This is the complete list of unit tests for both the agent and the server. All unit tests were passing at time of submission.

Agent Tests

Test Name	Component	System	Description
AuditParseValid	Agent	Audisp	A valid audit event is parsed into a corresponding data structure
AuditParseFull	Agent	Audisp	A full audit event included <i>auditd</i> header still parses
AuditParteFullNewline	Agent	Audisp	A full audit event terminated with a newline still parses
AudispValid	Agent	Audisp	Audisp system returns the correct number of valid events from data input
Username	Agent	Enrichment	Username is resolved from uid
SourceUser	Agent	Enrichment	Source user is resolved from the text uid entry in the event
FullCmd	Agent	Enrichment	The full exec command is reconstructed from its components in the event
FullCmdExtra	Agent	Enrichment	Extra information in the command is ignored
RateLimit	Agent	Ratelimit	The rate-limit system operates within an acceptable error range (5ms)
Delay1	Agent	Ratelimit	Delay is calculated correctly from messages/second
Delay2	Agent	Ratelimit	Further iteration
Delay3	Agent	Ratelimit	Further iteration
Config	Agent	Setup	A valid config file is parsed correctly
NoConfig	Agent	Setup	A missing config file errors
NoAddr	Agent	Setup	Missing server entry errors
BadPort	Agent	Setup	Port that's not a number errors
HighPort	Agent	Setup	Port above 65535 errors
LogFileBroken	Agent	Setup	Target log file cannot be created errors

```
[Arch skafe-agent]# go test
PASS
ok      github.com/andrewburian/skafe/skafe-agent    0.207s
```

Illustration 11: Agent tests passing

Server Tests

Test Name	Component	System	Description
ClientRejectNoTLS	Server	Link	Reject clients connecting plaintext when TLS is enabled
ClientRejectUntrusted	Server	Link	Reject clients that aren't signed by the correct CA
ClientAcceptTrusted	Server	Link	Accept when all parameters are correct
NewScriptWorker	Server	ScriptWork	Creating a new script worker with the correct bin
NewRbWorkerBadBin	Server	ScriptWork	Creating a new script worker with an invalid bin path throws an error
MatchNodeBlankEvent	Server	RuleEngine	A blank match event matches to any event
NoMatchMissingKey	Server	RuleEngine	A match where the event is missing a required key does not match
MatchNodeBlankBoth	Server	RuleEngine	A blank event that triggers a log and an alert will do so
MatchNodeBlankAlert	Server	RuleEngine	A blank event that triggers only alerts will do so
MatchNodeHalfMatch	Server	RuleEngine	Matching only some parameters does not trigger
RecursiveMatch	Server	RuleEngine	An event will recursively trigger subscribed events
CreateMatchRuleGoodSingleEvent	Server	RuleEngine	Create a match rule that only triggers event logs
CreateMatchRuleGoodSingleAlert	Server	RuleEngine	Create a match rule that only triggers alert logs
CreateMatchRuleGoodSingleBoth	Server	RuleEngine	Create a match rule that triggers both alert and event logs
CreateMatchRuleBadTrigger	Server	RuleEngine	A match rule with an invalid trigger throws an error
CreateMatchRuleBadRegex	Server	RuleEngine	A match rule with an invalid regex throws an error
CreateMatchRuleBadRegexType	Server	RuleEngine	A match rule with a regex whose type is not posix or perl throws an error
RulesConfInvalidDir	Server	RuleEngine	Specifying a rules directory that doesn't exist throws an error
RulesConfCountRules	Server	RuleEngine	The correct number of rules are created
RulesConfIgnoreOther	Server	RuleEngine	Superfluous data in the rules config does not cause problems
RulesConfPermissionError	Server	RuleEngine	Failure to open a rules file due to permissions causes an error

Test Name	Component	System	Description
CreateRuleSkipDefault	Server	RuleEngine	The default conf section is not treated as a rule
CreateRuleSkipNoWatch	Server	RuleEngine	A rule with no watch is skipped but creates a log event
CreateRuleSkipNoParent	Server	RuleEngine	A rule with no parent conf section is skipped but creates a log event
CreateRuleSkipNoAction	Server	RuleEngine	A rule that is not a match or script is skipped but creates a log event
CreateRuleBadAction	Server	RuleEngine	A rule with an invalid action causes an error
CreateRuleBadTrigger	Server	RuleEngine	A rule with an invalid trigger causes an error
CreateRuleMatch	Server	RuleEngine	A valid config creates the correct match rule
CreateRuleScript	Server	RuleEngine	A valid config creates the correct script rule
SetRuleTreePass	Server	RuleEngine	A complete correct conf parses correctly
SetupRuleTreeFail	Server	RuleEngine	Any error results in the parsing to fail
SetupScriptPoolRuby	Server	ScriptEng	A script pool is created with a ruby interpreter
GetWorkerWrongLang	Server	ScriptEng	A request for a nonexistent interpreter fails
TLSConfig	Server	Setup	A correct TLS section it parsed properly
TLSConfigMissingCert	Server	Setup	A missing TLS cert throws an error
TLSConfigMissingKey	Server	Setup	A missing TLS key throws an error
TLSSetupGood	Server	Setup	A complete TLS parse results in successful setup
TLSSetupGoodWithCa	Server	Setup	With a CA set, strict checking is enabled
TLSSetupBadCA	Server	Setup	Invalid CA cert throws an error
TLSSetupBadCert	Server	Setup	Invalid server cert throws an error
TLSSetupBadKey	Server	Setup	Invalid server key throws an error
FileLiggerSetupGood	Server	Setup	Valid file loggers are created successfully
FileLoggerSetupFailed	Server	Setup	Invalid file paths throw an error
StdioLoggerNonsense	Server	Setup	Stdio logger not stdout or stderr throws error
SetupLoggersFailServer	Server	Setup	Invalid server log throws an error
SetupLoggersFailEvent	Server	Setup	Invalid event log throws an error
SetupLoggersFailAlert	Server	Setup	Invalid alert log throws an error
SetupConfigFail	Server	Setup	Invalid loggers config section throws an error
TlsConfigNonbool	Server	Setup	Tls entry not true or false throws an error
TlsDisable	Server	Setup	TLS to false results in no TLS setup occurring
TlsSetupWithDisabled	Server	Setup	Config parses correctly with TLS disabled

Test Name	Component	System	Description
StdoutLoggers	Server	Setup	Stdout logger created correctly
StderrLoggers	Server	Setup	Stderr logger created correctly
SysLoggers	Server	Setup	Syslog logger created correctly
FileLoggers	Server	Setup	Log file logger created correctly
ListenPortDefault	Server	Setup	No listen port in conf results is default port
ListenPortSet	Server	Setup	Listen port in conf set correctly
ListenPortTooHigh	Server	Setup	Port greater than 65534 throws an error
ListenPortNan	Server	Setup	Port not a number throws an error

```

--- RUN TestStdioLoggerNonsense
--- PASS: TestStdioLoggerNonsense (0.00s)
=== RUN TestSetupLoggersFailServer
--- PASS: TestSetupLoggersFailServer (0.00s)
=== RUN TestSetupLoggersFailEvent
--- PASS: TestSetupLoggersFailEvent (0.00s)
=== RUN TestSetupLoggersFailAlert
--- PASS: TestSetupLoggersFailAlert (0.00s)
=== RUN TestSetupConfigFail
--- PASS: TestSetupConfigFail (0.00s)
=== RUN TestTlsConfigNonbool
--- PASS: TestTlsConfigNonbool (0.00s)
=== RUN TestTlsDisable
--- PASS: TestTlsDisable (0.00s)
=== RUN TestTlsSetupWithDisabled
--- PASS: TestTlsSetupWithDisabled (0.00s)
=== RUN TestStdoutLoggers
--- PASS: TestStdoutLoggers (0.00s)
=== RUN TestStderrLoggers
--- PASS: TestStderrLoggers (0.00s)
=== RUN TestSysLoggers
--- PASS: TestSysLoggers (0.00s)
=== RUN TestFileLoggers
--- PASS: TestFileLoggers (0.00s)
=== RUN TestListenPortDefault
--- PASS: TestListenPortDefault (0.00s)
=== RUN TestListenPortSet
--- PASS: TestListenPortSet (0.00s)
=== RUN TestListenPortTooHigh
--- PASS: TestListenPortTooHigh (0.00s)
=== RUN TestListenPortNan
--- PASS: TestListenPortNan (0.00s)
PASS
ok      github.com/andrewburian/skafe/skafe-server 0.043s

```

Illustration 12: Server unit tests passing

System Test

For the complete system test, or the end to end test, a simple proof of concept was devised. An audit event was captured from a live system containing some notable forensics, in this case a binary being executed by uid 0, and saved to a file.

```
[Arch skafe-agent]# cat sampleaudit.log
audit(1460751363.327:9): pid=8433 uid=0 ses=5 res=1 bin=/bin/tail argc=2 a0=tail a1=/etc/passwd
[Arch skafe-agent]# █
```

Illustration 13: Captured sample data

The SKAFE agent was then run in *auditd* mode, meaning it will read events from its standard input, and this audit event piped into it.

```
[Arch skafe-agent]# tail -f sampleaudit.log | ./skafe-agent -conf config/skafe-agent.conf
2016/04/21 19:32:45 Started skafe-agent
█
```

Illustration 14: Agent started but not connected

The SKAFE agent will immediately do the processing to resolve the uid to the username “root”, and note the path of the executed binary, and prepare the event for shipping. Meanwhile, since the startup of the agent, it will have been attempting to establish a connection to the server, which is not yet running.

The SKAFE server will then be started, and will load a few rules to test. The first is a simple match rule that will trigger on any event, confirming that the event was received by the server. The next is another match rule that triggers when the username is “root”. This will confirm the event was enriched with this information, and that the information persisted onto the server. Finally, a script rule will run that returns true when the binary “/bin/tail” is executed.

```

1 # Test rules for demo purposes
2
3 # Rule that will trigger on any event received
4 [catchall]
5 watch = base
6 action = match
7 trigger = log
8
9 # Check for things executed by root
10 [rootactivity]
11 watch = base
12 action = match
13 trigger = log
14 match_uname = root
15
16 # Run the script which checks for tail
17 [tailsript]
18 watch = base
19 action = script
20 trigger = log
21 script = tailcheck
22 []

```

Illustration 15: Rules loaded onto the server

When the server starts, we note the connection is established from the agent within the timeout period the agent has for connection retries, which is 10 seconds. Almost instantly later, we see three log events trigger from the three rules present on the server.

```

[Arch skafe-agent]# tail -f sampleaudit.log | ./skafe-agent -conf config/skafe-agent.conf
2016/04/21 19:32:45 Started skafe-agent
2016/04/21 19:34:25 Connected to skafe server

```

Illustration 16: Agent reporting connection established

```

[aburian@sengard skafe-server (server *+*)]$ ./skafe-server -conf config/skafe-server.conf
2016/04/21 12:34:20 SKAFE Server started!
2016/04/21 12:34:20 Loading rules file ./config/rules/default.rules
2016/04/21 12:34:20 Loading rules file ./config/rules/test.rules
2016/04/21 12:34:20 Rule [MatchRule] has no watch, skipping
2016/04/21 12:34:20 Rule [ScriptRule] has no watch, skipping
2016/04/21 12:34:20 Rule Engine started
2016/04/21 12:34:25 Connection accepted from 10.0.1.99:35736
2016/04/21 12:34:25 [catchall] - map[uname:root uid:0 a1:/etc/passwd pid:8433 res:1 a0:tail argc:2 cmd:tail /etc/passwd ses:5 bin:/bin/tail]
2016/04/21 12:34:25 [rootactivity] - map[bin:/bin/tail argc:2 cmd:tail /etc/passwd ses:5 a0:tail uname:root uid:0 a1:/etc/passwd pid:8433 res:1]
2016/04/21 12:34:25 [tailsript] - map[cmd:tail /etc/passwd ses:5 bin:/bin/tail argc:2 pid:8433 res:1 a0:tail uname:root uid:0 a1:/etc/passwd]

```

Illustration 17: Server start and events triggered

This verifies that all of the core functions delivered by SKAFE are working as expected.

G. Implementation Details

The design of SKAFE was carefully done to meet a number of objectives. These decisions shaped the way the application behaves, the speed in which it performs, and its ability to be extended and developed further in the future.

For Testing

One of the first decisions made in the development process was that testing would heavily take advantage of Golang's unit testing framework. This will allow automated testing, test coverage analysis, and help reduce and eliminate bugs all by encouraging test driven development.

To structure the code in such a way that unit tests would be most effective, a functional approach was taken, and code was divided up into as many discrete pieces as possible. The goal was to have the entire application consist mainly of independent functions, where the function would not internally call other parts of the application, and known inputs could be mapped easily to expected outputs without having to fake or shim external dependencies.

SKAFE's code is written this way, and in only a very few cases does a function call another portion of the code. Each system consists of a single function, that only ever calls unit tested functions, and pass returned values into other functions. So long as these primary system functions aren't doing anything that should need testing themselves, the unit test code will cover most cases.

Another design choice was to have the only communication between systems be through Go's buffered channels, and only use one common message object type. This allowed each system to also become further independent of each other, and allow unit tests for entire systems to input particular objects, and expect well defined outputs in return.

For Performance

One of the key goals and challenges of SKAFE was to have it perform such that it could be deployed in a large production environment, and be able to keep pace with the number of events such a situation would generate.

Concurrency was a key motivator in the decision to use individual systems within both the agent and the server. Each system is run in its own thread, allowing events to be pipelined through the application.

The slowest part of each SKAFE system, notably the server link on the agent and the rule engine on the server, were anticipated to be the bottlenecks of this pipeline system. This prompted the additional design choice to add an event queue or buffer system in the pipeline before each of the bottleneck systems, allowing the other systems to perform uninhibited by the slowdown.

Furthermore, the most complex and time consuming component of the entire application, the server's rule engine, required additional parallelism to be used most effectively. Where the set of systems leading up to the rule engine are a pipeline, the engine itself is structured as a producer-consumer problem. The queue stage prior presents the events to be processed, then a number of rule engine workers, running the same code, queue for the next available event and process it in its entirety before returning for another event.

For Flexibility

Another goal of SKAFE was to be able to fit into any operational situation, which means the entire application needs to have a high degree of configuration flexibility. It was decided that the easiest way to accomplish this was through the development of the setup system.

The setup system defines a single object, and runs only once in the lifetime of the application. At startup, setup parses the specified configuration file, and then from this configuration, sets every path, flag, and variable in an options object. This options object is then passed to every other system to affect their behaviours.

A version of this setup system is present in both the agent and the server application, and differs only in which configuration entries it will parse for and what members exist in the options object.

For Future Development

Lastly, it was important to realize from the start that SKAFE would undoubtedly need both minor and major updates in the future. In the duration of this project, it is simply not possible to cover all the possible cases that SKAFE might be subjected to in a real life deployment situation. So it was important that the code be developed in such a way that future maintenance could happen smoothly and easily, without needed to backtrack across large swaths of the code base.

In conjunction with the design choices that were made for testing, this requirement lead to the use of very isolated functions, and a strictly organized code base. The agent and the server's code bases were kept distinct, as there was not enough overlap to merit combining the two. Within each of the applications respectively, each system was coded entirely in its own source file. Each source file also had a matching tests file containing its unit test code.

For the agent enrichment, the simple function design allows for any single piece of the enrichment process to be isolated, and then updated or removed as desired. Adding new enrichment items, such as perhaps a different hash, are also simple, and require writing only the new function, and then adding a call in one single place.

The scripting engine itself was also made with heavy use of Go's generic classes. While the current implementation supports only Ruby script interpreters, an entirely new script language could be added to the engine, and immediately be available to the core of the application with no modification at all.

This is to ensure that if Ruby proves to not be sufficient, or even simply more flexibility is desired, the rule engine can run any number of different script engines at the same time, and rules may call scripts in any language seamlessly. The coder need only write a small script shim, and a wrapper for the generic script worker object in the application code itself.

H. Research Done

This section briefly summarizes the additional research that was done in the design process for SKAFE. This research does not include research into other projects and solutions, which are already covered in the [Other Solutions](#) section.

Kernel Auditing

As a kernel auditing system, detailed knowledge of how the kernel actually generates events, interfaces with the user space auditor, and the format of the messages all needed to be researched. Particularly important was how to combine the audit messages from the same event into one solitary event for processing.

Concurrent Systems

In addition to knowledge already obtained from the degree program at BCIT, it was necessary to further understand concurrent design for systems that may scale during the lifetime of the application, and vary system to system. Essentially, the SKAFE server needed to be designed to make maximum use of whatever resources it was given, so that should it be deployed in a production environment and run at it's capacity, increasing the resources available to it should result in a comparable increase in performance.

Golang Features

While not unfamiliar with Golang prior to this project, some of the underlying implementations of Golang's features needed to be understood in greater detail to make critical design choices.

Golang has its own garbage collection system and memory management, and it was critical to understand the performance implications of various methods of coding as having the garbage collector run unnecessarily would have had a notable impact.

Golang also runs its threads using a managed green-thread design rather than system threads. Understanding this system allowed for a design that could make use of more threads than the system has cores without the penalty incurred with kernel context switching and scheduling.

For inter thread and, in SKAFE's case, inter-system communication research was done into Golang's primitive object known as a buffered channel. These thread-safe shared memory pipes are possibly the single most efficient system for sharing data within the application. They proved a perfect solution for the producer-consumer problem in the rules engine, worked flawlessly for exchanging messages between systems, and the buffering made the queue systems almost obsolete.

Linux Daemons and Systems Design

Determining how SKAFE would be deployed in practice meant researching the entire system it would fit into. In this case, that meant learning the Linux daemon architecture in its entirety.

Firstly, every attempt was made to respect the Linux Filesystem Hierarchy Standard in the placement of the configuration files, binaries, logs, and library files. This results in SKAFE being spread across four different file system locations, and necessitating the creation of the makefile to ensure everything would be installed correctly.

Determining how daemons are started proved to be another challenge. Depending on the init system in use, there are upwards of four different ways that the agent and servers could potentially be started in common Linux systems. In addition to that, it could also be started by *auditd* and be independent of any init system altogether. This led to careful consideration of default values, and led overall to the decision to put more emphasis on configuration files than on command line arguments, as some systems would have had substantial difficulty with setting those consistently.

Daemon logging and reporting systems were also researched, and considerations such as syslog were added. Syslog is explained in more detail in the [Alternative Solutions](#) section.

I. Future Work

Like most software solutions, SKAFE could have a theoretically endless development life with new features and functionality added in endless cycles. However, SKAFE was designed specifically with some future advancements in mind, and the direction of those works is laid out here.

Documentation

One of the extra features that didn't make it in to the 400 hour time line of this project was the creation of man pages for the applications and configuration files that are part of SKAFE. This is the list of man entries that should be made to accompany a SKAFE install:

- skafe (1) – General overview of the SKAFE infrastructure and references to the over man pages
- skafe-server (1) – Server arguments list and explanation of the rule engine in a high level
- skafe-agent (1) – Agent arguments list and explanation of the agent's operations
- skafe-server.conf (5) – List of configuration entries allowable in the server config file
- skafe-agent.conf (5) – List of configuration entries allowable in the agent config file
- skafe.rules (5) – Documentation on how the rules files are structured, and the allowable entries
- skafe.scripts (5) – Documentation on how the user scripts must be constructed to work correctly

Installer Package

To streamline the process of installing SKAFE and all its dependencies and configuration files in the correct locations, it would be extremely helpful to create package files for a number of different Linux distributions. This was kept in mind somewhat during its design, and despite the fact that Golang has no explicit need for makefiles SKAFE has one anyways that includes an install command for creating and copying all the necessary resources to the right places. Now any installer package that can be built from a makefile will have a far easier time being constructed for SKAFE.

Script Languages

During the design process, some experts in security systems were consulted to see if there was a consensus on what the best script language to use for SKAFE's scripting engine would be. There was none. Consequently, SKAFE's script engine was written with Golang's object generics so that it could easily be wrapped around any script interpreter. The trial implementation was written in Ruby, but more languages should be added to allow the application to be readily usable for any developer.

Script Builtin Features

The surrounding code infrastructure is already in place to allow the script engines to make requests back into the SKAFE server, but no features have yet been implemented which take advantage of this. Originally proposed was the ability for a script to request copies of files, but this can be extended to any action that the agent is capable of performing and the results of which can be sent back over the same network link to the server.

Features that can be created to take advantage of this facility will likely be SKAFE's avenue of growth for the foreseeable future. As more use cases for SKAFE emerge, more features in this area will be needed or wanted to continue to make the application more effective and useful.

Automated Integration Testing

Currently, unit testing is the core of SKAFE's quality assurance and quality control processes. While this lets each system be tested extremely thoroughly in isolation, it does less than hoped for in terms of assisting integration testing. End to end tests still mostly need to be performed manually, and are harder to write in such a way that most scenarios and use cases are covered. SKAFE's testing framework should be expanded to include automated integration testing.

Performance Benchmarks

Golang's testing suite does actually support benchmarks out of the box, but no current tests take advantage of this. The manual end to end test currently are sufficient to verify that SKAFE has met its performance goals, but this should be further refined to measure the performance of individual systems. With this information, performance tuning measures can be taken in the future to accelerate specifically the portions of the application or of specific systems that need it.

J. Timeline

Development took slightly in excess of the anticipated 400h, with several features taking substantially longer than expected to develop and test. As a result, some features of the 4th stage of work were postponed to future work, and some other features were reduced in complexity. All critical deliverables were met, and even the features that were reduced had their code base established and unit tests created so that future work could finish them more completely.

This section was taken verbatim from the original project proposal, but the time column was split into the original expected time, along with the actual time taken for each section.

The total times for each stage, along with the time taken to develop the unit tests, and an overall breakdown of time spent, is available at the end of this section.

Stage Breakdown

Stage 0 – App Core

Stage 0 is the basic audit event shipping framework. An agent will be constructed that can receive audit events from the kernel, and a server created to receive them.

#	Name	Description	Exp	Act
0-1	Kernel NETLINK	Connect to the kernel and request audit messages. Receive and parse responses	32h	56h
0-2	Event Enrichment	Add useful information to event in user-space uid => user name, pid => process name, parent processes, binary names, hashes	8h	8h
0-3	Even Shipping	Create binary protocol for sending event data over the network. Create packing/unpacking functions	16h	16h
0-4	Server-side enrichment	Server adds more useful information such as agent id/name, and dns lookups	8h	-
0-5	Server Logging	Server logs all events to a local file	8h	8h

Stage 1 – Basic Rule Engine

Stage 1 introduces the rule engine in its most basic form. Simple matching rules create a decision tree, with alert or log rules also allowed. Focus is on performance.

#	Name	Description	Exp	Act
1-1	Config file format	Create text config file format that can be edited easily by users, and parsed to create decision tree	8h	16h

		nodes		
1-2	Create tree structure	Parse config files into software object and create the decision tree object.	16h	16h
1-3	Engine Workers	Create a thread-safe worker paradigm to allow the rule engine to be fully multi-threaded, and able to expand dynamically to the load.	32h	32h
1-4	Logging/Alerts	Build in the option for any node in the tree to also log the event in question, or trigger an alert.	8h	8h

Stage 2 – Application Durability

This stage will add all the features necessary to consider the app production-safe. Agents will be able to handle disconnects from the server smoothly, authenticate the connection, and not lose events.

#	Name	Description	Exp	Act
2-1	TLS Connection	Upgrade the link between the client and server to an authenticated TLS connection	8h	16h
2-2	Agent Caching	Agents will be able to cache events locally to disk in the event the server becomes unavailable, up to a limit	24h	32h
2-3	Server rate-limiting	Expand event communication protocol to allow the server to rate-limit the events being sent by clients, to avoid overburdening.	16h	-
2-4	Agent graceful death	Agents should be able to recover from any non-fatal error or OS signals without lose of events	4h	8h
2-5	Server graceful death	Server should be able to recover from non-fatal errors or OS signals without lose of events	4h	16h
2-6	Agent status events	Agents submit SKAFE events when they come online or are terminated for the record	16h	-
2-7	Server reload	SIGHUP will cause the server to gracefully reload	8h	24h

Stage 3 – Script Rules

This stage marks the primary goal for this application. Having completed this stage, the application will be fully functional. Rule scripts will be added to the rule engine, allowing complete freedom of analysis.

#	Name	Description	Exp	Act
3-1	Script rule in config	Add the option for a script rule to the config file	4h	8h

		standard		
3-2	Concurrency model	Design and develop the script engine to not slow the system on long operations.	32h	32h
3-3	Script worker pool	Implement Ruby interpreter pool	16h	24h
3-4	App-script protocol	Design the system for passing events and results back and forth between the application and the script interpreter pool	24h	32h

Overall Timeline

The project time-line concludes 8 hour working days as a general average.

Project Section	Expected hours	Actual hours
Preliminary Design	8 (1d)	24 (3d)
Stage 0 Development	72 (9d)	88h (11d)
Stage 0 Testing	8 (1d)	24h (3d)
Stage 1 Development	64 (8d)	72h (9d)
Stage 1 Testing	8 (1d)	8h (1d)
Stage 2 Development	80 (10d)	80 (10d)
Stage 2 Testing	8 (1d)	16 (2d)
Stage 3 Development	112 (14d)	112 (14d)
Stage 3 Testing	16 (2d)	24 (3d)
Stage 4 Development	16 (2d)	-
Stage 4 Testing	8 (1d)	-
TOTAL	400h (50d)	448 (56d)

Conclusion

The results of this project is the beginnings of a very useful security auditing tool more versatile than any solution currently available for Linux endpoints. From here, following the development plan listed in Future Work, SKAFE can grow from its current state as a functional and fast audit tool into a fully fledged, feature rich, and production ready tool to be added to the arsenal of security systems everywhere.

SKAFE will fill a niche in larger security systems, specifically the need for highly customized and site specific automation of Linux kernel auditing, something which is currently done poorly, if at all, by current products. SKAFE's design will allow it to scale in pace with the resources available to it, allowing it to remain useful in progressively larger operations without having to revisit the underlying design choices made. It's script flexibility, including the option to add new script languages to it, removes a language-specific technical barrier for its use which will assist in easy and wider spread adoption into current systems.

Personally, the design and development of SKAFE has given me a fantastic opportunity push the limits of my technical knowledge and skills. The research into current solutions has given me a better feel for the landscape of security tools that exist today, and how to best plan and work with them to maximum effect in a variety of scenarios. Everything I have learned to date about concurrent systems design and high performance applications was brought to bare in trying to make SKAFE perform to the level I felt it needed to to be successful in its purpose.

This project also substantially added to my knowledge and comfort with automated test based development practices. It was clear to me that if anyone was going to seriously place trust in this tool on real life systems, that trust was going to need to be built on more than my assurances, but rather on indisputable proof of design completeness, code quality, and bug prevention efforts. Deciding on a test driven design allowed me to accomplish more of these goals while also making it far easier for other open source developers to jump in on the project and for me to have a reliable and automated way to ensure the quality of their work matches the standards I set for myself.

SKAFE's future from here will be undoubtedly interesting. Lots of development remains before I will consider it completed. Currently, the code is hosted on Github and is open source for all to see and use under the GNU2 licence. I intend to maintain this, inviting contributions and development input from the community, and hopefully inspire some attempts to make use of the project both in the way I built it for, and in ways I could never have thought of.

Appendix

List of Terms and Program Names

- **auditd**
This can refer to one of two things. It can refer to the audit daemon program itself, which is the binary responsible for interfacing with the kernel, or it may refer to the entire project which includes audisp, auparse, aureport, and auditctl.
- **audisp**
The multiplexer daemon for auditd
- **syslog**
The original system logging daemon on Linux systems.
- **rsyslog**
The second generation of logging daemon. The 'r' prefix standing for “rocket fast”
- **syslog-ng**
The third generation of logging daemon. The “ng” suffix standing for “next generation”
- **systemd**
An init daemon responsible for controlling and starting all of userspace after the kernel has been set up on boot.
- **systemd-journald**
The logging daemon that comes with systemd. Called the journal, this daemon acts both as syslog, and performs other functions related to capturing output from daemon processes.