BACHELOROPPGAVE:

IDSanity

A centralized framework for managing IDPS and WAF rulesets.

FORFATTERE:
Tommy Berge Ingdal
Halvor Mydske Thoresen
Victor Ruldolfsson

DATO:
15.05.2015

Sammendrag av Bacheloroppgaven

Tittel:	IDSanity	Nr: -
	Et sentralisert og modulært rammeverk for å håndtere IDPS og WAF-regelsett.	Dato: 15.05.2015
Deltakere:	Tommy Berge Ingdal	
	Halvor Mydske Thoresen	
	Victor Ruldolfsson	
Veiledere:	Stewart Kowalski	
	Thomas Kemmerich	
Oppdragsgiver:	IT-Tjenesten, Høgskolen i Gjøvik	
Kontaktperson:	Christoffer Hallstensen, christoffer.hallstensen@hig.no	
Stikkord	IDPS, WAF, Rammeverk, Python	
Antall sider: 116	Antall vedlegg: 5	Tilgjengelighet: Åpen

Kort beskrivelse av bacheloroppgaven:
I et stort nettverk med flere IDPS og/eller WAF-noder kan det fort bli tidkrevende og komplisert å vedlikeholde og holde styr på hvilke regelsett som er aktive til enhver tid. Dette kan potensielt føre til at man benytter seg av utdaterte regelsett og i tillegg bruker altfor mye tid på vedlikehold.

IDSanity forsøker å løse dette ved å la systemadministratorer vedlikeholde alle noder ved hjelp av ett felles verktøy - enten via CLI eller det medfølgende web-grenesnittet. Applikasjonen bygger på klient-server-modellen hvor man har én server og én eller flere klienter.
IDSanity startes i slave mode på hver node og kommuniserer direkte med Master som kjører på en separat server.

Applikasjonen er i all hovedsak utviklet for å kjøre på Debian/Linux, men er såpass modulær at moduler kan utvikles for andre platformer om nødvendig.
I tillegg til å være et verktøy for å skape en god oversikt over aktive regelsett så fungerer også IDSanity som et fullstendig vedlikeholdsverktøy for IDPS/WAFs.

Summary of Graduate Project

Title:	IDSanity	Nr: -
	A centralized and modular framework for managing	Date: 15.05.2015
	IDPS and WAF rulesets.	
Participants:	Tommy Berge Ingdal	
	Halvor Mydske Thoresen	
	Victor Ruldolfsson	
Supervisor:	Stewart Kowalski	
	Thomas Kemmerich	
Employer:	IT-Tjenesten, Høgskolen i Gjøvik	
Contact person:	Christoffer Hallstensen, christoffer.hallstensen@hig.no	
Keywords	IDPS, WAF, Framework, Python	

Pages: 116	Appendixes: 5	Availability: Open

Short description of the main project:
In a big and complex network of IDPS and/or WAF nodes it may become time-consuming and complicated to maintain and keep control of all the different rule sets which are active at a given time. This can potentially lead to the use of out-dated rule sets and that the administrators spend way to much time on maintenance.

IDSanity aims to solve this problem by giving system administrators a way to maintain all nodes in a network with one easy-to-use tool, either through the command-line or web interface. The application makes use of a client-server model where you have one server and one or more clients.
IDSanity starts in slave mode on each node in the network and communicates directly with the Master which are running on a separate server.

The application was mainly developed to run on Debian/Linux, but since IDSanity is module based it's really easy to develop new modules that can be run on different operating systems.
In addition to give the administrators a good overview of the active rule sets on each node, IDSanity functions as a complete maintenance tool for IDPS/WAFs.

IDSANITY

Preface

IDSanity was developed by three students at Gjøvik University College during the spring of 2015 as a Bachelor's Thesis for The IT Departement at Gjøvik University College.
Since this project mainly was aimed at Information Security students we felt that this project was a good opportunity to learn more about Intrusion Detection And Prevention Systems, Web Applications Firewalls, as well as a programming language we previously had minimal experience with.

We feel that this has been a good experience for all of us and that we have learned many new things during this project period.

We would like to thank the IT Departement at Gjøvik University College for the opportunity to work on this project, and specially *Christoffer Hallstensen* for the good dialog during the project meetings.
We would also like to thank *Stewart Kowalski* and *Thomas Kemmerich* for positive and constructive feedback early in the project period.

Other people we would like to thank are:

- Niklas Lindén for help with designing the logo.

Gjøvik, 15.05.2015

Tommy B. Ingdal	[illegible]	Victor Rudolfsson
Tommy B. Ingdal	Halvor M. Thoresen	Victor Rudolfsson

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface.
CLI	Command-Line Interface.
GUI	Graphical user interface.
HIDS	Host-Based Intrusion Detection System.
HMAC	Hash Message Authentication Code.
HTTP	Hypertext Transfer Protocol.
IDS	Intrusion Detection System.
IDPS	Intrusion-Detection And Prevention System.
IPS	Intrusion-Prevention System.
JSON	JavaScript Object Notation.
LDAP	Lightweight Directory Access Protocol.
MITM	Man-in-the-middle.
NIDS	Network-Based Intrusion Detection System.
NIST	National Institute of Standards and Technology.
OSSEC	Open Source Security.
PKI	Public Key Infrastructure.
SIEM	Security Information And Event Management.
SSL/TLS	Secure Sockets Layer/Transport Layer Security. itemSQLi SQL Injection.
WAF	Web Application Firewall.
XML	EXtensible Markup Language.

Glossary

Agent In the context of IDPS an agent is an application installed on a host computer reporting system status to a Master or management server.

Debian Unix-like computer operating system and Linux distribution.

Flask Micro web application framework for Python.

Git Distributed revision control system with an emphasis on speed and data integrity.

Github A web-based Git repository hosting service, which offers revision control and source code management.

IPS/Inline The Intrusion Prevention System is placed in the direct communication path between the source and destination.

Master The master is a Node with the main server application installed, that communicates and controls the Slaves.

MITM Attack Is an attack where the attacker secretly relays and maybe alters the communication between source and destination.

Node A node is an entity in the system. A node can either be a Master, or a Slave.

Open Source Refers to a program in which the source code is available to the general public for use and/or modification free of charge.

OSSEC Open source host-based intrusion detection system.

PEP8 Style Guide for Python Source Code.

Python High-level programming language.

PostgreSQL Object-releational database management system.

RESTful Software architecture style for designing networked applications.

RESTless Software that does not adhere to RESTful principles.

Slave The Slave is a Node with the Slave application installed, that communicates with the Master.

Snort Open source network intrusion prevention system and network intrusion detection system.

SSL/TLS Cryptographic protocol designed to provide communications security over a computer network.

Suricata Open source intrusion detection system.

Twisted Event-driven networking engine written in Python.

Zero-Day Attack An attack that exploits a previously unknown vulnerability.

1 Introduction

The following chapter contains necessary information for the reader to get a basic understanding of what the IDSanity project aims to accomplish.

1.1 Problem Area

With the rapidly increasing acknowledgement of the importance of information security [1], more and more precautions are taken within businesses to detect threats, protect information and negate or prevent damage done by digital attacks. [1]
One of the most popular [1] ways to achieve better security is to implement one or more intrusion-detection systems and/or web application firewalls in the network.
However, due to the increase of applications used to tackle different aspects of the security infrastructure it is getting harder for system administrators to keep track off and manage all the different rules sets on different sensors.

1.2 Target Audience

This project is mainly developed for the IT Departement at Gjøvik University College and is meant to be a tool for network and system administrators already familiar with IDPS and/or WAFs.
If the IT Departement decides to release this project as open-source it may also be of use to other businesses in need of a centralized framework to manage their network security.
IDSanity may also be of interest to other developers wanting to expand or add new functionality to help maintain their network.

1.3 Employer

Employer for this project is the IT Departement at Gjøvik University College.
The IT Departement consists of 15 employees and works closely with Gjøvik University College's research and educational environment, in the operation of network & computer systems and information security.

Fig. 1: Gjøvik University College

1.4 Project Goal

This project aims to make the management of IDPS and/or WAF sensors less problematic and time consuming. With networks growing more complicated and complex each day the need for more security sensors increases.
When you have to manage a large set of IDPS and/or WAF sensors it may become cumbersome to maintain scripts or manage each of the sensors manually.

IDSanity aims to solve this problem by giving system administrators a centralized, modular framework, where the network administrators can manage a large set of sensors with *one* tool. When IDSanity is installed on each node in the network, network administrators can manage all of the sensors with a CLI on the Master or the web interface.
Since the application is modular it's also really easy to create new modules to make the network security management easier.

1.5 Project Description

The assignment is delivered by the IT Departement at Gjøvik University College and was originally called *GUC Security Rules Management* - later renamed *IDSanity*.
The IT Departement has actively been working towards better detection of security related events, and in a big and complicated network it may become cumbersome to manage rules sets, rule revisions and clients.
The task is to develop a centralized framework for managing HIDS , NIDS , IPS , WAFs and system audit policy rules.
In addition to the framework it self a web interface should also be developed as part of the project assignment.

The *first* and most important part of this project is to develop the framework. IDSanity should be able to push new rule sets out to the nodes, edit existing rule sets, go back to a previous rule set as well as maintain an overview of the current state of the network. It should be as modular as possible allowing other developers to create new modules for other IDPS and WAF s. The appplication will also include an API , allowing the IT Departement to expand or use other programming languages to maintain the clients in the network.
The *second* part is to develop a web interface. Most system administrators use scripts or CLI to maintain clients in a network, but since a web interface makes it easy to get an overview of the current state of the network, the IT Departement also wanted this to be a part of the project assignment.

Since the IT Departement mainly use Debian and CentOS as part of their infrastructure, IDSanity is developed with those operating systems in mind. The application will be packaged for easy installation on Debian but should also work on CentOS since it's also a part of the infrastructure.

2 Background

The following chapter contains necessary information for the reader to get a basic understanding of what the IDSanity application is dependent on. And a short summary of similar projects.

2.1 IDPS

Intrusion Detection And Prevention Systems, abreviated IDPS , is a network security appliance that monitor the network or host for potential malicious activity and take appropriate measures, such as blocking a specific packet or alerting the system administrators if a match is found.
When we talk about IDPSs we usually divide the term in two different parts: network-based and host-based. And each of these two approaches can either be signature-based or anomaly-based.
Each of these different types of IDPSs aim to solve the same problem, but function in very different ways.

2.1.1 Network-based

A network-based IDPS is installed on the network itself and monitors the traffic for potentially malicious traffic. It analyzes network, transport and application protocols to identify suspicious activity [2].
An IDPS sensor can generally be installed in two different ways: Inline or Passive.

Inline

The sensor is installed directly in the communication path between the source and destination. This means that *all* traffic going to and from the network is sent through the IDPS sensor.

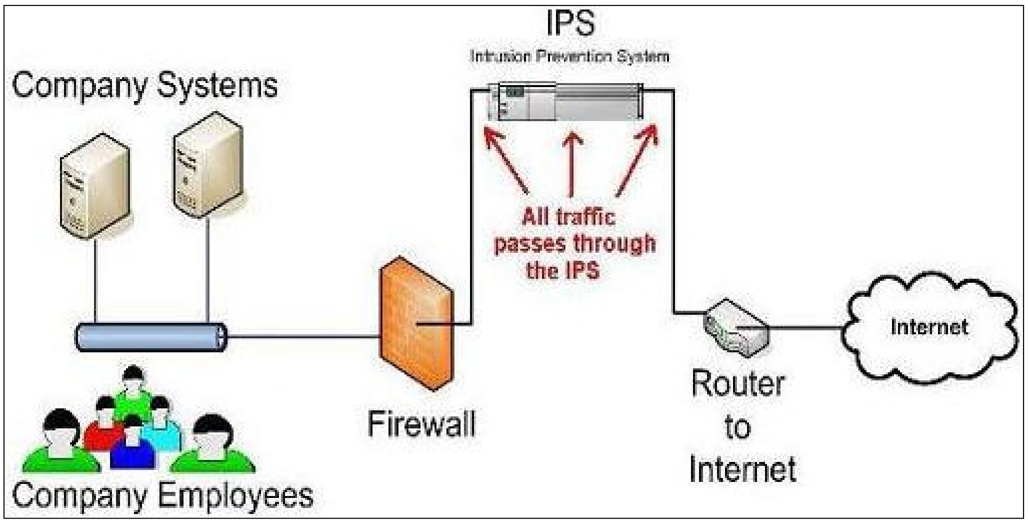


Fig. 2: Example of an inline IDPS.[3]

Passive

A passive sensor is installed in such a way that it receives a *copy* of all the traffic going to and from the network. They are typically installed in key network locations (e.g between two networks).

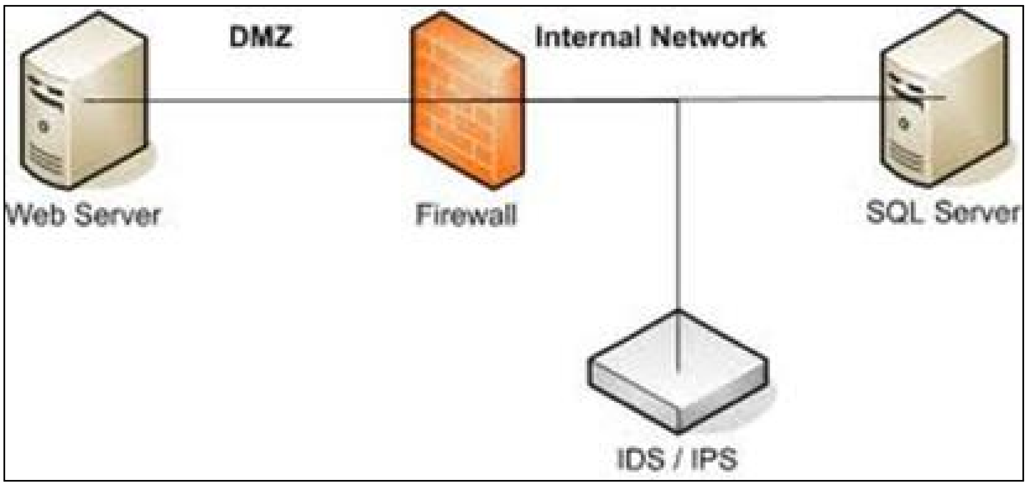


Fig. 3: Example of a passive IDPS.[4]

2.1.2 Host-based

This type of IDPS is installed on the host machine and monitors system logs, file modification/access, which processes are running, if there are any changes to the system etc.
It is typical for a host-based IDPS to have an *agent* installed on the host which communicates directly with the Master or managment servers. The agent reports what is happening on the host computer and the Master or management servers then take appropriate measures if needed.
One example of host-based IDPS is OSSEC, which are discussed later in this chapter.

2.1.3 Signature-based

This form of detection is using a pre-existing signature, which may have been created by other security firms, in order to detect attacks against the network.
This is the simplest detection method available [2] since it just analyzes packets and/or log entries. This data is then compared against a signature, and if a match is found, an attack may be ongoing or already happened.
NIST lists the following as examples a signature-based IDPS may detect:

- A telnet attempt with the username "root".
- An email with the subject of "Free Pictures!!"

Signature-based IDPSs is very effective at detecting threats already known, but ineffective against unknown/0-day attacks.

2.1.4 Anomaly-based

Unlike signature-based IDPS which relies on having the correct signatures at all times, this approach function in a complete different way. By defining a *normal* behaviour, anomalies can be detected by analyzing the state of the network. If the behaviour of

the network is out of the ordinary an alert may be triggered.
McAfee [5] lists a number of anomalies that can occur in a network:

- HTTP traffic on a non-standard port, say port 53 (protocol anomaly)
- A segment of binary code in a user password (application anomaly)
- Too much UDP compared to TCP traffic (statistical anomaly)

The positive thing about anomaly-based IDPS is that it can detect 0-day attacks. As long as the network state is out the ordinary, an alert may be triggered.
While anomaly-based IDPS do alot of good things, the false-positive ratio may be alot higher than signature-based IDPS.

2.2 WAF

A Web Application Firewall (WAF) is a device on your network, a plugin for your server or a filter that applies different rule sets to the HTTP communication between the client and server. A WAF is meant to protect against attacks such as XSS and SQLi.
Some well-known Web Application Firewalls include:

- ModSecurity
- WebKnight
- IronBee

2.3 Suricata

Even though Snort has been the de facto standard IDS for many years now, Suricata (IDPS) are becoming more and more popular. With about the same feature set as Snort and support for multi-threading, Suricata is a very good choice for system administrators wanting to secure their network.

2.3.1 Signatures

Suricata makes use of signatures (also called *rules*) to detect potential dangerous and malicious network traffic.
System administrators can write their own signatures or subscribe to rule feeds provided by security firms around the world.

Example on a Suricata signature:

```
alert http $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"ET
    WEB_SPECIFIC_APPS 20/20 Auto Gallery SQL Injection Attempt --
    vehiclelistings.asp sale_type UNION SELECT"; flow:established,to_server
    ; content:"/vehiclelistings.asp?"; nocase; http_uri; content:"sale_type
    ="; nocase; http_uri; content:"UNION"; nocase; http_uri; pcre:"/UNION\s
    +SELECT/Ui"; reference:cve,CVE-2006-6092; reference:url,www.
    securityfocus.com/bid/21154; reference:url,doc.emergingthreats.net
    /2007517; classtype:web-application-attack; sid:2007517; rev:9;)
```

Fig. 4: Suricata Signature.

A signature consists of three parts: **Action**, **Header** and **Rule-options**.

Action
The Action can be either *Pass*, *Drop*, *Reject* or *Alert*.

Pass: When a signature find a match, the scanning will stop and skip to the end of all rules.
Drop: If the *drop* action is used in a signature and a match is found the packet is dropped immediately.
Note: This will only work in IPS/Inline mode.
Reject: Suricata will reject the packet and both the source and destination will receive a reject packet.
Alert: The packet will be threated like any other non-threatening packet but Suricata will generate an alert. The system administrators will see this alert and can then take appropriate actions.

Header
The second part of the header dictates which protocol Suricata will detect. This value can be one of four values: *tcp*, *udp*, *icmp* or *ip*. If a rule use the option ip this means "all". It's worth mentioning that Suricata 2.0 added a few other protocols as well: *http*, *ftp*, *tls*, *smb* and *dns*.

Rule-options
Rule-options is a set of parameters you can add to your Suricata rules, in order to analyze packets in more depth. Rule-options have a set format:
name: setting;

In Suricata there are specific settings for *meta-information*, *headers*, *payloads* and *flows*.

2.4 OSSEC

OSSEC is a free, open-source HIDS. It combines log analysis, integrity checking, windows registry monitoring, rootkit detection, time-based alerting, and active response, in to a full platform to monito and control the user's systems. [6][7]

2.4.1 Architecture

OSSEC works by having a centralized manager that receives information from different sources to monitor and analyze, eg. OSSEC agents, syslogs, databases or agentless devices [6].

In this OSSEC architecture setup, the OSSEC agent is installed on a number of devices in the system. These agents send their logs and information to the centralized OSSEC server. The server then handles this information based on the configuration, ie. sends an email alert, or performs active response.

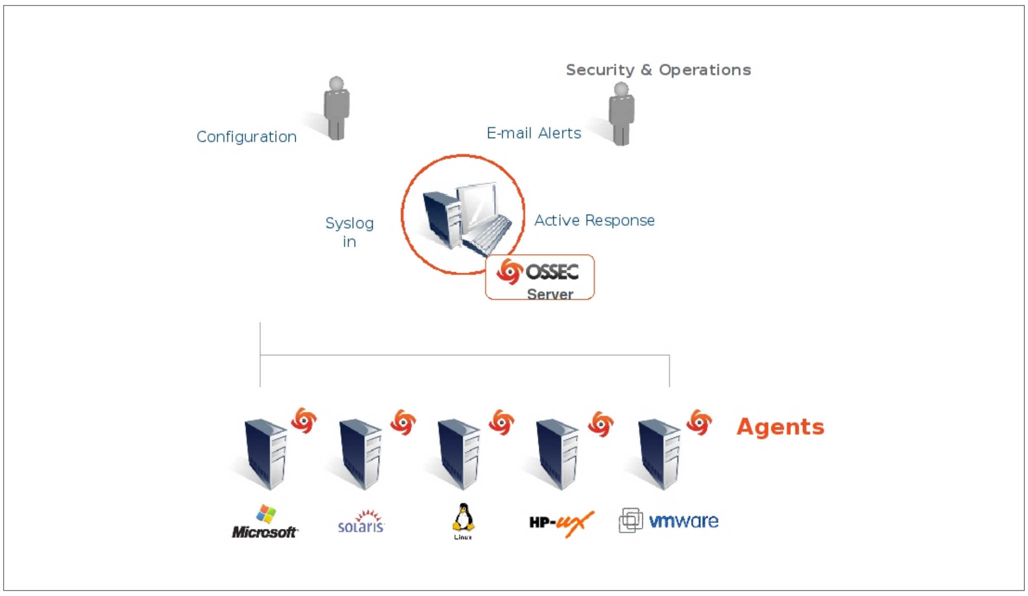


Fig. 5: Example of a OSSEC architecture.[8]

2.4.2 Rule Sets

OSSEC comes with an extensive list of default rules. It is highly discouraged to modify these rules as they are overwritten on every OSSEC upgrade [9]. However, there is a file *local_rules.xml* that one should use to write custom rules for OSSEC. If the user wants to change one of the default rules, a copy of the rule can be added to the *local_rules.xml* file, with the wanted modification and the <*overwrite*="*yes*"> attribute added.
OSSEC rules are written in the well known XML format and supports a wide selection of attributes for optimal customization.
The following example rule detects whenever a USB is inserted into the agent, and generates an alert.

```
 <rule id="10000000" level="12">
        <if\_sid>5100</if\_sid>
        <match>scsi</match>
        <regex>Direct—Access</regex>
        <description>NEW USB FOUND</description>
</rule>
```

Fig. 6: OSSEC Rule.

2.5 ModSecurity

ModSecurity is a Web Application Firewall which provides protection from a range of attacks, including Cross-Site Scripting and SQL Injection.
As mentioned in the ModSecurity Reference Manual [10], it monitors the HTTP traffic and can do real-time analysis with little or no changes to the existing infrastructure.

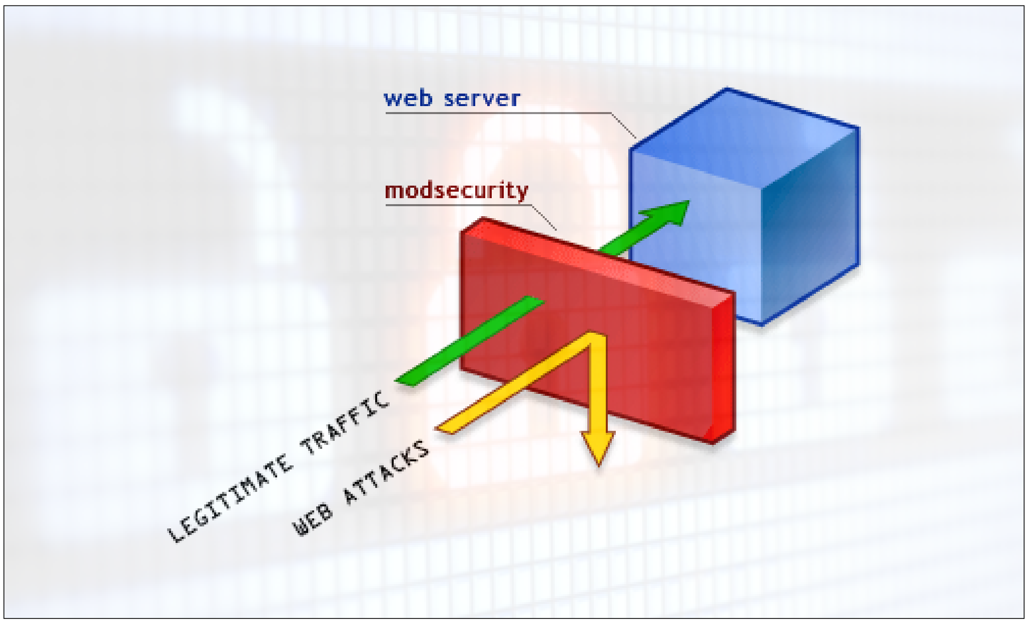
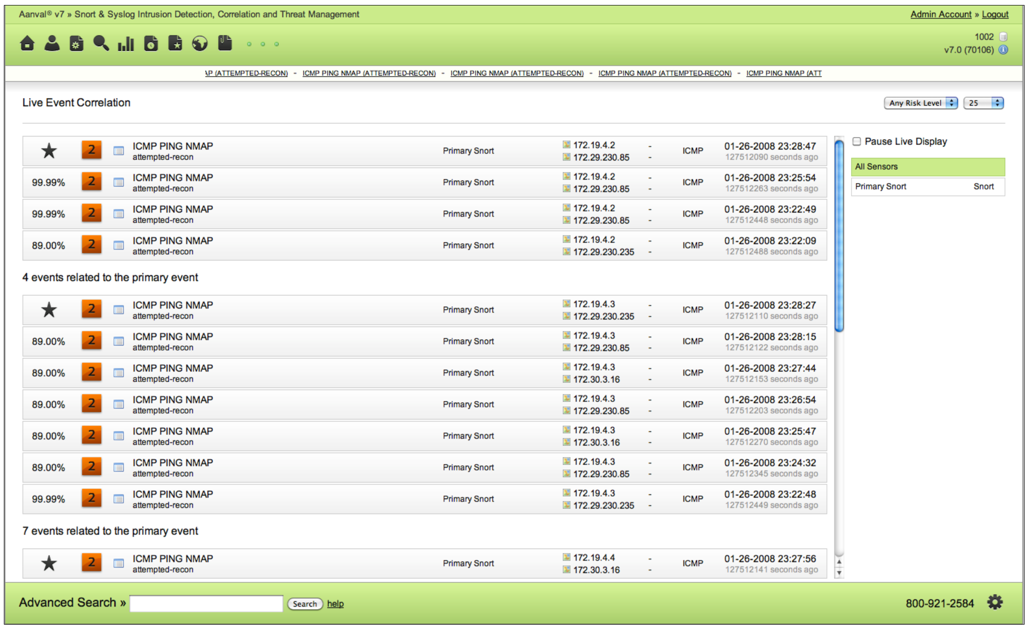


Fig. 7: Example on how ModSecurity works.[11]

2.6 Similar Projects

Aanval

Aanval ("*attack*" in Dutch) is a commercial product designed to maintain Snort, Suricata and Syslog data. Originally developed by Loyal Moses in 2003 it still remains one of the longest Snort capable SIEM products in the industry.



Aanval® v7 » Snort & Syslog Intrusion Detection, Correlation and Threat Management

Admin Account » Logout

1002
v7.0 (70106)

P (ATTEMPTED-RECON) - ICMP PING NMAP (ATTEMPTED-RECON) - ICMP PING NMAP (ATTEMPTED-RECON) - ICMP PING NMAP (ATTEMPTED-RECON) - ICMP PING NMAP (ATT

Live Event Correlation

Any Risk Level | 25

★	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.2 - 172.29.230.85 -	ICMP	01-26-2008 23:28:47 127512090 seconds ago
99.99%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.2 - 172.29.230.85 -	ICMP	01-26-2008 23:25:54 127512263 seconds ago
99.99%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.2 - 172.29.230.85 -	ICMP	01-26-2008 23:22:49 127512448 seconds ago
89.00%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.2 - 172.29.230.235 -	ICMP	01-26-2008 23:22:09 127512488 seconds ago

4 events related to the primary event

★	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.3 - 172.29.230.235 -	ICMP	01-26-2008 23:28:27 127512110 seconds ago
89.00%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.3 - 172.29.230.85 -	ICMP	01-26-2008 23:28:15 127512122 seconds ago
89.00%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.3 - 172.30.3.16 -	ICMP	01-26-2008 23:27:44 127512153 seconds ago
89.00%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.3 - 172.29.230.85 -	ICMP	01-26-2008 23:26:54 127512203 seconds ago
89.00%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.3 - 172.30.3.16 -	ICMP	01-26-2008 23:25:47 127512270 seconds ago
89.00%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.3 - 172.29.230.85 -	ICMP	01-26-2008 23:24:32 127512345 seconds ago
99.99%	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.3 - 172.29.230.235 -	ICMP	01-26-2008 23:22:48 127512449 seconds ago

7 events related to the primary event

★	2	ICMP PING NMAP attempted-recon	Primary Snort	172.19.4.4 - 172.30.3.16 -	ICMP	01-26-2008 23:27:56 127512141 seconds ago

☐ Pause Live Display

All Sensors	
Primary Snort	Snort

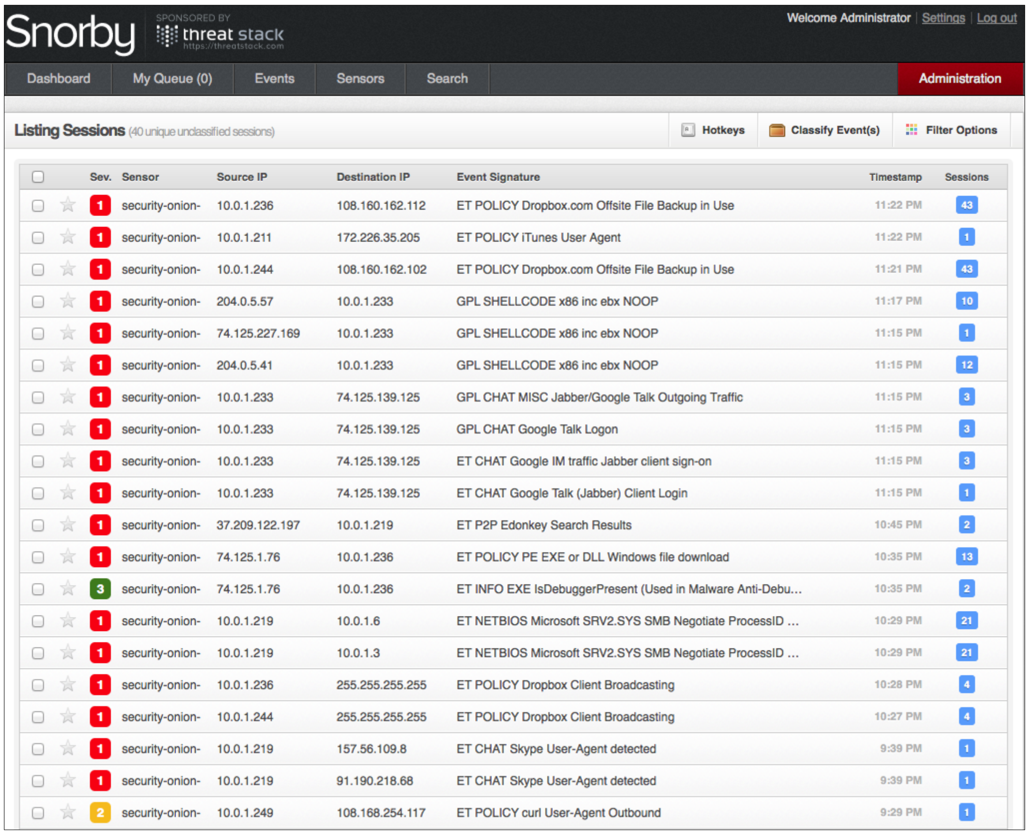
Advanced Search » Search help

800-921-2584

Fig. 8: Aanval Live Event Correlation[12]

Snorby

Snorby is a web application written in Ruby On Rails for network security monitoring. It currently works with systems such as Snort, Suricata and Sagan. Since Snorby is open source it's a very popular choice amongst system administrators that want a nice and powerful web application to monitor the security of their network infrastructure.



Snorby
SPONSORED BY threat stack https://threatstack.com
Welcome Administrator | Settings | Log out
Dashboard
My Queue (0)
Events
Sensors
Search
Administration
Listing Sessions (40 unique unclassified sessions)
Hotkeys
Classify Event(s)
Filter Options

Sev.	Sensor	Source IP	Destination IP	Event Signature	Timestamp	Sessions
1	security-onion-	10.0.1.236	108.160.162.112	ET POLICY Dropbox.com Offsite File Backup in Use	11:22 PM	43
1	security-onion-	10.0.1.211	172.226.35.205	ET POLICY iTunes User Agent	11:22 PM	1
1	security-onion-	10.0.1.244	108.160.162.102	ET POLICY Dropbox.com Offsite File Backup in Use	11:21 PM	43
1	security-onion-	204.0.5.57	10.0.1.233	GPL SHELLCODE x86 inc ebx NOOP	11:17 PM	10
1	security-onion-	74.125.227.169	10.0.1.233	GPL SHELLCODE x86 inc ebx NOOP	11:15 PM	1
1	security-onion-	204.0.5.41	10.0.1.233	GPL SHELLCODE x86 inc ebx NOOP	11:15 PM	12
1	security-onion-	10.0.1.233	74.125.139.125	GPL CHAT MISC Jabber/Google Talk Outgoing Traffic	11:15 PM	3
1	security-onion-	10.0.1.233	74.125.139.125	GPL CHAT Google Talk Logon	11:15 PM	3
1	security-onion-	10.0.1.233	74.125.139.125	ET CHAT Google IM traffic Jabber client sign-on	11:15 PM	3
1	security-onion-	10.0.1.233	74.125.139.125	ET CHAT Google Talk (Jabber) Client Login	11:15 PM	1
1	security-onion-	37.209.122.197	10.0.1.219	ET P2P Edonkey Search Results	10:45 PM	2
1	security-onion-	74.125.1.76	10.0.1.236	ET POLICY PE EXE or DLL Windows file download	10:35 PM	13
3	security-onion-	74.125.1.76	10.0.1.236	ET INFO EXE IsDebuggerPresent (Used in Malware Anti-Debu...	10:35 PM	2
1	security-onion-	10.0.1.219	10.0.1.6	ET NETBIOS Microsoft SRV2.SYS SMB Negotiate ProcessID ...	10:29 PM	21
1	security-onion-	10.0.1.219	10.0.1.3	ET NETBIOS Microsoft SRV2.SYS SMB Negotiate ProcessID ...	10:29 PM	21
1	security-onion-	10.0.1.236	255.255.255.255	ET POLICY Dropbox Client Broadcasting	10:28 PM	4
1	security-onion-	10.0.1.244	255.255.255.255	ET POLICY Dropbox Client Broadcasting	10:27 PM	4
1	security-onion-	10.0.1.219	157.56.109.8	ET CHAT Skype User-Agent detected	9:39 PM	1
1	security-onion-	10.0.1.219	91.190.218.68	ET CHAT Skype User-Agent detected	9:39 PM	1
2	security-onion-	10.0.1.249	108.168.254.117	ET POLICY curl User-Agent Outbound	9:29 PM	1

Fig. 9: Snorby Listing Sessions[13]

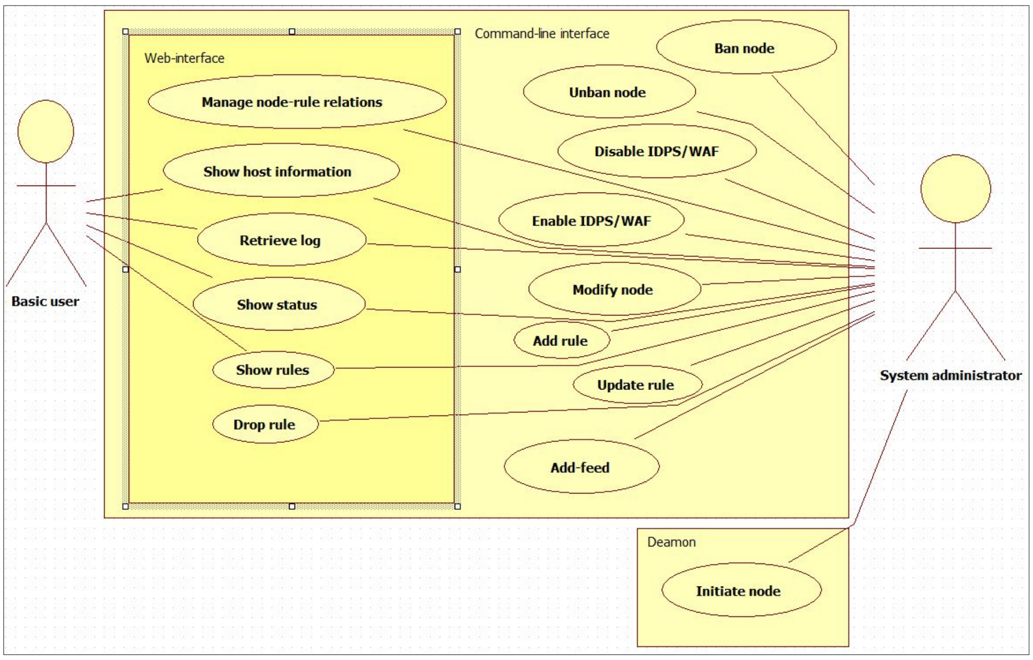
3 Requirement Specification

The requirement specification is based on the employers wishes and the developers decisions of what should be implemented in the IDSanity application to make the usage experience as good as possible for the employer. The chapter will discuss the program flow, operational, functional and system requirements.

3.1 Use Case

The following section will utilize use case diagrams to depict the general flow of the IDSanity application. It is not absolute, but it will give a general overview of interaction between the users and the system.

3.1.1 Use Case Diagram



3.1.2 Comments on Use Case Diagram

In the use case diagram there are only two entities which makes up the flow of the application. There is the System administrator which has access to all functionality in the system, and the basic user which has only the most basic read-only permission. This is because of the security issues that occurs when a user has more permissions than they are equipped to handle.
The use case is of a fairly simplistic art due to how the IDSanity should application operates. There are generally very little nesting of the main functionalities, instead there are a lot of hidden functionality going on in the background, that are not described in this use case diagram.
Third-party library entities functionalities that none of the users of the system has any control over, are also excluded.

Every use case in this diagram happens after authentication is verified.

3.1.3 High level Use Case Descriptions

Due to the simplistic nature of the use case diagram, all use cases are described in a high level format to give some clarification and basic summarization of their functionality.

Use Case	Show host information
Entity	System administrator & Basic user
System	Command-line interface & Web-interface
Goal	Display information about the selected node
Description	The system administrator and a basic user can select a node and show all relevant host information.
Output	Host name, IP-address, nickname, operating system, last seen, last changed, software, list of rules

Use Case	Ban node
Entity	System administrator
System	Command-line interface
Goal	Stop communication with node
Description	The system administrator can ban a node. The node will no longer be able to communicate with the master
Limitation	Executed from Master node only.

Use Case	Unban node
Entity	System administrator
System	Command-line interface
Goal	Regain communication with node
Description	The system administrator can unban a node. The node will now be able to communicate with the master again
Limitation	Executed from Master node only, and is dependent on that the node has previously been banned

Use Case	Enable IDPS/WAF
Entity	System administrator
System	Command-line interface
Goal	Enable the relevant IDPS or WAF program on a node
Description	The system administrator can choose to enable the relevant IDPS or WAF program on a node to start functionality
Limitation	Executed from Slave node only

Use Case	Disable IDPS/WAF
Entity	System administrator
System	Command-line interface
Goal	Disable the relevant IDPS or WAF program on a node
Description	The system administrator can choose to disable the relevant IDPS or WAF program on a node to stop functionality
Limitation	Executed from Slave node only

Use Case	Modify node
Entity	System administrator
System	Command-line interface
Goal	Modify settings or information on a node
Description	The system administrator can change settings or information on a node. E.g Nickname or public-key path

Use Case	Add rule
Entity	System administrator
System	Command-line interface
Goal	Add a new IDPS or WAF rule
Description	The system administrator can add a new rule for use in an IDPS or WAF on a node.

Use Case	Drop rule
Entity	System administrator
System	Command-line interface & Web-interface
Goal	Deactivate or delete an IDPS or WAF rule
Description	The system administrator can deactivate or delete an already existing rule from an IDPS or WAF on a node.

Use Case	Show rules
Entity	System administrator & Basic user
System	Command-line interface & Web-interface
Goal	Show relevant information about the selected rules
Description	The system administrator or a basic user can list information of rules based on id, software, IP-range or direction

Use Case	Update rule
Entity	System administrator & Basic user
System	Command-line interface
Goal	Modify a rules information
Description	The system administrator can modify rule information

Use Case	Add-feed
Entity	System administrator
System	Command-line interface
Goal	Add a new rule feed from a selected source
Description	The system administrator can add rule-feeds from a source that are kept up to date and based on extensive threat intelligence conducted by the feed vendor

Use Case	Manage node-rule relations
Entity	System administrator
System	Command-line interface & Web-interface
Goal	Select what rules should be active on which nodes
Description	The system administrator can select a set of rules to be activated on the different nodes.

Use Case	Retrieve log
Entity	System administrator & Basic user
System	Command-line interface & Web-interface
Goal	Get relevant logs from the selected node
Description	The system administrator and a basic user can chose to retrieve relevant logs from the different nodes in the system.
Output	All log files

Use Case	Show status
Entity	System administrator & Basic user
System	Command-line interface & Web-interface
Goal	Get relevant status about a node or rule
Description	The system administrator and a basic user can retrieve current status of a node or a rule.
Output	If a node or rule is activated or not, number of triggers or error messages.

Use Case	Initiate node
Entity	System administrator
System	Deamon
Goal	Start a new deamon instance on a Master or Slave node
Description	The system administrator start or stop the deamon on a selected node. This will result in if the node is set up and a part of the IDSanity system or not.

3.1.4 Detailed use case

The following two detailed use cases is just a small part of the use cases, but it shows the flow of the most important functions.

Name: Manage node-rule relations	**Actors:** System administrator

Pre-condition: The CLI or the Web-interface must be set up and working. User is authenticated.

Post-condition: Modified settings will be sent to the relevant nodes and the entire system will be updated.

Trigger: User want to modify what rules that are active on a node.

Event flow:

1. System administrator requests the necessary information for options on what to do, from either the Web-interface or through the CLI.
    1. Request node overview
        1. Enable or disable rules based on the list of available rules on the selected node
    2. Request rule overview
        1. Select which nodes the given rule should be enabled or disabled on, based on the list of available nodes for that rule
2. System verifies that only valid options are selected
3. System initiates the communication to sync modifications
4. System verifies that the sync was successful

Event variation:

1. There are no valid information to request
    1. The system administer will be prompted with an error explaining the situation
    2. The system administer can chose whether or not to try again, based on the error message
3. There are no valid options to modify
    1. The system administer will not be able to modify anything, and must have to add rules or nodes to the system to continue
2. Invalid options are selected
    1. The system administer tries to enable a rule on an invalid node.
        1. The node does not have to appropriate software to handle the rule
            1. The system administrator has to update or install the required software to continue, and find the bug in the program that allowed the user to select an invalid option.
        2. The node is not enabled
            1. The system administrator has to enable the node to continue, and find the bug in the program that allowed the user to select an invalid option.
2. Sync verification failed
    1. The system administrator is prompted the option to retry the sync, or to discard current changes

Name: Add rule	**Actors:** System administrator
Pre-condition: Connection to the relevant node is up. The system administrator has written a valid rule to add to the node	
Post-condition: A new rule has been added to one or more valid nodes	
Trigger: The system administrator needs better rule coverage in their system	
Event flow: 1. The system administrator has to decide which nodes the newly created rule should be activating on 2. Then the system administrator has to execute the correct command useing the CLI 3. The system has to verify the rule 4. The system has to verify that the rule fits the selected node(s). 5. The system adds the new rule to the node(s) 6. The system must verify that the rule was added to the node(s)	
Event variation: 1. The rule is not valid, or does not fit one or more of the selected nodes 1. The administrator gets an appropriate error message 2. The administrator can decide if he wants to force add the rule or not 3. The system failed to add the rule to the node 1. The administrator gets an appropriate error message 2. The administrator must debug what might cause the problem	

3.2 Functional Requirements

The functional requirements is a combined list of specific features that is desired in the IDSanity application by the employer and the developers.

3.2.1 Usability

There should be two different main ways to access IDSanity, which covers different functional needs.
These are: CLI & API.
The API allows for custom Web Interfaces and even GUIs to communicate with IDSanity in a standardized way.

CLI

The CLI should cover all administrative and operational functionality, and will be the system administrator's main tool to manage the system.
All setup and initiation functionality, such as starting a Master or Slave node, can be done through the CLI only.
The CLI should have an extensive amount of functionality to meet the user's needs.

Node related functionality

enable Starts the ID(P)S on a node. Slaves can only enable themselves.
disable Stops the ID(P)S on a node. Slaves can only disable themselves.
ban Only available for master. Ignore messages from the node.
unban Only available for master. Allow messages from the node (default).

add-feed Adds a rule feed from a vendor to a node

status Checks the current status of a node and gets **all** sub-command information

- **last-seen** Gets only the last-seen info for a node
- **last-update** Gets the last-update time for a node
- **ipaddress** Gets only the IP-address for a node (only master available for slaves)
- **hostname** Gets only the host name of a node

set Allows modifying host information (not unique ID)

- **nickname** Sets the nickname for a node.
- **public-key** Sets the path to the public key on the node

rule Performs actions on rules for that specific node (see rule actions below)

log Retrieves the most recent log information from a node.

Rule related functionality

add Adds a new rule

update Modifies a rule

drop Removes a rule

show Shows all rules

- **id** Shows a specific rule by ID.
- **software** Shows rules by software.
- **ip** Shows rules by IP-address or IP-range.
- **outgoing** Shows rules for outgoing direction
- **incoming** Shows rules for incoming direction

Web Interface

The main functionality of the Web Interface is to give a simple, intuitive and fast overview of system status. The Web Interface is the system administrator's main way to monitor and document the situation. This should also be more accessible for the users without extensive knowledge of headless systems, such as upper management or trainees.

The Web Interface should support the newest release of all WebKit and gecko -based browser, including IE 7 and above.

It should also be accessible through the use of mobile devices, but limited to the most essential functionality ie. node status.

A RESTful API should be implemented in such a way that the Web Interface can communicate with the master directly and perform the same actions as the CLI.

The main information screens in the Web Interface is the Node overview and the Rule overview screen.

Node overview The Node overview screen should at least have the following basic functionality.

- Show node status - e.g enabled, disabled, IP address, host name, ID(P)S software
- Number of active rules
- Information about the node host
- Information about the installed software on the node

By selecting a specific rule in the node overview, information about that rule should be displayed and options to administer the rule comes up.

Rule overview When entering the Rule overview screen the following information should be displayed.

- Overview of all rules on all nodes
- Detailed list of available rules
- Functionality to manage what rules should be on which nodes

3.2.2 Performance

- One of the biggest performance problems with the IDSanity application will be the amount of bandwidth required for all the communication between nodes. To reduce the bandwidth requirement, all communication should be compressed.
- To minimize hardware resource requirements, only well tested libraries and the newest software releases should be utilized.

3.2.3 Security

- Information leakage is always a possibility when sending information over the network. To ensure that the information can not be read in case of a leak, all communication must go over SSL
- To prevent MITM attacks, public key cryptology should be utilized. By doing this, the nodes can be sure that they communicate with the correct counterpart.

API

IDSanity should provide an API which should run in the background and provide the same functionality as the CLI provides to allow applications such as custom GUIs or Web Interfaces to interact with IDSanity. This API should restrict functionality such as directly creating Nodes, or modifying existing nodes' unique_identifier, and should also restrict access based on a pre-set API key or IP address.

Access to the API should only be allowed with the pre-set key used in an algorithm such as HMAC to create temporary tokens for each individual action, and access should optionally be restricted to the local host.

Each resource should be accessible using standard RESTful behaviour, where HTTP states such as GET represents a SELECT query, PUT represents an UPDATE, POST represents an INSERT query and DELETE represents a DELETE query.

3.3 Operational Requirements

The operational requirements is a combined list of vague functionality that is desired in the IDSanity application by the employer and the developers.

3.3.1 Usability

Requirements for usability is divided into CLI usability and Web Interface usability, due to different targeted user groups and functionality.

CLI

- The IDSanity CLI should be targeted to system administrators with basic knowledge of how Snort, Suricata, OSSEC and ModSecurity operates.

- Familiar and relatable terminology should be used to reduce the time spent learning the program.
- The navigational structure, such as command options, should be similar to how how Snort, Suricata, OSSEC and ModSecurity.

Web Interface

- The IDSanity Web Interface should be targeted to users without the basic knowledge of how Snort, Suricata, OSSEC and ModSecurity operates.
- The design should be in such away that every action requires the minimal amount of key presses possible.
- The Web Interface should communicate with the API
- The Web Interface must display an overview of all nodes and their status, such as rules and activity, in an intuitive way so that the user can gather the necessary information easily.

3.3.2 Availability

IDPS and WAF rule sets are regularly updated due to the increasing threat-intelligence community and technology. It is important to always be up-to-date and a crisis where access to the system is need, can happen at any time. Because of this, availability is a crucial factor.

- The IDSanity application system should have a minimum uptime of 98%, where the last 2% should cover updates, restarts, rule generation and system errors.
- Try-Catch-blockers shoud be implemented, and exceptions should be captured to ensure stability and reduce program-crashes.

3.3.3 Reliability

When dealing with security, reliability is an important aspect to consider. To ensure reliability of the of the information and rule sets, the following requirements must be met.

- To prevent corruption or unwanted tampering of the rules sets, the entire rule set on each node should be regenerated each time a change is committed on the master and with regular intervals.
- Access to direct manipulation of the database should be restricted to IDSanity application, and the system administrator.
- Every change made to the system should be logged for future reference and for reliability control.
- Rule sets should always be able to return to a previous state, in case of user mistakes or system errors.
- All slaves should always mirror their masters rule sets, to ensure synchronous rule sets across the system.

3.3.4 Performance

Due to the lack of hardware on most IDPS and WAF nodes, it is important that the IDSanity application performance is optimized.
Failure to meet the performance requirements might result in stability issues, or in worst case scenario, not being able to run the application. To ensure performance of the system, the following requirements should be met

- Only libraries that are proven to be compatible with each other should be used
- The minimal amount of packages and libraries should be loaded at any time

3.3.5 Environment

IDSanity stability and functionality is reliant on a overall stable environment, both software and hardware.

Software

- IDSanity should run on OSX, Windows and *nix

Hardware

- The hardware should meet the minimal requirement to run the base IDPS or WAF application in addition to the IDSanity application.
- A stable access to electricity, is important to ensure uptime.

3.3.6 Documentation

Documentation is important to ensure that future development is done properly and to ensure that the user is operating the IDSanity application as intended, and that future development can continue without unnecessary confusion.

Inline documentation should be provided to the point where no function should baffle a potential future developer, and each class and function should begin with a *docstrings* that explains that class or functions attributes or parameters, and the intended purpose.

- All Python development code should follow the PEP8 standard. This will make the code more readable, which helps with future development and increases the change to catch code-design flaws.
- The documentation tool Sphinx should be used to generate detailed and easy to read documentation.
- Each class and function should begin with a docstring explaining its purpose
- Docstrings should contain reStructuredText notation to allow Sphinx to generate decent documentation

3.3.7 Security

To ensure the security within the IDSanity application itself, the following requirements must be met.

- A Public Key Infrastructure must be used between the master and the slave, to ensure secure communication and prevent information leakage.

- To restrict access to the system, LDAP and PKI should be used for authentication.
- Every event should be logged.
- Source-code analysis should be conducted regularly to ensure minimal amounts of bugs and possible exploits.

3.4 System requirements

To ensure that IDSanity runs optimally in the GUC IT Departements system environment, the following technical requirements must be met.

Database
PostgreSQL 9.2.x or newer should be used when implementing database functionality.

Programming language
The IDSanity system and all related applications can be developed by the following programming languages.

- PHP
- Python
- C/C++
- HTML/CSS/Javascript

Operating System
IDSanity must run on CentOS 6.5 and newer, or Debian 7.x

Authentication
LDAP should be supported to gather information about system users.

Libraries And Frameworks
To implement secure and tested functionality without re-inventing the wheel, the following libraries and frameworks should be used.

PyCrypto For PKI implementation and signatures
Beautiful Soup For parsing of XML
Python-Ldap For implementing LDAP authentication when accessing the Master node
Python-Json-logger For logging and parsing in JSON format
ConfigParser For maintaining and parsing configuration files
Twisted For network communication and service creation
SQLAlchemy For database object-relational mapping
psycopg2 For PostgreSQL support in SQLAlchemy
Flask For simple, lightweight web services
Flask-restful For creating a RESTful API with Flask
marshmallow For JSON serialization of Python objects
blessings For CLI formatting with colors
dmidecode For parsing smbios and DMI data to generate unique hardware based IDs

4 Design

In this chapter we will describe how IDSanity is designed and how the underlying architecture work. Since IDSanity is a prototype and not a production-ready system we expect that the application will change over time, both in terms of architecture and GUI.

4.1 Sequence Diagram

The sequence diagram shows the IDSanity applications internal communication flow in detail. The diagram will take in account the main entities, Slave and Master, their databases, and their relevant models.
The following diagram displays the synchronization functionality which ensures that the rules are the same between the Slave and the Master.

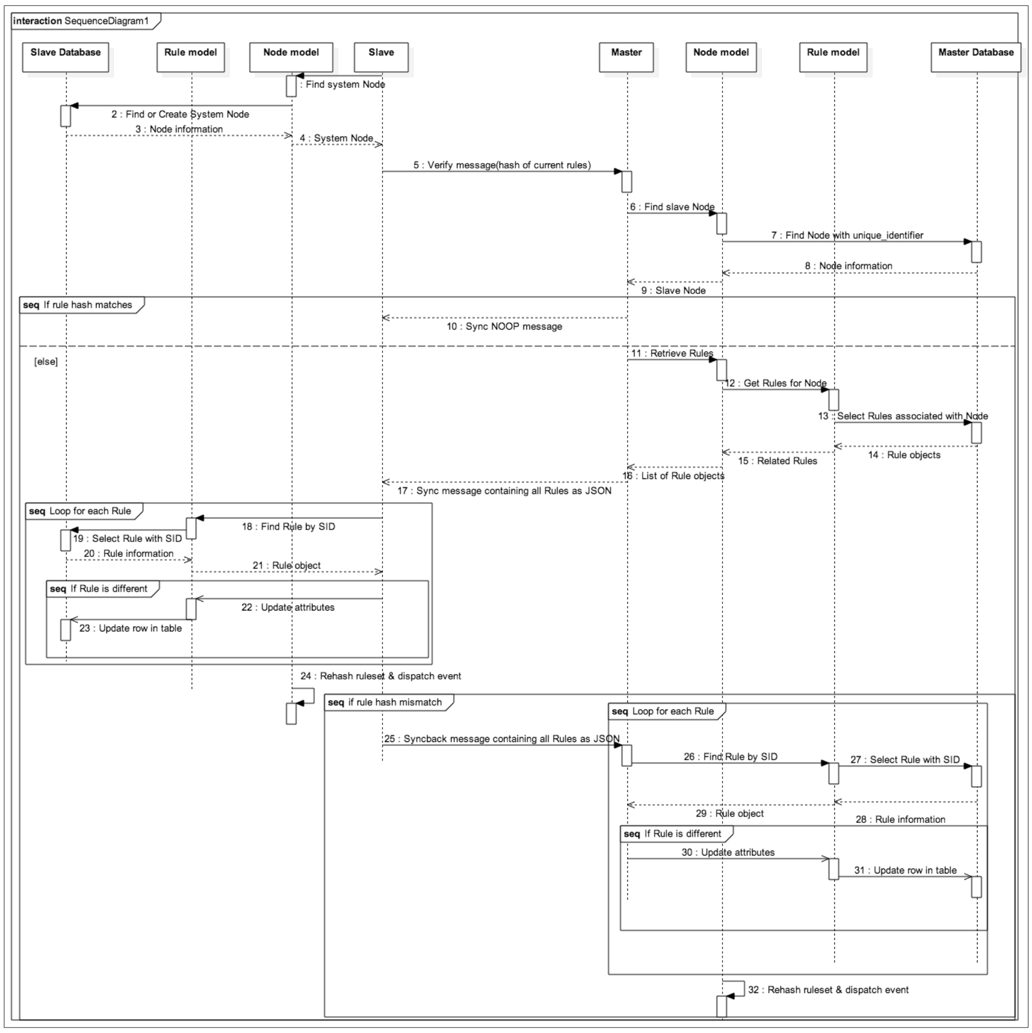


Fig. 10: Sequence Diagram.

1. The initial step in the synchronization functionality is for the Slave to find it's system

Node

2. To do this the Node model contacts the Slave database.
3. If the requested information exists in the database, it is returned to the Node model. However, if no node information can be found, it will create it.
4. After the Node model receives the node information, it relays it directly to the Slave
5. When the Slave has gathered all the initial information, it can verify it's information by creating a hash of it's rule set, and sending it to the Master
6. The Master then has to contact it's Node model to find the correct Slave information
7. The Node model has to relay the request to the Masters Database
8. The database returns it's relevant information back to the Node model
9. Which again sends the information to the Master
10. Now the master can compare the rule set hash it got from the Slave, and the information it had in it's own database. If the hash matches, it means that no changes has happened since the last synchronization check, and the function terminates by responding with a NOOP message
11. However, if the hash do not match, the rest of the synchronization is executed. This happens when the Master requests all the current rules from it's database
12. This rule set request has to go though the Rule model
13. Which again executes the correct database select query
14. The database then returns it's query results to the Rule model
15. The Rule model filters the results and sends the related rules to the Node model
16. Which again formates the list of rules for future use, and sends the rules as a list message of JSON objects back to the Slave
17. Now the Slave has to loop through the list of objects.
18. This loop starts by sending the current rules SID to the Rule model
19. Which executes the correct select query in the Slave database.
20. The database then returns the information it has on the relevant SID to the Rule model
21. The Rule model creates a Rule object based on the information, and sends it to the Slave
22. If the Slave finds that the rule it received is different from the one it already has in it's database, then it sends a update request at the relevant attributes to the Rule model
23. The Rule model executes the requested update query. Repeat until all rules are looped through
24. The Slave then rehashes it's ruleset, and dispatches the relevant event The last stages will be performed in reverse if the Slave had the newest version of the rule set, and the master need an update.

4.2 Deployment Diagram

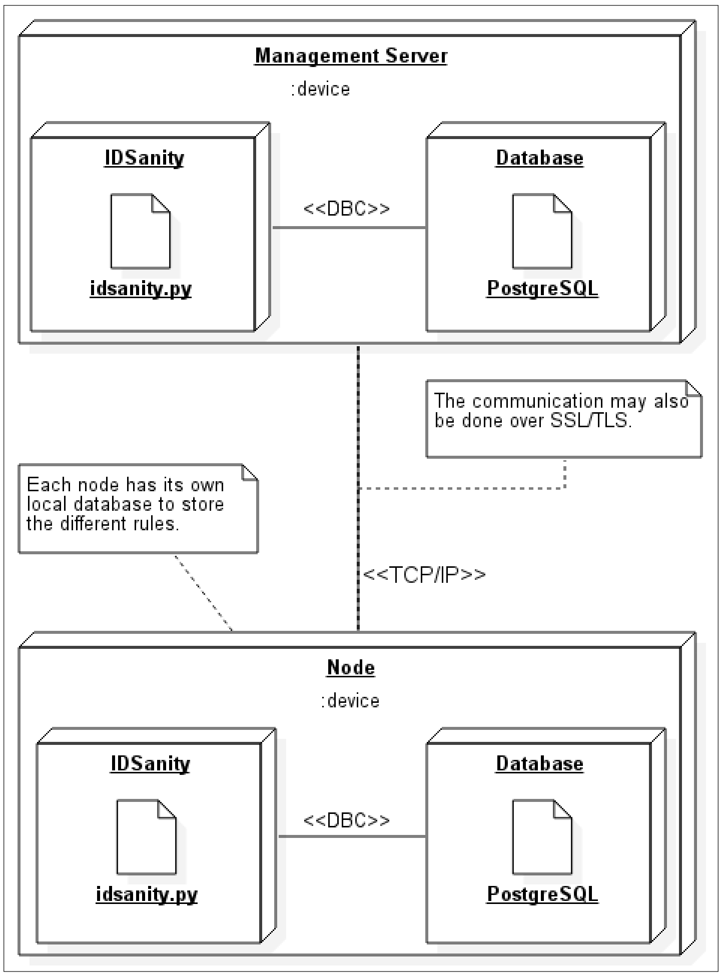


Fig. 11: Deployment Diagram.

In the deployment diagram above we have illustrated how the application is meant to be deployed in a working infrastructure.

Management Server

The *Management Server* is running an operating system (*nix, Windows or OSX) and IDSanity is running as a service in the background. IDSanity communicates with a running database in the background (i.e. PostgreSQLk) on the same server. The *Management Server* have a copy of all the different rule sets which are active or inactive on each of the nodes in the network.

Node

Each node on the network communicates directly with the *Management Server*. The nodes are either running *nix, Windows or OSX and sends all of its rules to the *Management Server*.
There's also a database running in the background on each node, which keeps track of all the active or inactive rules.

Communication

Since IDSanity is using Twisted as its core framework, it's easy to do the communication over SSL/TLS. Even though IDSanity can communicate with the *Management Server* using only HTTP, it is recommended that the communication is done over SSL/TLS to ensure a secure way of transmitting rule sets.

4.3 Class Diagram

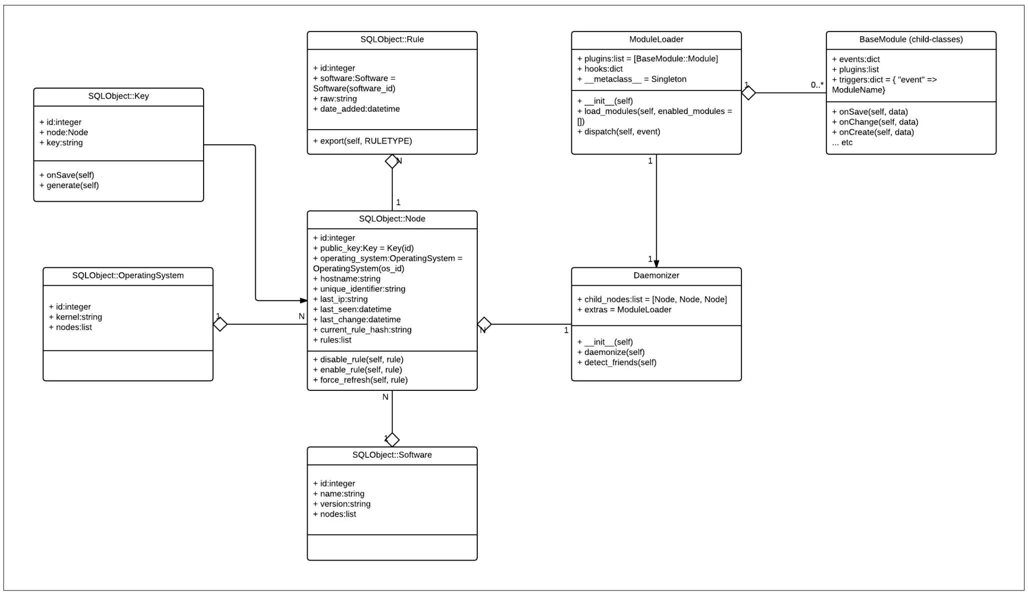


Fig. 12: IDSanity Class Diagram

The class diagram illustrated above shows the most vital classes for the IDSanity application, their properties, and their relationship to one another.
All 3rd party library classes are intentionally omitted from the diagram, since the diagram serves the purpose to provide an overview of the relationships between the main functionality classes withing the application.

4.4 ER-Diagram

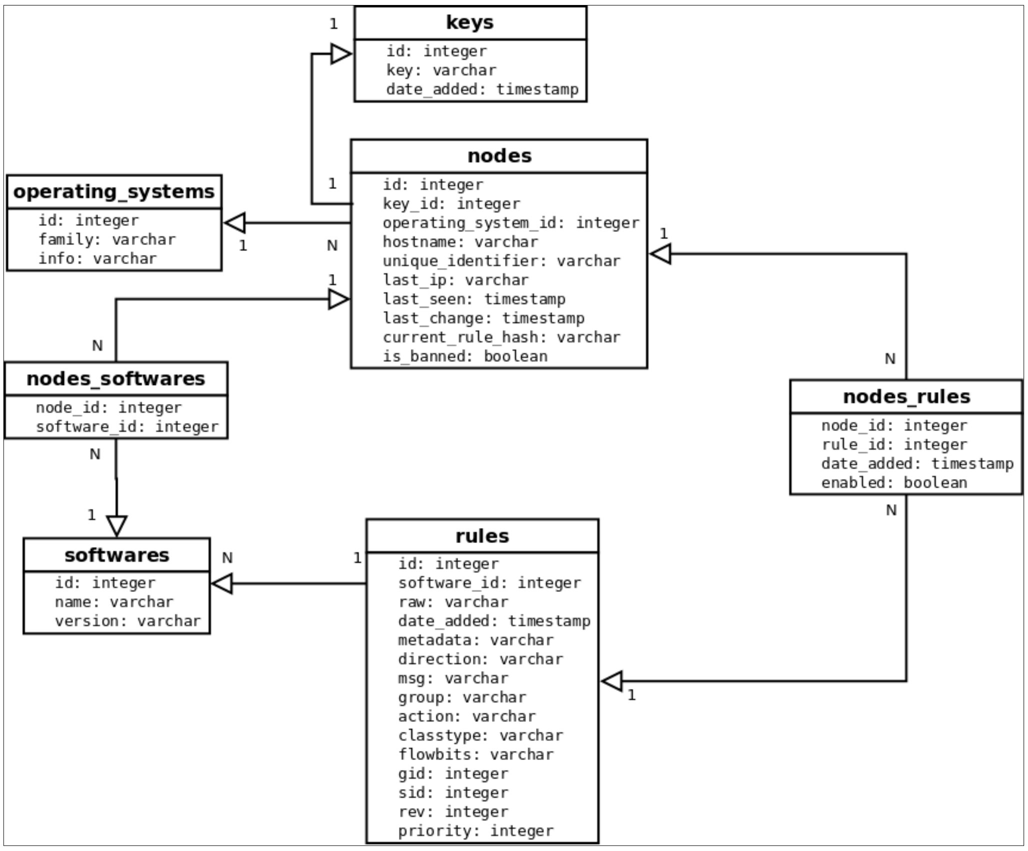


Fig. 13: Entity-Relationship Model

Since we have a relatively complex database layout we have created an ER-diagram in order to give an overview of the database.
This diagram shows how all the different tables in the database are connected together and how the *relationships* and *dependencies* between them are.

5 Implementation

This chapter describes how IDSanity is implemented both in terms of underlying technology and functionality, and how everything described in the previous chapters has been solved. This also includes parts of the source code and illustrations.

5.1 Software Licence

Since this application is meant to be open-source we had a few different software licenses to choose from. We mainly discussed wether to use the BSD, Apache or MIT lisence.
Even though there are similarities between the three, there is also a few important differences.

BSD

With this license you let people do anything with your code without warranty, as long as the author is attributed.

Apache

This is a permissive license that provides an express grant of patent right from contributers to the users.

MIT

A short and to the point license that allows people to anything with the code, as long as the autors are attributed. The code is also provided without warranty.

After a short discussion within the group and with the employer we decided to use the MIT license.
As stated before this license is short and to the point and easy to understand. It is also required to include the license as well as a copyright notice.
The authors can also not be hold liable of any problems or damages because of the code.
Because of this we think the MIT lisence is the best choice for the IDSanity project.

5.2 Development Environment

This section describes the technology used to develop IDSanity and how the backend of this application was implemented during the development period.

Python

Even though Python is at version 3.4.3 we decided to use Python 2.7.x. After a discussion and some research into the subject matter, we discovered that Python 2.7.x still has the best support. And since IDSanity is using a good amount of pre-existing packages, we needed a Python version with good support for the newest ones available.

PostgreSQL

One of the requirements from the employer was that IDSanity should use PostgreSQL as the database backend.
PostgreSQL is becoming a more and more popular [14] database management system

and is widely used in many different applications around the world. PostgreSQL is a object-relational database management system (ORDBMS).
This is similar to a relational database, but uses a object-oriented approach where objects, classses and inheritance is directly supported by the database.

Operating System

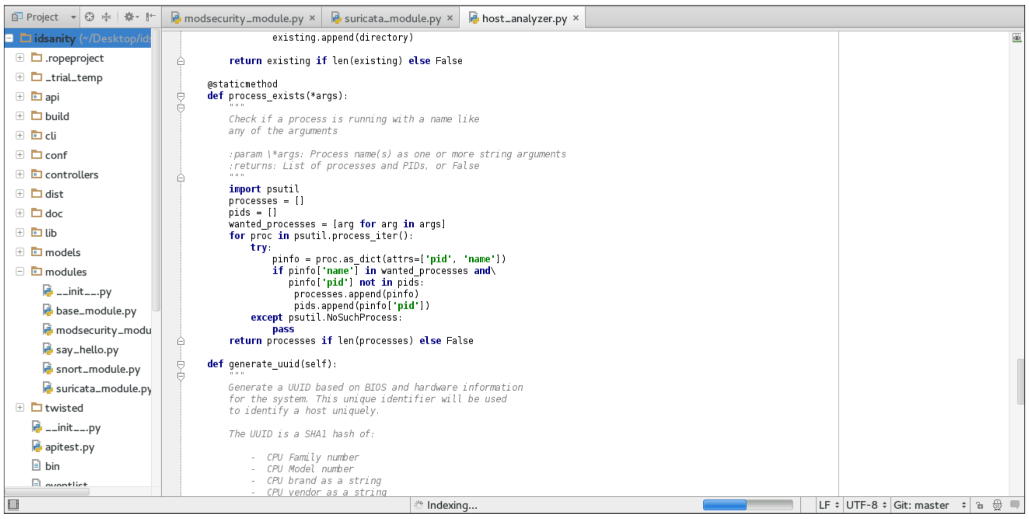
IDSanity is mainly developed for CentOS and Debian since this is the operating systems used by the employer, but the application may also work on other Linux distributions.
IDSanity is made as cross-platform as possible, making it possible to run the application on *nix, Windows and OSX.

Integrated Development Environment (IDE)

As a part of the development process two editors/IDEs, namely PyCharm and Atom, has been used.

PyCharm

PyCharm is a Integrated Development Environment used for programming in Python. It includes code-analysis tools, a debugger, unit testing, version control and also have support for web development with Django.
It's a cross-platform application and does provide a free version as well as a paid, pro version.

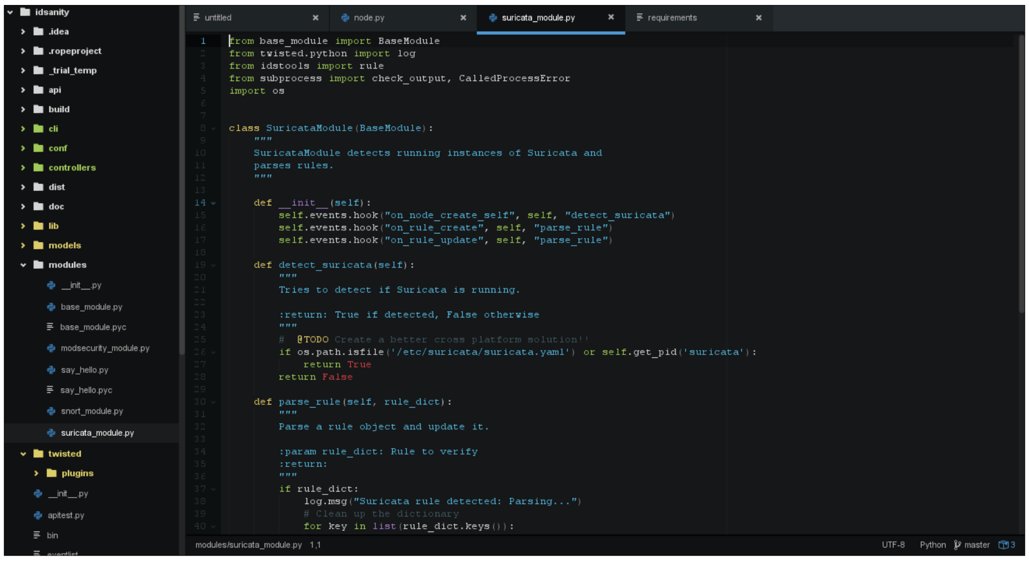


```
                existing.append(directory)

        return existing if len(existing) else False

    @staticmethod
    def process_exists(*args):
        """
        Check if a process is running with a name like
        any of the arguments

        :param \*args: Process name(s) as one or more string arguments
        :returns: List of processes and PIDs, or False
        """
        import psutil
        processes = []
        pids = []
        wanted_processes = [arg for arg in args]
        for proc in psutil.process_iter():
            try:
                pinfo = proc.as_dict(attrs=['pid', 'name'])
                if pinfo['name'] in wanted_processes and\
                   pinfo['pid'] not in pids:
                    processes.append(pinfo)
                    pids.append(pinfo['pid'])
            except psutil.NoSuchProcess:
                pass
        return processes if len(processes) else False

    def generate_uuid(self):
        """
        Generate a UUID based on BIOS and hardware information
        for the system. This unique identifier will be used
        to identify a host uniquely.

        The UUID is a SHA1 hash of:

            -  CPU Family number
            -  CPU Model number
            -  CPU brand as a string
            -  CPU vendor as a string
```

Fig. 14: PyCharm Integrated Development Environment.

Atom

Atom is a hackable and open-source editor made available through Github. It's highly customizable using for example CSS and Javascript.
Node.js is integrated so it's really easy to create your own extensions or download pre-existing ones.
The design and functionality mimic that of *Sublime Text* which has been a very popular editor for many years now.



```
from base_module import BaseModule
from twisted.python import log
from idstools import rule
from subprocess import check_output, CalledProcessError
import os


class SuricataModule(BaseModule):
    """
    SuricataModule detects running instances of Suricata and
    parses rules.
    """

    def __init__(self):
        self.events.hook("on_node_create_self", self, "detect_suricata")
        self.events.hook("on_rule_create", self, "parse_rule")
        self.events.hook("on_rule_update", self, "parse_rule")

    def detect_suricata(self):
        """
        Tries to detect if Suricata is running.

        :return: True if detected, False otherwise
        """
        #  @TODO Create a better cross platform solution!!
        if os.path.isfile('/etc/suricata/suricata.yaml') or self.get_pid('suricata'):
            return True
        return False

    def parse_rule(self, rule_dict):
        """
        Parse a rule object and update it.

        :param rule_dict: Rule to verify
        :return:
        """
        if rule_dict:
            log.msg("Suricata rule detected: Parsing...")
            # Clean up the dictionary
            for key in list(rule_dict.keys()):
```

Fig. 15: Atom Editor.

5.3 Writing New Modules

This section will describe how other developers can create their own modules to use with IDSanity. It will go through what is required and what's optional in order to get the module up and running.

Inherit The Base Module

IDSanity ships with a BaseModule all other modules need to inherit from in order to work and get the necessary functionality.
In order for this to work we first need to import the BaseModule, as seen in *line #1*.
Then you create a class of your choice and *inherit* from the BaseModule you included before.

```
1 from base_module import BaseModule
2
3 class SuricataModule(BaseModule):
4     """
5     SuricataModule detects running instances of Suricata and
6     parses rules.
7     """
```

Fig. 16: Creating a module: Inheritance.

Hooking Into Events

The modules have no functionality if you don't make them react to the events available in IDSanity. For a full list of the *Core Events* available, check out subsection 5.6.4.
In order for this to work you need to setup an event hook. An example on how to accomplish this is shown in the code example below.

We first create a *contructor* by using the built-in function "*__init__(self)*". In this constructor we setup the *event hook* by specifying which events we want to react to and which function that should be run when a specific event is dispatched.

On *line #10* we tell IDSanity that when the event "*on_node_create_self*" is dispatched, the method "*detect_suricata()*" should trigger.
"*detect_suricata()*" is a method we define later in the class.

```
1  from base_module import BaseModule
2
3  class SuricataModule(BaseModule):
4      """
5      SuricataModule detects running instances of Suricata and
6      parses rules.
7      """
8
9      def __init__(self):
10         self.events.hook("on_node_create_self", self, "detect_suricata")
```

Fig. 17: Creating a module: Event Hooking.

Creating The Method That Will Trigger

The last thing to do is to create the method that will run when the event is dispatched. This is just a normal method as any other method in Python. This method obviously need to be named exactly the same as in the event hook in "*__init__(self)*".

```
1  from base_module import BaseModule
2
3  class SuricataModule(BaseModule):
4      """
5      SuricataModule detects running instances of Suricata and
6      parses rules.
7      """
8
9      def __init__(self):
10         self.events.hook("on_node_create_self", self, "detect_suricata")
11
12     def detect_suricata(self):
13         """
14         Tries to detect if Suricata is running.
15
16         :return: True if detected, False otherwise
17         """
18         if os.path.isfile('suricata.yaml') or self.get_pid('suricata'):
19             return True
20         return False
```

Fig. 18: Creating a module: Event Hooking.

Wrapping Up

As soon as the Module is done, you need to register the Module in IDSanity. The way IDSanity works is that it will only load and activate modules that exists in the configuration file, as shown in the figure below.

```
[configuration]
master = 1
port = 9595
modules =
  SurricataModule

[postgresql]
host = localhost
database = idsanity
user = idsane
password = idsanity
```

Fig. 19: IDSanity: Configuration file.

In the modules section you specify which modules IDSanity should load. So as long the module name is included in the configuration file and the module itself exists in the modules folder, the module should now be loaded by IDSanity and react to the events you have specified.
The complete source code of the Suricata module is listed on the next page.

End-result: Suricata Module

```
from base_module import BaseModule
from idstools import rule
from subprocess import check_output, CalledProcessError
import os

class SuricataModule(BaseModule):
    def __init__(self):
        self.events.hook("on_node_create_self", self, "detect_suricata")
        self.events.hook("on_rule_create", self, "parse_rule")
        self.events.hook("on_rule_update", self, "parse_rule")

    def detect_suricata(self):
        """
        Tries to detect if Suricata is running.

        :return: True if detected, False otherwise
        """
        if os.path.isfile('suricata.yaml') or self.get_pid('suricata'):
            return True
        return False

    def parse_rule(self, rule_dict):
        """
        Parse a rule object and update it.

        :param rule_dict: Rule to verify
        :return:
        """
        if rule_dict:
            for key in list(rule_dict.keys()):
                if rule_dict[key] == []:
                    del rule_dict[key]

            if "direction" in rule_dict:
                if "<" in rule_dict["direction"] \
                   and ">" in rule_dict["direction"]:
                        rule_dict["direction"] = "BIDIRECTIONAL"
                elif ">" in rule_dict["direction"]:
                        rule_dict["direction"] = "OUTBOUND"
                elif "<" in rule_dict["direction"]:
                        rule_dict["direction"] = "INBOUND"

    def validate_rule(self, raw_rule):
        """
        Validate rules by using the parse method from idstools.

        :param raw_rule: Rule to parse (string)
        :return: Parsed rule or False
        """
        parsed_rule = rule.parse(raw_rule)

        if parsed_rule:
            return parsed_rule
        else:
            return False

    def get_pid(self, procname):
        try:
            return int(check_output(["pidof", procname]).strip())
        except CalledProcessError:
            return False
```

Fig. 20: Source Code: Suricata Module.

5.4 Master daemon

When the application initializes, it starts up as a Twisted plugin which parses the command line options and determines whether or not IDSanity is running in Master or Slave mode, and then uses the appropriate Factory to set up either the Master (server) daemon, or the Slave (client) daemon. The only difference between these are whether or not to listen for - or attempt connecting to - a remote host.

class `idsanity.idsanityd.Idsanityd`

Bases: `twisted.internet.protocol.Protocol`

This is the IDSanity Master daemon, which listens for connections on the given port, and sets up the ModuleLoader.

It dispatches the `ON_INIT` event when it's done instantiating.

`connectionMade()`

This function is run as soon as a connection is made to the other end, and begins the identification process by calling `Communicator.shake_hands()`. Dispatches the `ON_CONNECT` event

Todo Dispatch on_node_connect event

`dataReceived(`*data*`)`

Whenever data is received over the connection, this function is run and passes the data to `Communicator.receive()`.

5.5 Client daemon

class `idsanity.idsanity.Idsanity`

Bases: `twisted.protocols.basic.LineReceiver`

This is the IDSanity client (Slave), which establishes a connection to an endpoint at the specified IP.

`connectionMade()`

Overriden hook that will be run as soon as a connection is made to the other end of the line.

Triggers `Communicator.shake_hands()` to initiate the identification process.

`dataReceived(`*data*`)`

Overridden hook that will be run as soon as anything is received over the connection. This data will be passed to `Communicator.receive()` where parsing takes place.

`end` = **'die'**

5.6 Events

Most of the IDSanity architecture is based around events, where core functions in the program, such as the models, register events as soon as they're loaded, and dispatch them as things happen.

For example, when a new node object is inserted into the database, `ON_NODE_CREATE` will be dispatched, which lets any function react to the event, and work with the object that triggered it.

This could be a potential security issue if a malicious function hooked into an event, however, if a hypothetical adversary has access to the source and permission to edit it, the problem is on a much more serious level. As with most things Python, *we're all responsible adults here*.

5.6.1 The Events class

This is the class responsible for keeping everything glued together, by keeping track of events and assigning them an integer values which are automatically incremented, similar to an `enum` in certain languages.

The `Events` class, like the ModuleLoader, uses the Singleton metaclass and instantiating it will always return the same instance - there can never be two or more instances of the Events class to ensure the events it contains are consistent, and can be accessed from anywhere after importing it and requesting an instance through `Events()`. This provides access to registering events and hooks, and then dispatching events.

Registering events is done by calling `Events().register('my_cool_event')` anywhere after importing the class. When it comes to *modules*, it's recommended that all event registration is done in a dedicated method specifically named `register_events`, as the `ModuleLoader` will attempt to run this as soon as it has loaded each module, and events *must* be registered before they can be hooked into.

When an event is registered, it's turned into a class constant of Events, and can be accessed through `Events().MY_COOL_EVENT` directly, but most functions that take an event as parameter also accept a string representation of its name.

Just like an `enum` in other languages, every new item is automatically assigned a unique incrementing integer value, which makes it efficient for use as indices in lists, and consequently it is also iterable; it's possible to iterate through it and get a tuple of the event name and its value in a `for` loop like so:

```
for (event_constant, value) Events():
    print(event_constant, value)
```

After an event has been registered, it can be *dispatched*. If no hooks have been set up for the event, nothing happens. But if hooks have been set up, anywhere in the program after the event was registered, dispatching the event will cause it to execute the desired function on the desired class.

The `SayHi` example module is a good example of this. When IDSanityd starts up, Idsanityd overrides Twisted's `connectionMade()` callback which is called when a connection is received on the listening port. If we would put **all** the code in here for anything we might want to do, now and in the future, when a connection is made ... well, we'd end up with a lot of unorganized cluttered code in the main class.

Instead, we register an event in the constructor, like so: `Events().register('on_connect')`

Now inside the `connectionMade` function, we just call `Events().dispatch('on_connect')`, or `Events().dispatch(Events().ON_CONNECT)` if you prefer, which will cause *all* functions that have hooked into this event to be executed.

class `idsanity.lib.events.Events`

Bases: `object`

Contains constants for Event hooks - similar to an Enum but less complex. self.append('MY_HOOK') in an empty Events, will give

```
>>> self.MY_HOOK
1
 >>>
```

`action_map` **= {}**

Keeps a list of functions to call for each event upon dispatch

`counter` **= 0**

Denotes the highest used index, and value, for event constants

`dispatch`(*evt*, *parameter*={})

Dispatches an event

Parameters

- `event` – Event to dispatch
- `parameters` – Parameters to send to event hook

`event_value_pair`(*event*)

Retrieves the actual value for an event

Parameters `event` – String or Integer representation of event

Returns tuple of event name and its const value, i.e (EVENT_NAME, 1)

`exists`(*event*)

Verifies that an event has been registered and exists

Parameters `event` – String representation of event name

Returns True if event exists, False if it doesn't

`hook`(*event*, *cls*, *method*)

Hooks into an event

Parameters

- `event` – Event to hook into
- `cls` – Class to run method on
- `method` – Class method to run when event is dispatched

Returns True if successful, False if event could not be found

`i` **= 1**

Iterator indicating current position in list of events, used to iterate

`instance` **= None**

`last_event` **= None**

Keep track of the last dispatched event for testing and verification

`next`()

Gets the next element when iterating this object, and make sure it exists, otherwise skip to the next one if it has been removed

Returns String representation of event name

`register`(*new_event*)

Register a new event

Parameters `new_event` – Name of event

Returns True on success, False if event already existed

`reverse_map` **= {}**

Maps integer value to constant name to allow retrieving the event from its value

`unregister`(*event*)

Removes an event, but does not decrement the maximum event const value

Parameters `event` – Event to remove

Returns The value of the removed event, or False if unknown event

5.6.2 Hooking into Events

Hooking (or *listening) is what we call a reaction* to an event, because events are thrown out as a “do something or dont I dont care”, like waving a flag, and a hook specifies that when this ‘flag’ pops up, respond by running the specified function. Hooks are therefore defined as `Events().hook('on_connect', class, method)` where class is an actual class object, like `self`, and the method is represented as a string name.

The dispatch method also takes an optional extra argument, which can be a dict of arguments or anything else you wish to receive in your function. This will be passed to the method when the hook triggers. Most core Events pass the object the event was triggered for as parameter, i.e `ON_NODE_CREATE` will pass the newly created Node object. Exactly what is passed, and thus can and should be received, is documented in the *Core Events* table.

5.6.3 Naming convention

Since hooks are responses to events, it’s common practice to name the events `on_` something, and therefore all existing events are named hierarchically, with the scope following the `on_` keyword, such as `on_cli` for the **c**ommand **l**ine **i**nterface, followed by the resource they operate on and the action they perform. For example, an event for changing the nickname for a node in the command line interface is thus called `on_cli_node_set_nick`, and to display information about a node we have the event `on_cli_node_print_info`.

5.6.4 Core Events

Events that exist by default in IDSanity are listed here, together with when they are dispatched and what argument needs to be passed along with them.

Event name	Dispatched	Parameter
ON_CLI_NODE_STATUS	When *all* status info about a node is printed via the CLI	
ON_CLI_NODE_STATUS_HOSTNAME	When a user requests to print the hostname of a node via CLI	
ON_CLI_NODE_STATUS_IPADDRESS	When a user requests to print the IP address of a node via the CLI	
ON_CLI_NODE_STATUS_NICKNAME	When a user requests printing the nickname of a node via CLI *Note: * Not implemented yet	
ON_CLI_NODE_STATUS_LAST_SEEN	When a user requests printing the `last_seen` info for a Node via the CLI	
ON_CLI_NODE_STATUS_LAST_UPDATE	When a user requests printing the `last_update` info for a Node via the CLI	

Continued on next page

Table 1 – continued from previous page

Event name	Dispatched	Parameter
ON_CLI_NODE_SET_NICK	When a user requests setting a new nickname for a Node *Note: * Not yet implemented	
ON_CLI_NODE_SET_KEY	When a user sets the pubkey for a Node via the CLI	
ON_CLI_NODE_ENABLE	When a user enables a Node that was previously disabled. Enabling also enables the IDS.	
ON_CLI_NODE_DISABLE	When a user disabled a previously enabled Node via the CLI	
ON_CLI_NODE_BAN	When a user *bans* a node via the command line interface. Banning a Node prevents all communication with it.	
ON_CLI_NODE_UNBAN	When a user *bans* a node via the command line interface. Banning a Node prevents all communication with it.	
ON_CLI_RULE_ADD	When a user adds a new Rule through the command line.	
ON_CLI_RULE_ASSIGN	When a user assigns an already existing Rule to a Node via the command line interface	
ON_CLI_RULE_UNASSIGN	When a user removes an existing Rule from a Node via the CLI	
ON_CLI_RULE_SHOW	When a user prints information about a Rule via the command line interface	
ON_CLI_RULE_UPDATE	When a user updates a rule via the command line interface.	
ON_CLI_RULE_DROP	When a user removes a Rule from all nodes via the command line interface.	

Continued on next page

Table 1 – continued from previous page

Event name	Dispatched	Parameter
ON_SOFTWARE_CREATE	When a new Software is created and added to the database for the first time.	Software() instance
ON_SOFTWARE_UPDATE	When a Software object is changed and saved to database.	Software() instance
ON_SOFTWARE_DESTROY	When a Software object is deleted.	Software() instance
ON_RULE_CREATE	When a new Rule is created and inserted into the database.	Rule() instance
ON_RULE_UPDATE	When an existing Rule is updated and saved to database.	Rule() instance
ON_RULE_DESTROY	When an existing Rule is deleted from the database.	Rule() instance
ON_KEY_CREATE	When a public key is inserted into the database.	Key() instance
ON_KEY_UPDATE	When an existing public key is updated and saved to database	Key() instance
ON_KEY_DESTROY	When an existing public key is deleted from the database.	Key() instance
ON_NODE_CREATE	When a new Node object is created and inserted into the database.	Node() instance
ON_NODE_CREATE_SELF	When the Node representing the current system is created for the first time and saved to db	Node() instance
ON_NODE_UPDATE	When a Node object is modified and saved to the database.	Node() instance
ON_NODE_DESTROY	When a Node object is deleted.	Node() instance
ON_NODE_CONNECT	When a Node connects to another Slave/Master, or vice versa.	Node() instance (the node that just connected)
ON_NODE_DISCONNECT	When a Node's connection to another Node (whether master or slave) is closed.	Node() instance (the node that just disconnected)

Continued on next page

Table 1 – continued from previous page

Event name	Dispatched	Parameter
ON_NODE_BAN	When a Node that was previously unbanned gets banned, i.e when the *Node.is_banned* changes.	Node() instance
ON_NODE_UNBAN	When a Node that was previously banned gets unbanned, i.e when the *Node.is_banned* changes.	Node() instance
ON_NODE_RULE_CREATE	When a new association between a Rule and a Node is created.	NodeRule() instance
ON_NODE_RULE_UPDATE	When an existing association between a Rule and a Node is updated.	NodeRule() instance
ON_NODE_RULE_DESTROY	When an existing association between a Node and Rule is deleted from the database. *Note:* This is very rare, NodeRules should not be deleted but instead, disabled.	NodeRule() instance
ON_OPERATING_SYSTEM_CREATE	When a new OperatingSystem is inserted into the database.	OperatingSystem()
ON_OPERATING_SYSTEM_UPDATE	When an OperatingSystem is updated and saved to database.	OperatingSystem()
ON_OPERATING_SYSTEM_DESTROY	When an OperatingSystem is removed from the database.	OperatingSystem()
ON_CONNECT	When the system's Node connects to another Node (Master/Slave)	None
ON_INIT	When IDSanityd initializes.	None
ON_NODE_RULE_DISABLE	When a NodeRule that used to be enabled becomes disabled in the database, regardless of how. *Note:* Use instead of the ON_NODE_RULE_DESTROY event.	NodeRule() instance

Continued on next page

Table 1 – continued from previous page

Event name	Dispatched	Parameter
ON_NODE_RULE_ENABLE	When a NodeRule that used to be disabled has its `enabled` attribute set to `True`	NodeRule() instance
ON_NODE_SYNC_NOOP	When a Synchronization request triggered no sync because the Rule-set hashes matched.	JSON message that triggered no sync to occur.
ON_NODE_SYNC_SEND	When a synchronization is sent from the Master to a Slave. This event only occurs on the *Master* node.	JSON message that will be sent to the slave node for sync
ON_NODE_SYNC_RECEIVE	When a *Slave* receives a sync message from the Master and begins synchronization.	List of NodeRule() objects that were modified in sync
ON_NODE_SYNC_BACK	When the *Master* receives unsync'ed data from a slave after a sync has occured.	JSON object with the information received from slave
ON_RULE_REQUEST_SID	When a *Slave* is about to send a message reserving Rule slots and SIDs from the Master.	Number representing the amount of SIDs to request.
ON_RULE_RECEIVE_SID	When a Slave receives new SIDs from the Master.	JSON object with list of received SIDs from Master
ON_SYSTEM_MESSAGE_SEND	Whenever *any* message is sent from the current Node to the Node on the other end.	JSON message to send on the network
ON_SYSTEM_MESSAGE_RECEIVE	Whenever *any* message is received by the current node from the other end of the line.	JSON received over the network.
ON_SYSTEM_MESSAGE_ERROR	Whenever an *error message* is received from the Node on the other end of the line.	JSON message with error message.

Table 1: Core Events.

5.7 Internal Communication

Inter-node communication in IDSanity work in two ways, the API and the internal communication - and generally, Nodes should not interact via the RESTful API as this is mainly to be used for communication with other local processes, and mostly aimed at writing alternative interfaces.

The internal communication is done over TLS and requires a protocol to follow.

That's where this chapter comes in, to answer such questions as:

- What is sent, when?
- What responses are to be expected?
- What happens to the received information?
- How is this information safeguarded?

5.7.1 Message Structure

Messages are sent as JSON internally, making parsing simple. Each message contains the resource the message is about, an action it wishes to perform, and a data field containing the payload.

Each message also contains a 'signature'. Signatures are created by signing the JSON object (without the signature field) with the current nodes private key.

Each node makes sure to discard any message that does not contain a valid signature.

Example:

```
{
  'action': 'update',
  'resource': 'node',
  'data':
          {
            'node':
                         {
                           'id': '9',
                           'unique_identifier': '124d-13dd-3r43-5fee',
                          }
          },
  'signature': 'wefe4r403m44f84cjc4f9kr42'
}
```

Fig. 21: IDSanity: Message Structure.

5.7.2 Message Response Structure

Message responses are always the updated, created or deleted object, which lets the client verify that the changes it requested occurred. The only exception is when an error occurs, which can be verified by the client as the message is now a JSON formatted error message.

Example:

```
{
   'resource': 'error',
   'action': 'failed',
   'data':
            {
              'error':
                         {
                           'message':  'An error occurred'
                         }
              },
   'signature': 'ngruktpd32t5k35kf4125'
}
```

Fig. 22: IDSanity: Message Response Structure.

5.7.3 Recurring communication

There are mainly two common types of messages occurring regularly between the master and the slave: Updating the Rule list, and identifying one another.

Most other messages are responses to these.

Identification

A node, master or slave, may at any time ask for identification. Identification implies that the node requesting it is not entirely sure about what node is on the other end, and would like to know the information it requires to create (or update) a Node object in its local database.

The response message contains all this information, as well as the public key required to verify any subsequent signatures made by the node.

Request

```
{
  'action': 'identify',
  'resource': 'node',
  'data': {},
  'signature': 'qweqwod3r392fdmf4'
}
```

Fig. 23: IDSanity: Identification Request.

Response

A node identifies by sending its host information, including public key and its unique identifier as a response to an identification request.

```
{
  'action': 'identification',
  'resource': 'node',
  'data':
          {
            'node':
                      {
                        'unique_identifier': '124d-13dd-3r43-5fee',
                        'public_key': 'PUBKEYHERE',
                        'hostname': 'Slave1',
                        ....
                      }
          },

  'signature': '14e1djd1rjfjnd49rfes4'
}
```

Fig. 24: IDSanity: Identification Response.

At this point, if there is no node with this unique identifier in the database, the public_key field will be used to verify the signature. If it matches, the master continues to add this node to its database.

Note: Only when the database has no public key stored will it be inserted for the node. An existing node cannot change public key without the administrator manually removing the existing public key from the node in its database.

Synchronizing the rule list

Each node running in *slave* mode will periodically request to get an updated rule set. It keeps track of the last time it received one, and requests it again every 30 minutes.

This is the most common type of message, and is very simple: It contains the node's unique identifier, a hash digest of its current *enabled* rule set, and looks like this:

```
{
  'resource': 'node',
  'action': 'sync',
  'data':
        {
          'node':
                 {
                   'current_rule_hash': 'longhashhere'
                 }
        },
  'signature': 'qwe1j42342j2nr2f24oif'
}
```

Fig. 25: IDSanity: Sync Request.

If the `current_rule_hash` matches what the slave has in its database, it sends a response with an empty data field. If it doesn't match, the data field will contain all the rules that have been changed.

5.8 External Communication (API)

The RESTful API is how outside applications can communicate with IDSanity, which is especially useful when writing GUIs or web interfaces. It's entirely based on `Flask` and `Flask-Restful`, and provides full access to the underlying database.

5.8.1 Route structure

The API follows default RESTful routing, like so:

Action	HTTP Verb	URL
Find by ID	GET	/resource/1
Find all	GET	/resources
Update	PUT	/resource/1
Create	POST	/resources
Delete	DELETE	/resource/1

Table 2: IDSanity: API Route Structure

More specific information about each route can be found in the documentation for each individual view.

5.8.2 IDSanityApi module

This is the basis of the API, which sets up all routes and links them to the respective views, and creates the Flask application. This class is instantiated by default if IDSanity is running as a *Master*, and turned off by default if running as a slave.

The API can be forced on or off using the `--api 1` or `--api 0` option when starting the idsanity daemon.

class `idsanity.api.idsanity_api.IdsanityApi(`*import_name='api'*`)`

Bases: `flask.app.Flask`

The RESTful Webservices exposing functionality for building GUIs and web interfaces

`idsanity.api.idsanity_api.add_cors_headers(`*response*`)`

API views

Each view in the API subclasses the two base classes, one for single objects and one for multiple objects.

By subclassing these and setting the class attributes `model` and `serializer`, standard RESTful behaviour is provided to the resource.

The contents of the view is defined mainly by the given `serializer`, which handles serialization and deserialization of JSON <-> Python objects.

Base view (Single object)

class `idsanity.lib.api.views.single_base_view.SingleBaseView`

Bases: `flask_restful.Resource`

Base view subclassed by all *single* resources. Should make all resources that reference a single model record "just work"

`delete(`*id*`)`

Deletes a single model record

Parameters `id` – id of the record to delete

`get(`*id*`)`

Get a single model

Parameters `id` – id of the model to retrieve

`methods` = **['DELETE', 'GET', 'PUT']**

`put(`*id*`)`

Updates a single model record

Parameters `id` – id of the record to update

Base view (Many objects)

class `idsanity.lib.api.views.many_base_view.ManyBaseView`

Bases: `flask_restful.Resource`

Base view subclassed by all resources. Hopefully, this class should make all models "just work"

`get()`

Get a list of every <model name>

`methods` = **['GET', 'POST']**

`post()`

Creates a new record of <model>

Key

The Key resource provides access to the Key model in the database, which contains public keys belonging to the different nodes. This resource only allows `GET` and `PUT` by default, as shown in the table below, since keys should not be created or deleted unless done automatically by IDSanity.

Action	HTTP Verb	URL
Find Key by ID	GET	/api/key/1
Find all	GET	/api/keys
Update	PUT	/api/key/1

Table 3: IDSanity: API Key Routes.

class `idsanity.lib.api.views.key_view.KeyView`

Bases: *`idsanity.lib.api.views.single_base_view.SingleBaseView`*

`methods` = **['DELETE', 'GET', 'PUT']**

`model`

alias of `Key`

`serializer`

alias of `KeySchema`

class `idsanity.lib.api.views.keys_view.KeysView`

Bases: *`idsanity.lib.api.views.many_base_view.ManyBaseView`*

`methods` = **['GET', 'POST']**

`model`

alias of `Key`

`serializer`

alias of `KeySchema`

Node

Every machine running IDSanity is considered a `Node`, and all information we need to know about the system is contained in the Node object, like system information, and relationships to which Software(s) it's running, what Rule(s) it should have, and what OperatingSystem it runs on.

Generally, a Slave node will only have two Nodes in its database: itself, and the Master, since it only stores information about the Nodes it knows and a Slave only communicates with the Master.

The Master on the other hand will have a record for each Node it communicates with.

Nodes **cannot** be created through the API, because this should happen automatically.

Action	HTTP Verb	URL
Find Node by ID	GET	/api/node/1
Find all Node	GET	/api/nodes
Update a Node	PUT	/api/node/1
Delete	DELETE	/api/node/1

Table 4: IDSanity: API Node Routes

class `idsanity.lib.api.views.node_view.NodeView`

Bases: *`idsanity.lib.api.views.single_base_view.SingleBaseView`*

`methods` = **['DELETE', 'GET', 'PUT']**

`model`

alias of `Node`

`serializer`

alias of `NodeSchema`

class `idsanity.lib.api.views.nodes_view.NodesView`

Bases: *`idsanity.lib.api.views.many_base_view.ManyBaseView`*

`methods` = **['GET', 'POST']**

`model`

alias of `Node`

`serializer`

alias of `NodeSchema`

Rule

A Rule contains all columns needed by a Snort-like rule, in addition to the `raw` field which contains the raw rule. All rules in the database are linked to the `Node` (s) they have been added to with a `NodeRule` object.

Rules can be created, updated, deleted and retrieved through the API.

Action	HTTP Verb	URL
Find a Rule by ID	GET	/api/rule/1
Find all Rules	GET	/api/rules
Update a Rule	PUT	/api/rule/1
Create new Rule	POST	/api/rules
Delete a Rule	DELETE	/api/rule/1

Table 5: IDSanity: API Rule Routes.

class `idsanity.lib.api.views.rule_view.RuleView`

Bases: *`idsanity.lib.api.views.single_base_view.SingleBaseView`*

`methods` = **['DELETE', 'GET', 'PUT']**

`model`

alias of `Rule`

`serializer`

alias of `RuleSchema`

class `idsanity.lib.api.views.rules_view.RulesView`

Bases: *`idsanity.lib.api.views.many_base_view.ManyBaseView`*

`methods` = **['GET', 'POST']**

`model`

alias of `Rule`

`serializer`

alias of `RuleSchema`

NodeRule

The `NodeRule` is what makes the association between a `Node` and a `Rule`, and should be created as soon as a `Rule` is created to link it to the `Node` (s) it should run on.

`NodeRule` (s) cannot be deleted, but can be disabled. By disabling it, it will no longer exist on the node until it's enabled again.

Action	HTTP Verb	URL
Find NodeRule by ID	GET	/api/node-rules/1
Find all NodeRules	GET	/api/node-rule
Update a NodeRule	PUT	/api/node-rule/1
Create new NodeRule	POST	/api/node-rules

Table 6: IDSanity: API NodeRule Routes.

class `idsanity.lib.api.views.node_rule_view.NodeRuleView`

Bases: *`idsanity.lib.api.views.single_base_view.SingleBaseView`*

`methods` = **['DELETE', 'GET', 'PUT']**

`model`

alias of `NodeRule`

`serializer`

alias of `NodeRuleSchema`

class `idsanity.lib.api.views.node_rules_view.NodeRulesView`

Bases: *`idsanity.lib.api.views.many_base_view.ManyBaseView`*

`methods` = **['GET', 'POST']**

`model`

alias of `NodeRule`

`serializer`

alias of `NodeRuleSchema`

OperatingSystem

An `OperatingSystem` contains information about an OS, mainly to keep track of simple system information such as kernel version and OS family. As soon as a `Node` is created, this information is also created or updated.

Action	HTTP Verb	URL
Find OperatingSystem	GET	/api/operating-system/1
Find all OperatingSystems	GET	/api/operating-systems
Update an OperatingSystem	PUT	/api/operating-system/1
Create new OperatingSystem	POST	/api/operating-systems
Delete an OperatingSystem	DELETE	/api/operating-system/1

Table 7: IDSanity: API OperatingSystem Routes.

class `idsanity.lib.api.views.operating_system_view.OperatingSystemView`

Bases: *`idsanity.lib.api.views.single_base_view.SingleBaseView`*

`methods` = **['DELETE', 'GET', 'PUT']**

`model`

alias of `OperatingSystem`

`serializer`

alias of `OperatingSystemSchema`

class `idsanity.lib.api.views.operating_systems_view.OperatingSystemsView`

Bases: *`idsanity.lib.api.views.many_base_view.ManyBaseView`*

`methods` = **['GET', 'POST']**

`model`

alias of `OperatingSystem`

`serializer`

alias of `OperatingSystemSchema`

Software

The `Software` object contains information about individual IDPS / Firewall softwares that exist in the network. These objects are created automatically by its respective module - for example, the `Suricata` module will detect if Suricata is running on a Node, or if a rule that was just added is actually a Suricata rule, and if the software (and the software version) does not exist in the database, it will be created.

Each Rule belongs to a software, and each Node has (hopefully) at least **one** Software.

Action	HTTP Verb	URL
Find a Software by ID	GET	/api/software/1
Find all Softwares	GET	/api/softwares
Update a Software	PUT	/api/software/1
Create new Software	POST	/api/softwares
Delete a Software	DELETE	/api/software/1

Table 8: IDSanity: API Software Routes.

class `idsanity.lib.api.views.software_view.SoftwareView`

Bases: *`idsanity.lib.api.views.single_base_view.SingleBaseView`*

`methods` = **['DELETE', 'GET', 'PUT']**

`model`

alias of `Software`

`serializer`

alias of `SoftwareSchema`

class `idsanity.lib.api.views.softwares_view.SoftwaresView`

Bases: *`idsanity.lib.api.views.many_base_view.ManyBaseView`*

`methods` = **['GET', 'POST']**

`model`

alias of `Software`

`serializer`

alias of `SoftwareSchema`

5.9 External Communication (CLI)

The CLI is how users (and scripts) can interact with the database in idsanity, since the daemons run autonomously. Actions performed through the CLI will be noticed by the daemons, and synchronized when a sync occurs.

IDSanity uses Twisted's built-in module for Command Line Parsing. This means that each command and its options are contained as a `list` in a class that subclasses `usage.Options`.

There are different types of lists providing different kinds of functionality, such as `subCommands` which defines commands, and `optFlags` which provides `--flag` functionality.

IDSanity keeps each subcommand and its options, optflags and subcommands in separate files and directories, and all commands related to `node` or `rule` can be found in its own directory under the `lib/cli/arguments` directory.

5.9.1 CLI Argument Events

The ArgumentEvents class is responsible for *registering* all events that will be dispatched for the different arguments.

When an argument is passed to IDSanity CLI, all the options will be sent as an argument with the dispatched event, and any module can technically hook into this and provide additional behaviour.

By default, IDSanity utilizes two command line event controllers to manage this. The Event Controllers hook into the different events dispatched, and perform the requested action.

class `idsanity.lib.cli.argument_events.ArgumentEvents`

Bases: `object`

Parses commands and arguments, and determines what action to take

`arguments` = **{}**

`react()`

Parse options and dispatch hooks. This must be done *after* all hooks have been registered, otherwise no functions will be run!

`register_events()`

5.9.2 CLI Argument Hooks

ArgumentHooks creates the "root" of the argument tree by linking the idsanitycli command to its two subcommands, `node` and `rule`, from which all other actions branch out.

class `idsanity.lib.cli.argument_hooks.ArgumentHooks`

Bases: `twisted.python.usage.Options`

`subCommands` = **[['node', None, <class 'idsanity.lib.cli.arguments.node.node.NodeHooks'>, 'Perform**

5.9.3 CLI Subcommands

The Commandline Interface is resource based, which means that each subcommand affects a resource in one way or another, and is thus on action upon that resource. IDSanity mainly manages two resources, `Node` and `Rule`, and everything else is to describe these resources and their relationships.

As such, the two main subcommands on the command-line are `node` and `rule`.

Resource Arguments: Node

Node command

class `idsanity.lib.cli.arguments.node.node.NodeHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['show', None, 'Shows all linked nodes and their IDs'], ['enable', None, 'Enables the ID(**

`parseArgs`(*argument*)

`subCommands` = **[['status', None, <class 'idsanity.lib.cli.arguments.node.status.NodeStatusHooks'>, 'S**

Node feed

class `idsanity.lib.cli.arguments.node.feed.NodeFeedHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['add', None, 'Adds a vendor rule feed to the node'], ['remove', None, 'Removes a vendo**

`parseArgs`(*argument*)

Node set custom settings

class `idsanity.lib.cli.arguments.node.set.NodeSetHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['nickname', 'n', 'Set the nickname for a node'], ['public-key', 'k', 'Set public key from p**

`parseArgs`(*argument*)

Node status

class `idsanity.lib.cli.arguments.node.status.NodeStatusHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['hostname', 'h', "Shows a nodes's hostname"], ['nickname', 'n', 'Shows the nickname fo**

`parseArgs`(*argument*)

Resource Arguments: Rule

Rule command

class `idsanity.lib.cli.arguments.rule.rule.RuleHooks`

Bases: `twisted.python.usage.Options`

`parseArgs`(*argument*)

`subCommands` = **[['show', None, <class 'idsanity.lib.cli.arguments.rule.show.RuleShowHooks'>, 'Print**

Add Rule

class `idsanity.lib.cli.arguments.rule.add.RuleAddHooks`

Bases: `twisted.python.usage.Options`

`optParameters` = **[['node', 'n', 0, 'Node to add the Rule to (0 means self)'], ['host', 'h', 'Amnesthesias**

`parseArgs`(*argument*)

Assign rule

class `idsanity.lib.cli.arguments.rule.assign.RuleAssignHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['rule', None, 'Rule ID to assign to a node']]**

`optParameters` = **[['node', 0, 'Node to assign rule to']]**

`parseArgs`(*argument*)

Drop rule

class `idsanity.lib.cli.arguments.rule.drop.RuleDropHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['id', None, 'Rule ID to delete']]** `parseArgs`(*argument*)

Show rule

class `idsanity.lib.cli.arguments.rule.show.RuleShowHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['node', None, 'Show all rules for node'], ['all', None, 'Show all rules'], ['id', None, 'Sho**

`parseArgs`(*argument*)

Unassign rule

class `idsanity.lib.cli.arguments.rule.unassign.RuleUnassignHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['rule', None, 'Rule ID to unassign from a node']]**

`optParameters` = **[['node', 'n', 0, 'Node to unassign rule from']]**

`parseArgs`(*argument*)

Update rule

class `idsanity.lib.cli.arguments.rule.update.RuleUpdateHooks`

Bases: `twisted.python.usage.Options`

`optFlags` = **[['id', None, 'Rule ID to update']]**

`parseArgs`(*argument*)

5.10 Controllers

Controllers make up the layer between what a user wants to do, and the resulting output of the users desired action.

Thus, controllers are how IDSanity reacts to user specified events, and are used by the command line interface after commands have been parsed, by dispatching one of the CLI events which the appropriate controller hook into.

5.10.1 IDSanity CLI Controllers

The Command Line Interface consists of two major parts, the commands & event triggers, and the event controllers.

Every event controller sets up hooks to the events dispatched by the commands it operates on, and then perform the requested actions in the hooked method.

There are two different event controllers: `NodeController` and `RuleController`, one for each resource.

Node Controller

class `idsanity.lib.cli.controllers.node_controller.NodeController`

Bases: `object`

Perform actions on a node by hooking into events dispatched by command line arguments.

This is where all things related to a Node on the command line actually *happens*.

`TERMINAL_DELIMITER` = ' | '

Used to separate columns in text printed to the command line

`TERMINAL_LINE` = '==================================='

Used to distinctively separate rows (headlines) on the command line

`ban`(*id*)

Blacklist a node from further communication with the master

Parameters `id` – Node ID

Returns Bool

`disable(`*id*`)`
Disable the ID(P)S or Firewall software running on a Node's host
Parameters `id` – Node ID
Returns Bool

`enable(`*id*`)`
Enable the ID(P)S or Firewall software running on a Node's host
Parameters `id` – Node ID
Returns Bool

`print_status(`*id='0'*`)`
Print extensive information about a node's current status, such as amount of (active/inactive) rules, ip address, hostname, software, operating system, nickname, ID, etc
Parameters `id` – Node ID
Returns True

`print_status_hostname(`*id*`)`
Display a Node's hostname
Parameters `id` – Node ID
Returns the hostname

`print_status_ip(`*id*`)`
Display a Node's ip address
Parameters `id` – Node ID
Returns the IP

`print_status_nick(`*id*`)`
Display a Node's nickname
Parameters `id` – Node ID
Returns the nickname

`print_status_seen(`*id*`)`
Display a Node's last seen date
Parameters `id` – Node ID
Returns Last seen date

`print_status_updated(`*id*`)`
Display a Node's last update date
Parameters `id` – Node ID
Returns Last update date

`register_hooks()`
Register all Event hooks used by this controller

`set_key(`*id*`)`
Set the public key for a Node
Parameters `id` – Node ID
Returns True

`set_nick(`*id*`)`
Set the nickname on a Node
Parameters `id` – Node ID
Returns True

`unban(`*id*`)`
Remove Node from blacklist
Parameters `id` – Node ID
Returns Bool

Rule Controller

class `idsanity.controllers.cli.rule_controller.RuleController`

Bases: `object`

Perform actions on a rule by hooking into events dispatched by command line arguments.

This is where actions related to individual rules on the command line actually occur, and output presented to user is generated.

`TERMINAL_DELIMITER` = ' | '

Used to separate columns on the command line

`TERMINAL_LINE` = '================================='

Used to separate headlines on the command line

`add_rule`(*args*)

Adds a new rule - if rule is an ID, it assigns an existing rule to a node; otherwise a rule is created and assigned

Parameters `args` – Dict containing {"node_id": id, "software": software_name, "version": software_version, "raw": The rule to add}

Returns The newly created Rule, or None

`assign_rule`(*args*)

Assigns a rule to a node

Parameters `args` – List containing {"rule_id": id_of_rule, "node_id": id_of_node}

Returns True or False

`drop_rule`(*args*)

Removes a rule by ID

Note: This will affect ALL nodes this rule is assigned to. Confirm this from user.

Parameters `args` – Dictionary containing key 'rule_id'

Returns True or False

`register_hooks`()

`show_rules`(*args*)

Show all rules for a node (or all nodes) If no valid id, hostname or unique identifier is found, show all rules

Parameters `args` – dict with {"node_id": id_hostname_unique_identifier }

Returns List of rules

`unassign_rule`(*args*)

UnAssigns a rule from a node

Parameters `args` – List containing {"rule_id": id_of_rule, "node_id": id_of_node}

Returns True or False

`update_rule`(*args*)

Updates an existing rule by replacing its' content

Parameters `args` – List containing {"rule_id": id, "raw": new_rule}

Returns True or False

5.11 Models

IDSanity uses SQLAlchemy's ORM, which turns every row and its relationships in the database into Python objects by mapping the tables and columns onto `models`.

When a model object is modified and saved in the code, the changes are reflected in the database, which makes the database a whole lot easier to work with.

All models subclass the BaseModel which is provided by SQLAlchemy, and then also subclass the `ModelExtension` mixin class which provides additional functionality to each model, like `.find()`, `.update()` and `.save()` methods.

Furthermore, each model must also have a serializer if it will be accessible via the RESTful API.

5.11.1 Model Mixin (providing extra functionality)

class `idsanity.lib.models.model_extension.ModelExtension`

Bases: `object`

Provides class methods for all models that don't come with SQLAlchemy by default, and makes updating and finding models much much easier and much more readable.

classmethod `create(`***arguments*`)`

Creates a new object and commits it immediately. Do not use this in a loop, instead, use create_many.

Parameters `**arguments` – All attribute=value required for the object

Returns The created object

classmethod `create_many(`*arguments=[]*`)`

Creates multiple new objects of a given model and commits them. Use this to create many instances of a model

Parameters `arguments` – List of arguments to create model objects from

Returns List of created objects

`delete(`*commit=True*`)`

Deletes an object

Parameters `commit` – Whether or not to commit the current session afterwards

classmethod `exists(`***arguments*`)`

Checks if an object exists

Parameters `**arguments` – Column=Value pair to find object on

Returns bool

classmethod `find(`*record_id=None*, ***arguments*`)`

Find a model by any argument (defaults to ID).

Parameters

- `record_id` – ID of object to find (optional)
- `**arguments` – attribute=value to find model by

Returns None or an instance of the requested model

classmethod `id_in(`*ids=[]*`)`

Gets all models that match the criteria, using SQL "column IN (...)"

Parameters `ids` – All IDs to match

Returns All models with a column value matching one in the list

classmethod `id_not_in(`*ids=[]*`)`

Gets all models that do NOT match the criteria, using SQL "column NOT IN (...)" This is the inverse of id_in

Parameters `arguments` – All IDs to exclude

Returns All models with a column value matching one in the list `save()`

Saves the object immediately

`update(**arguments)`

Updates any model by setting the attributes as arguments, like Node.update(hostname='CoolHost')

Parameters `**arguments` – Dynamic arguments

Returns True or False depending on the success of the operation

classmethod `where(**arguments)`

Find a model by any arguments

Parameters `**arguments` – attribute=value to find models by

Returns All matching models

5.11.2 Key

class `idsanity.lib.models.key.Key(**kwargs)`

Bases: *`idsanity.lib.models.model_extension.ModelExtension`*, `sqlalchemy.ext.declarative.api.B`

This is the class which holds public keys used by different nodes in the network.

It's a simply class associated to the nodes.

`date_added`

Date the key was added to the database

`id`

The Key ID

`key`

The raw public key

5.11.3 Node

class `idsanity.lib.models.node.Node(**kwargs)`

Bases: *`idsanity.lib.models.model_extension.ModelExtension`*, `sqlalchemy.ext.declarative.api.B`

The Node contains all information about a system, whether Master or Slave. It's not clear from the database whether or not a Node is a Master or Slave as the only difference is in the communication and responses between them.

The Node object is, together with the Rule object, the most important objects in IDSanity, and all other objects exist to aid these. The Node contains host information such as what supported softwares it runs, what operating system it runs on, and system information such as hostname, last known IP, and the last time it was seen.

Every node, regardless of ID in the local database on either the slave or the master, can be identified by its *unique_identifier*. The unique identifier is a SHA1 hash generated using a set of hardware based parameters, such as the CPU ID, CPU model, vendor and brand, and the motherboard serial number for Windows and Linux. This hash is then truncated to sixteen characters separated by a dash after every fourth character, creating a unique identifier for each Node, which allows a Master and Slave to uniquely identify each other in their respective local database, and allows the system administrator to uniquely identify a Node even if two nodes, for whatever reason, should end up having the same hostname (i.e in the case of a rogue node attempting to impersonate an existing node on the network to retrieve its set of rules from the master). This value is generated in the HostAnalyzer, and the complete description of how it's generated can be found in 5.12.7 HostAnalyzer

A Node can be banned from communication, or disabled (which should turn) off IDS / Firewall functionality. Only the Master can push these changes to slaves, and these attributes cannot be changed from the slave.

From the Node object, it's possible to retrieve all Rules associated with the Node (and the hash of all these together), and the public key used to verify its signatures.

Only messages sent with a valid signatures will be accepted, any other messages will be ignored.

`current_rule_hash`

The hash of all rules that currently exist or should exist the Node

static `find_or_create_self()`

Looks up the Node for the current system in the database, or creates it if it doesn't exist

Todo Decide which IP is being used by using Communicator.transport

`hash_current_rules()`

Creates a hash from all enabled rules on the node

`hostname`

The hostname of the Node

`id`

The ID and primary key of the Node in the local database

`is_banned`
: Indicates whether or not this Node is banned from communication

`key`
: Relationship pointing to an actual Key object

`key_id`
: The ID and ForeignKey of the Key object representing the Nodes public key

`last_change`
: Last time this Node was synchronized

`last_ip`
: The last known IP this Node had, updated last time it connected

`last_seen`
: Last time this Node connected or disconnected

`node_rules`
: List of NodeRule objects associated with this Node

`operating_system`
: Relationship pointing to an OperatingSystem object representing the Nodes OS

`operating_system_id`
: The ID and Foreign Key of the OperatingSystem this Node has

static `refresh_system_info(`*ignored_param=''*`)`
: Refreshes system information, such as the IP, hostname, and date of last_seen for the system node.
: This method is run on startup

    Parameters `ignored_param` – Discarded parameter, because all events must take at least one parameter

    Returns Node object

`rules`
: Direct list of Rule objects (through nodes_rules) associated with the Node

`softwares`
: List of Software objects associated with this Node - the softwares it runs

`unique_identifier`
: A unique identifier based on hardware information to distinguish nodes

5.11.4 NodeRule

class `idsanity.lib.models.node_rule.NodeRule(`*`**kwargs`*`)`

Bases: *`idsanity.lib.models.model_extension.ModelExtension`*, `sqlalchemy.ext.declarative.api.B`

This is an object implementing Association pattern, as in a many-to-many relationship with additional columns in the relationship. In this case, *date_added* and *enabled*.

NodeRule makes the association between Nodes and Rules, adding information about whether a rule is enabled (and thus exists on file) on a Node, and when it was added there.

NodeRules should not be deleted once they have been created, instead, they should just be disabled as this effectively keeps the changes in the sync messages but causes the change to be reflected on the filesystem.

`date_added`

The date it was first created

`enabled`

Whether or not this Rule is enabled on the Node

`id`

The ID and primary key of the NodeRule

`node`

Relationship pointing to the Node object it belongs to

`node_id`

A reference to the Node it belongs to

`rule`

Relationship pointing to the Rule object it belongs to

`rule_id`

A reference to the Rule it belongs to

5.11.5 Rule

class `idsanity.lib.models.rule.Rule(`*`**kwargs`*`)`

Bases: *`idsanity.lib.models.model_extension.ModelExtension`*, `sqlalchemy.ext.declarative.api.B`

A Rule is a single rule for any supported Firewall or Intrusion Detection System that has been added to the database. Rules are added through one of three ways:

- By the user, manually (either via CLI or through the API)
- Automatically after a sync message was received from the Master
- Automatically after a sync-back message was received from a Slave

Modification of Rules trigger events like ON_RULE_CREATE, ON_RULE_UPDATE and ON_RULE_DESTROY which any module can hook into, with the target Rule passed as a parameter to functions that hook into them.

Modules for the supported IDS/Firewalls are responsible for dissecting a Rule and populating the other fields - by default, only the `date_added`, and `raw` fields are populated apart from the ID.

The Rule object represents a typical Snort-like rule, and OSSEC rules are stored just as raw.

A hope for the future is that by dissecting rules, they can be reassembled for different types of IDS' and it will be possible to translate between different rule flavors.

`action`
    Action to take if this Rule is triggered

`classtype`
    Classtype for this rule

`date_added`
    Date this rule was created

`direction`
    The direction of this Rule, i.e OUTBOUND, INBOUND or BIDIRECTIONAL

`flowbits`
    Flowbits, if any

static `generate_sid()`
    Locates the highest SID currently in the database and returns the next free SID

`gid`
    Group ID

`group`
    The group the Rule belongs to

`id`
    ID and Primary Key of this Rule in the local database

`message`
    Message contained within the Rule

`meta`
    Metadata for the Rule

`priority`
    Priority number

`raw`

Raw representation of the rule as a string. This is what's printed to file!

`rev`

Revision number (version of the Rule). When a Rule is updated, a new Rule should be created with an incremented revision number.

static `set_sid`(*m*, *c*, *target*)

Sets the SID to the next free SID if no SID is defined already

`sid`

Numeric unique identifier - unique across all nodes in the network!

`software`

Relationship pointing to the Software object this Rule has identified as belonging to.

`software_id`

ID and Foreign Key of the Software type this Rule is identified as

5.11.6 Software

class `idsanity.lib.models.software.Software`(***kwargs*)

Bases: *`idsanity.lib.models.model_extension.ModelExtension`*, `sqlalchemy.ext.declarative.api.B`

A Software is a Firewall or Intrusion Detection (+ Prevention) System such as Suricata, Snort or OSSEC. Every Rule in the database should be associated with a Software + version, and every Node should have at least one Software if the system runs a software supported by IDSanity.

When a Node is created, each IDS Modules should attempt to detect if the Software it provides support for exists on the machine, and if it does, create it in the database if it doesnt already exist and associate with the Node.

Whenever a new Rule is added, these Modules should detect whether or not the Rule is valid for that Software, and make an association between the Rule and the Software objects.

It's always possible to retrieve all Nodes or Rules associated with a Software in the database.

`id`

ID and Primary Key of the Software object in the local database

`name`

Name of the Software, i.e 'Suricata'

`nodes`

List of Node objects that has been associated with the Software

`rules`

List of Rule objects associated with this Software version

`software_versions` = **UniqueConstraint()**

`version`

Version of the Software (name + version combination is unique)

5.11.7 OperatingSystem

class `idsanity.lib.models.operating_system.OperatingSystem(`*`**kwargs`*`)`

Bases: *`idsanity.lib.models.model_extension.ModelExtension`*, `sqlalchemy.ext.declarative.api.B`

An OperatingSystem object is merely a representation of an operating system that a node runs. It's likely that most slaves will only contain two records, one for the master and one for itself, since it won't need to know about any others unless one of them changes.

OperatingSystem objects only contain a `family` and an `info` column, and the combination of these is unique.

`family`

The OS family, i.e 'Linux2' or 'OS X'

`id`

ID and Primary Key of the OperatingSystem in the database

`info`

Additional information, such as kernel version, release, and so on

5.11.8 JSON Serializers

Serializers are responsible for turning models into JSON objects. Each serializer has a `serializer` method that takes a model as an argument, and returns it as a JSON object.

If a serializer is instantiated with `many=True`, it can serializer a list of objects.

It's the output of the serializers that are returned from the API, and each model that should be accessible from the API should also have its own serializer.

Serializers subclass `marshmallow.Schema`, which is Marshmallow's base serializer class, and provides some basic serialization functionality.

More often than not, serializers contain the same fields as the model they serialize, but can optionally customize these fields to create the desired output. In idsanity, relationships such as `nodes` for a `Key` object return a list of IDs instead of nested objects, requiring a second query to retrieve the desired objects.

Key Serializer

class `idsanity.lib.serializers.key.KeySchema(`*obj=None, extra=None, only=None, exclude=None, prefix=u'', strict=False, many=False, skip_missing=False, context=None*`)`

Bases: `marshmallow.schema.Schema`

`make_object(`*data*`)`

static `serializer(`*instance*`)`

Node Serializer

class `idsanity.lib.serializers.node.NodeSchema`(*obj=None, extra=None, only=None, exclude=None, prefix=u'', strict=False, many=False, skip_missing=False, context=None*)

Bases: `marshmallow.schema.Schema`

`make_object`(*data*)

static `serializer`(*instance*)

NodeRule Serializer

class `idsanity.lib.serializers.node_rule.NodeRuleSchema`(*obj=None, extra=None, only=None, exclude=None, prefix=u'', strict=False, many=False, skip_missing=False, context=None*)

Bases: `marshmallow.schema.Schema`

`make_object`(*data*)

static `serializer`(*instance*)

Rule Serializer

class `idsanity.lib.serializers.rule.RuleSchema`(*obj=None, extra=None, only=None, exclude=None, prefix=u'', strict=False, many=False, skip_missing=False, context=None*)

Bases: `marshmallow.schema.Schema`

`make_object`(*data*)

static `serializer`(*instance*)

Software Serializer

class `idsanity.lib.serializers.software.SoftwareSchema`(*obj=None, extra=None, only=None, exclude=None, prefix=u'', strict=False, many=False, skip_missing=False, context=None*)

Bases: `marshmallow.schema.Schema`

`make_object`(*data*)

static `serializer`(*instance*)

OperatingSystem Serializer

class `idsanity.lib.serializers.operating_system.OperatingSystemSchema(`*obj=None, extra=None, only=None, exclude=None, prefix=u'', strict=False, many=False, skip_missing=False, context=None*`)`

Bases: `marshmallow.schema.Schema`

`make_object(`*data*`)`

static `serializer(`*instance*`)`

5.12 Library

The `lib` folder contains internal classes for IDSanity, used for managing communication between master and slave, connection to the database, event management, the command line interface and API, and everything else that is managed internally.

The `lib/core` directory contain most of the core files that aren't exposed to the outside other than through the CLI or API, whereas the `lib/models` directory contains all models that allow for interacting with the database through the ORM layer.

Two factories are to be found in the core directory as well, and these factories are responsible for accepting settings and then loading the appropriate daemon class to instantiate - there's one factory for each mode IDSanity can run in, thus, there's one for slave, and one for master.

Each factory also dispatches events, such as the common *ON_INIT*, but depending on the mode IDSanity is running in, additional mode-specific events may also be dispatched by the factories. The important thing to note is that a factory does nothing but set up everything required for the daemon to begin operating.

5.12.1 Singleton Metaclass

class `idsanity.lib.core.singleton_metaclass.Singleton(`*name, bases, dict*`)`

Bases: `type`

This is a metaclass that allows the module loader and events class to follow the singleton pattern - that is, only ever create *one* instance, and always return the same instance if it's instantiated, since there should never be a need for multiple module loaders, and a module loader should be accessed without reinstantiation. The same thing applies for the Events class, where multiple instances would lead to some events not being accessible from the same object as the others.

5.12.2 Factory (master)

class `idsanity.lib.core.factory.IdsanityMasterFactory(`*settings*`)`

Bases: `twisted.internet.protocol.Factory`

This factory sets up everything required to initialized the app, and is called by the service maker to daemonize IDSanity.

It's responsible for loading the module loader and thus make sure all modules are loaded, and use the settings passed to it to set up the database, and create a Node object for the current system if one doesn't already exist.

Events such as `ON_CONNECT`, `ON_INIT` and `ON_NODE_CREATE_SELF` must be registered here to ensure they can be dispatched when they occur.

`protocol`

alias of `IdsanityMaster`

5.12.3 Factory (slave)

class `idsanity.lib.core.factory_slave.IdsanitySlaveFactory`

Bases: `twisted.internet.protocol.ClientFactory`

This is the factory that sets up IDSanity in *slave mode*. It essentially does the same thing as the master factory, except instead of listening for connections, it connects to the specified IP and port.

It's initialized by the service maker when the client is daemonized after starting up, and spawns an instance of the IDSanity client daemon.

It's responsible for loading the module loader and thus make sure all modules are loaded, and use the settings passed to it to set up the database, and create a Node object for the current system if one doesn't already exist.

Events such as `ON_CONNECT`, `ON_INIT` and `ON_NODE_CREATE_SELF` must be registered here to ensure they can be dispatched when they occur.

`clientConnectionFailed(`*connector*, *reason*`)`

`clientConnectionLost(`*connector*, *reason*`)`

`protocol`

alias of `IdsanitySlave`

5.12.4 Communicator

class `idsanity.lib.core.communicator.Communicator`

Bases: `object`

Manages all communication between master/slave, by filtering all messages sent and received through its static methods.

This is where encryption and decryption on messages sent and received internally by IDSanity should be managed, as well as signature creation and verification.

The following events are registered here:

- ON_NODE_RULE_DISABLE
- ON_NODE_RULE_ENABLE
- ON_NODE_SYNC_NOOP
- ON_NODE_SYNC_SEND
- ON_NODE_SYNC_RECEIVE
- ON_NODE_SYNC_BACK
- ON_NODE_UPDATE
- ON_RULE_REQUEST_SID
- ON_RULE_RECEIVE_SID
- ON_SYSTEM_MESSAGE_RECEIVE
- ON_SYSTEM_MESSAGE_SEND
- ON_SYSTEM_MESSAGE_ERROR
- RULE_SID_REQUEST

`last_sent = ''`

static `receive`(*msg*, *transport=None*)

Parses a received message (i.e decrypts it) and returns the results.

This function automatically calls Communicator._parse on the message.

This function can, if a transport is provided, reply via .send()

Parameters

- `msg` – The received message
- `transport` – Optional transport to write reply to

Returns Parsed message as dict

static `request_sids`(*amount=0*)

This triggers a slave to immediately send a request for a certain number of SIDs from the master.

Once it receives SIDs, it should update rules in the database that do not have a SID assigned.

Params amount Amount of SIDs to request from master

static `send`(*msg*, *transport=None*)

Sends a message to the other end of the connection.

If a twisted transport is provided, it will write directly to the transport, otherwise it returns the prepared message.

Parameters

- `msg` – Message to send
- `transport` – Optional transport

static `shake_hands()`

Begin handshake by sending an ‘identify’ message to the other end of the line. The node at the other end will receive this message, parse it, and create an ‘identification’ message as a response.

This message will be parsed, and the node will be added (or updated) in the database.

static `slave_request_sync()`

Called from the slaves end to begin synchronization, by creating a ‘verify’ message containing the current rule hash, and sending it to the Master.

When the Master receives it, it will check if the hash matches, and either reply with a node sync_noop message, or a node sync message containing all the rules to be synchronized.

This function should be called every X minutes from the slave.

5.12.5 ModuleLoader

class `idsanity.lib.core.module_loader.ModuleLoader(`*names=[]*`)`

Bases: `object`

Loads all modules dropped into the modules/ folder, using baseclass BaseModule. These are stored in modules, which is effectively static.

Can be accessed anywhere by ModuleLoader.modules

`MODULE_PATH` = **‘../../modules/’**

The path where modules to be imported are located. This should be made crossplatform by implementing an absolute path

`instance` = **None**

`load_modules()`

Loads all modules and assigns them to indexes in ModuleLoader.plugins

Return list All loaded modules

`plugins` = **[]**

Contains instances of each module

`whitelist` = **[]**

List of which modules to load. These are retrieved from the config file.

5.12.6 HostAnalyzer

class `idsanity.lib.core.host_analyzer.HostAnalyzer`

Bases: `object`

Gathers information about the system, to be used when creating a Node for it

static `directory_exists(*args)`

Check if dir(s) exists on the host

Parameters `*args` – Path to dir(s) as one or more string arguments

Returns List of existing directories or False

static `file_exists(*args)`

Check if file(s) exists on the host

Parameters `*args` – Path to file(s) as one or more string arguments

Returns List of existing files or False

`generate_key(bits=4096)`

Generates a key pair for the Node

Returns private key, public key

`generate_uuid()`

Generate a UUID based on BIOS and hardware information for the system. This unique identifier will be used to identify a host uniquely.

The UUID is a SHA1 hash of:

- CPU Family number
- CPU Model number
- CPU brand as a string
- CPU vendor as a string
- Motherboard / Chassis serial number

The hash is truncated to 16 characters, and separated by dashes after every 4th letter.

Example: 8915-4f39-1aa1-1250

Returns String representation of a unique ID

`get_dmi()`

Retrieves DMI information for the system and returns it. This should work on OS X, Windows and Linux.

Todo Make this crossplatform by adding a proper call to dmidecode.exe

Returns DMI contents as string

`get_hostname()`

Retrieves the hostname

Todo Make cross-platform!

`get_ip()`

Retrieves the current IP address(es) for each interface

Returns Dict with IP-address(es) of each NIC on the host

`get_key()`

Checks for a key pair in the conf directory, and returns the public key.
If no key exists, calls self.generate_key()

Returns The public key for this system as a string

`get_os_architecture()`

Find out whether the system is x86 or x86_64

Returns Architecture as str (x86 or x86_64)

`get_os_family()`

Detect the OS family (Linux, Mac, Windows)

Returns OS Family as str

`get_os_info()`
: Get all information about the OS in a dict

Note Does NOT return private key!

Returns dictionary `self.info` with OS information

`get_os_kernel()`
: Get the current kernel of the host system as a string

Returns str representation of kernel

`get_os_version()`
: Get the current operating system platform version

Returns OS version in str form

`get_unique_identifier()`
: Generates a unique ID based on system information

Returns ID for this system

static `process_exists(*args)`
: Check if a process is running with a name like any of the arguments

Parameters `*args` – Process name(s) as one or more string arguments

Returns List of processes and PIDs, or False

5.13 Modules (plugins)

Modules for IDSanity all subclass BaseModule, which should provide some basic functionality for modules.

As of now, it only provides the class variable `events`, providing access to register, hook into and dispatch events.

Modules may have a method called `register_events`, and if it does, it will be executed as soon as the module is loaded when IDSanity starts up to ensure that events are registered before they can be dispatched.

If a module has any new events to register, they should be registered in `register_events`.

Writing new modules is easy - simply subclass `BaseModule` and hook the desired methods into the events you'd like it to listen for.

5.13.1 IDS Modules

A typical usecase is adding new ID(P)S softwares to IDSanity, since all supported softwares are represented by a module.

Software/IDS modules are required to provide a `detect` method, and a `parse` method.

Detecting software

The `detect` method *must* hook into `ON_DETECT` and take a an instance of `Node` as an argument. It should then perform necessary checks on the system, to determine if the software it should detect exists.

`lib.host_analyzer.HostAnalyzer` provides some useful methods for this, namely `HostAnalyzer.file_exists()`, `HostAnalyzer.directory_exists()` and `HostAnalyzer.process_exists()` which all take one or multiple strings as arguments and returns what it finds or `False` if it finds nothing.

If the software exists on the system, it *must* try to locate it in the database, or create it if it doesn't exist, and append the software to the Node's list of softwares.

Parsing & Validating rules

Each IDS module must also be able to parse and validate rules for the software its meant for.

The Rule-parsing method *must* hook into both `ON_RULE_CREATE` and `ON_RULE_UPDATE` and take a Rule object as an argument. If the Rule is a valid Rule for the software, it should assign any fields except for `raw` on the Rule, and set the `software` attribute of the Rule to the right software.

The Rule-validation method *must* take a raw string as argument, and return False if the Rule is not a valid rule for the software, or the parsed Rule if it is.

BaseModule

class `idsanity.modules.base_module.BaseModule`

Bases: `object`

Base Module to build all other modules on, to create a module, create a file of the same name and subclass BaseModule.

`any_method()`

`events` **= <lib.events.Events object>**

A reference to the Events instance to be accessed by any submodules

static `register_events()`

Events that need to be registered should be registered in the `register_events` method, so override this method and call `self.events.register('your_event_name')` in this function.

SayHello (example module)

class `idsanity.modules.say_hello.SayHello`

Bases: *`idsanity.modules.base_module.BaseModule`*

Sample module that only prints "HELLO" when a connection is made

`hello(`*parameters*`)`

ModSecurity

class `idsanity.modules.modsecurity_module.ModSecurityModule`

SurricataModule detects running instances of mod_security and parses rules.

`detect_modsecurity()`

Detect if mod_security is enabled and running on the node.

Returns True if detected, False otherwise

`parse_rule()`

Parse mod_security rule and returned the parsed rule.

Parameters `rule_dict` – Rule to verify

Returns None or dict with parsed rule

`validate_rule(`*r*`)`

Validates a mod_security rule to see if it's valid.

Parameters `r` – Rule string to parse (string)

Returns False or parsed rule

Snort

class `idsanity.modules.snort_module.SnortModule`

Bases: *`idsanity.modules.base_module.BaseModule`*

SnortModule manages detection of running Snort instances on the machine and parsing of Snort rules.

Hooks into ON_RULE_CREATE and ON_RULE_UPDATE to parse and update rule with dissected fields.

`detect()`

Attempts to find out if Snort is installed on this machine by first checking running processes, and then falling back to looking for common files

`dissect_rule(`*r*`)`

Parse a rule and update it with dissected fields

Parameters `r` – Rule object to verify as Snort rule

`validate_rule(`*raw*`)`

Validates a rule by attempting to parse it. If it comes out as None, it's invalid and False is returned.

Parameters `r` – Rule string to parse

Returns False or parsed rule

Suricata

class `idsanity.modules.suricata_module.SuricataModule`

Bases: *`idsanity.modules.base_module.BaseModule`*

SuricataModule detects running instances of Suricata and parses rules.

`detect_suricata()`

Tries to detect if Suricata is running.

Returns True if detected, False otherwise

`get_pid(`*procname*`)`

`parse_rule(`*rule_dict*`)`

Parse a rule object and update it.

Parameters `rule_dict` – Rule to verify

Returns

`validate_rule(`*raw_rule*`)`

Validate rules by using the parse method from idstools.

Parameters `raw_rule` – Rule to parse (string)

Returns Parsed rule or False

6 Testing

Tests are written using a combination of Twisted's extension of python's built-in unit tests, called *twisted.Trial*, and the python testing framework *py.test*. Each folder, such as the `lib` and `models`, have a subdirectory dedicated for tests aptly named 'tests'. All tests in these folder begin with the name *test_* followed by what they test, `test_connection.py` which tests the connection to a master daemon.

There are often some files in these directories that **do not** begin with the test prefix, and these are base classes which provide functionality for the other tests which subclass these. For example, the base class for database tests ensure a database is set up for testing and that changes can be rolled back after the tests are performed.

Each function test a specific functionality and provide appropriate error messages should a test fail. To execute tests, the *PYTHONPATH* environment variable must include the root directory of IDSanity, and for Unix based systems with a **sh**ell, there's a shell-script called `run-tests.sh` in the root directory of IDSanity that ensures this is done before executing the tests.

6.1 Model Tests

Database Test base class

class `idsanity.lib.models.test.database_test_case.DatabaseTestCase(`*methodName='runTest'*`)`

Bases: `twisted.trial._asynctest.TestCase`

A base test case for tests connecting to the database. Provides access to session variable and makes sure any changes made are rolled back after the test is complete.

`setUp()`

`tearDown()`

Key Test Case

class `idsanity.lib.models.test.test_key.KeyTestCase`(*methodName='runTest'*)

Bases: *`idsanity.lib.models.test.database_test_case.DatabaseTestCase`*

`test_create_event()`
Ensure the CREATE event has been dispatched for this resource

`test_destroy_event()`
Ensure the DESTROY event gets dispatched for this resource

`test_save()`
Makes sure saving the object works

`test_update_event()`
Ensure the UPDATE event gets dispatched for this resource

Node Test Case

class `idsanity.lib.models.test.test_node.NodeTestCase`(*methodName='runTest'*)

Bases: *`idsanity.lib.models.test.database_test_case.DatabaseTestCase`*

`node_rule_event()`
Ensure the CREATE event has been dispatched for this resource

`test_create_event()`
Ensure the CREATE event has been dispatched for this resource

`test_destroy_event()`
Ensure the DESTROY event gets dispatched for this resource

`test_node_key_relationship()`
Ensure that a relationship exists in the database between Node and Key after being created on the model.

`test_node_rule_relationship()`
Ensure that a relationship between a Node and a Rule exists in the form of a NodeRule object and that it can be found in the database after creating a new Rule for the node.

`test_save()`
Ensure that saving a model after modifying its attributes actually updates the information in the database.

`test_update_event()`
Ensure the UPDATE event gets dispatched for this resource

Operating System Test Case

class `idsanity.lib.models.test.test_operating_system.OperatingSystemTestCase(`*methodName='runTes*

Bases: *`idsanity.lib.models.test.database_test_case.DatabaseTestCase`*

`test_create_event()`

Ensure the CREATE event has been dispatched for this resource

`test_destroy_event()`

Ensure the DESTROY event gets dispatched for this resource

`test_save()`

Makes sure saving the object works

`test_update_event()`

Ensure the UPDATE event gets dispatched for this resource

Rule Test Case

class `idsanity.lib.models.test.test_rule.RuleTestCase(`*methodName='runTest'*`)`

Bases: *`idsanity.lib.models.test.database_test_case.DatabaseTestCase`*

`test_create_event()`

Ensure the CREATE event has been dispatched for this resource

`test_destroy_event()`

Ensure the DESTROY event gets dispatched for this resource

`test_save()`

Makes sure saving the object works

`test_update_event()`

Ensure the UPDATE event gets dispatched for this resource

Software Test Case

class `idsanity.lib.models.test.test_software.SoftwareTestCase(`*methodName='runTest'*`)`

Bases: *`idsanity.lib.models.test.database_test_case.DatabaseTestCase`*

`test_create_event()`

Ensure the CREATE event has been dispatched for this resource

`test_destroy_event()`

Ensure the DESTROY event gets dispatched for this resource

`test_save()`

Makes sure saving the object works

`test_update_event()`

Ensure the UPDATE event gets dispatched for this resource

6.2 IDSanity Library Tests

Base class for tests that require a connection

This is the base class for all tests that require communication between master and slave. It does this by setting up a "mock" client that communicates with the other end, and allows sending and receiving messages.

All tests that require a connection should subclass this and use the functions it provides.

class `idsanity.lib.core.test.connection_test_case.ConnectionTestCase`(*methodName='runTest'*)

Bases: `twisted.trial._asynctest.TestCase`

Test the server functionality.

`setUp()`

Reads database configuration from the `settings.test.conf` file with connection parameters to the test database, and provides functions to send and receive messages.

This function will be run automatically when the class is instantiated.

Communicator Test Case

class `idsanity.lib.test.test_communicator.CommunicatorTestCase`(*methodName='runTest'*)

Bases: `lib.test.connection_test_case.ConnectionTestCase`

Test the communication between Master and Slave

`test_node_update()`

Tests so that sending a message to update a Node's metadata actually updates it in the database. The message is signed and the signature will be verified before applying the update.

Should also test so that an invalid signature causes no change.

`test_receive()`

Test so that messages can be received, parsed and reacted to by the Master

`test_send()`

Test so that messages can be sent

`test_sign_message()`

Ensures a signature is added to a message after calling Communicator._sign_message() with a dict parameter

`test_verify_signature()`

Ensure a signature can be verified using its public key from the database

Communicator Test Case

Connection Test Case

class `idsanity.lib.core.test.test_connection.ConnectTestCase(`*methodName='runTest'*`)`

Bases: *`idsanity.lib.core.test.connection_test_case.ConnectionTestCase`*

`test_connected()`

Ensure that a connection is possible

`test_event_dispatched()`

Ensure that the Events().last_event is set after a message was sent, and that the `ON_SYSTEM_MESSAGE_SENT` was dispatched.

Events Test Case

class `idsanity.lib.core.test.test_events.EventsTestCase(`*methodName='runTest'*`)`

Bases: `twisted.trial._asynctest.TestCase`

Test the functionality of the Events class

`setUp()`

Set up the Events class for use in all test functions

`test_dispatch()`

Test that a hook is properly dispatched when dispatch is run on that event.

`test_event_value_pair()`

Make sure a proper event value pair is returned from Events().event_value_pair('event_name')

`test_exists()`

Make sure the existence of an event can be reliably checked with the exists method

`test_hook()`

Test that an event hook is properly registered after .hook and thus able to be dispatched.

`test_next()`

Ensure the Events() class is iterable

`test_register()`

Make sure registering an event works and the event exists in the event dictionary

`test_unregister()`

Make sure unregistering an event works and it's properly removed after it has been unregistered

`test_unregister_unknown_event()`

Make sure unregistration of an event that has not been registered fails gracefully

`verify_dispatch(`*args*`)`

Method that will be run when `test_dispatch()` dispatches the event, used to verify that hooking into events actually works

HostAnalyzer Test Case

class `idsanity.lib.core.test.test_host_analyzer.HostAnalyzerTestCase(`*methodName='runTest'*`)`

Bases: `twisted.trial._asynctest.TestCase`

Test the functionality of the Host Analyzer

`setUp()`

`test_hostname()`

Test that a valid hostname is returned

`test_ip()`

Test that valid IPv4 or IPv6 addresses are returned for each interface

`test_os_architecture()`

Test that valid system architecture information is returned

`test_os_family()`

Test that valid OS family information is returned

`test_os_hostname()`

Test that the hostname is identical to the current machine's hostname

`test_os_info()`

Test that valid operating system information is returned

`test_os_kernel()`

Test that valid kernel information is returned

`test_os_version()`

Test that valid OS version / release information is returned

`test_unique_identifier()`

Test generation of unique identifier and that it is returned in the correct format.

ModuleLoader Test Case

class `idsanity.lib.core.test.test_module_loader.ModuleLoaderTestCase(`*methodName='runTest'*`)`

Bases: `twisted.trial._asynctest.TestCase`

Test the functionality of the Module Loader

`setUp()`

`test_load_modules()`

Test plugins are loaded properly

`test_load_modules_instances()`

Make sure module loader has a list of module instances

`test_load_modules_whitelist()`

Test whether modules are properly white-listed in configuration

7 Conclusion

7.1 Missing Functionality

As said previous in this report, IDSanity is meant to be a prototype and not a fully functional application ready for a production environment.
Because of this there are some functionality that is missing or not yet completed. This section will describe the missing funtionality in IDSanity.

Enable/Disable Node

During the planing phase we discussed the pos.sibilities of adding a way of enabling and disabling a running node. This was never fully implemented but should be easy to implement in the future if this functionality is needed by the system administrators.

Ban/Unban Node

This was never implemented due to the time frame and the fact that other, more important, features of IDSanity was prioritized.

Rule Feeds

We discusses the possibility of adding a way to subscribe to rules feeds from third-party vendors, but never implemented this feature. Rules feeds are basically just plain-text available on the internet, so this is also something that should be fairly simple to implement in the future if needed.

Node Nickname

As of now each node gets a unique ID generated by creating a hash of hardware IDs etc. This isn't the best way to distinguish between running nodes, but sadly *nicknames* was never implemented in IDSanity.

LDAP

This is probably the most missed functionality as of today. LDAP was also a part of the design specification, but as the project grew in complexity, LDAP was pushed further and further away from the actual implementation plan of IDSanity. Python however has great support for LDAP, so this should not be a problem to implement in the future if needed.

Database

Even though the database is up and running and some information is saved into the database, full rule sets are not yet saved.
Rule sets are dissected and parsed and is ready to be inserted into the database, but this functionality was never finished.

BaseModule

Modules that inherit from BaseModule does not get as much functionality as it should.

7.2 Discussion

This last section is a discussion on a personal level on our thoughts about the project and what we have learned.
The combination of a new programming language and the amount of technologies we weren't familiar with, turned out to be a bigger challenge than our initial thoughts. We succeeded and we failed on a weekly basis. We underestimated and overestimated. We learned, forgot, and then rapidly learned again when the same mistakes hit us right back in the face. And now, at the very end of a long, but interesting project, we're left standing with some mixed feelings.

The Good: We, all three members of the group, has learned extreme amounts of new information and skills throughout this project. A whole new programming language is now in our repertoire, and we can use and manage applications and systems which are some of the most relevant programs for our future careers in information security.
One of our highest rated threats in the weighted threat analysis we performed was the possibility that the project would be too difficult. Even though there was some truth to this, we managed to get get through the project without too big of problems, and did not have to change to a different project.

The Bad: This was the biggest project we've had so far in our lives, in such a small time frame. We dramatically underestimated exactly how much work that had to be done to transform this project from the basic idea, to a working system. However, it's not all bad. Even though we didn't manage to implement all the desired functionality our employer and our imagination came up with in the beginning, we did manage to create a working prototype of the framework. And just by accomplishing that, we are all quite satisfied.

The Ugly: We learned about some minor character flaws in ourself that only such a big group project could get out. However, by being able to work though those flaws we where able to grow and emerge even stronger. One of the biggest mistakes we did was in the beginning of the project was that we did not test our theories before deciding to go for them, so ideas that looked good on paper, did not automatically work as well in reality. This was a result of lack of knowledge of all parts of the project, and how they worked and interacted with each other.

Throughout the entire project, we discussed and reflected about every wall we hit, and always came up with solutions. Some of the solutions worked, some did not. But no matter what, we learned how to handle difficult situation in an environment where every member is dependent on each other. We went in motivated, and came out on the other side even more so.
All in all we learned something from every aspect of this project, and we will cherish our newly toned skills for the years to come.

Bibliography

[1] Sikkerhetsråd, N. March 2014. Mørketallsundersøkelsen. http://www.nsr-org.no/getfile.php/Dokumenter/NSR%20publikasjoner/M%C3%B8rketallsunders%C3%B8kelsen/M%C3%B8rketall_2014.pdf.

[2] NIST. February 2007. Guide to intrusion detection and prevention systems. http://csrc.nist.gov/publications/nistpubs/800-94/SP800-94.pdf.

[3] Infotech, R. 2015. Intrusion detection & prevention system. http://www.rubikinfotech.com/wp-content/uploads/2014/09/intrustion.jpg.

[4] Institute, I. 2013. Network design firewall ids/ips. http://2we26u4fam7n16rz3a44uhbe1bq2.wpengine.netdna-cdn.com/wp-content/uploads/040813_1643_NetworkDesi2.jpg.

[5] McAfee. March 2003. Deciphering detection techniques: Part ii anomaly-based intrusion detection. http://www.mcafee.com/japan/products/pdf/Deciphering_Detection_Techniques-Anomaly-Based_Detection_WP_en.pdf.

[6] Micro, T. 2015. Ossec how it works. http://www.ossec.net/?page_id=169.

[7] Micro, T. 2015. Ossec features. http://www.ossec.net/?page_id=165.

[8] Micro, T. 2015. Ossec architecture example. http://www.ossec.net/wp-content/uploads/2012/04/ossec-arch2-1024x586.jpg.

[9] Cid, D. B. 2007. Ossec log analysis. http://www.ossec.net/files/auscert-2007-dcid.pdf".

[10] Trustwave Holdings, I. May 2015. Modsecurity reference manual. https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual#Introduction.

[11] VPS, S. 2014. How to install & configure mod_security on cpanel/whm vps. https://www.solvps.com/blog/how-to-install-mod-security-cpanel-vps/".

[12] Aanval. 2008. Aanval live event correlation. https://www.aanval.com/images/screenshots/7.0/15.png.

[13] Chaos, D. 2013. Snorby listing sessions. http://www.drchaos.com/wp-content/uploads/2013/11/Snorby-Event-Screen.png.

[14] DB-Engines. May 2015. Trend of postgresql popularity. http://db-engines.com/en/ranking_trend/system/PostgreSQL.

A Project Proposal

Problemstilling for bacheloroppgaver ved IMT våren 2015

Oppdragsgiver

Oppdragsgiver: Høgskolen i Gjøvik, IT-tjenesten

Kontaktperson: Christoffer Hallstensen

Addresse: Teknologiveien 22, 2815 Gjøvik

Telefon: 61135145 / 48135180

Epost: christofferh@hig.no

GUC Security Rules management

Utvikle et rammeverk for et sentralisert web/kommandolinje basert verktøy for å håndtere HIDS, NIDS, IPS, WAF og system audit policy regler.

IT-tjenesten har en stund arbeidet med å implementere bedre deteksjon av sikkerhetsrelevante hendelser. I et distribuert system kan det fort bli uoversiktlig og vanskelig å ha kontroll på versjoner av regelsett og policyer samt hvilke klienter som har hvilke regelsett og om de er policy compliant.

Beskrivelse av oppgaven

Dette prosjektet går ut på å:

- Utvikle et modulært rammeverk for effektiv distibuering og håndtering av regelsett for IDS og WAF
- Utvikle et web interface for enkel oversikt

Denne oppgaven passer for Informasjonssikkerhet og Programvareutvikling

Studenten vil gjennom prosjektet få erfaring innen

- Utvikling av programvare
- Systemutvikling
- Åpen kildekode systemer for systemsikkerhet
- Katalogtjenester

B Preliminary Project

Project Plan

Preliminary Project For The Bachelor Thesis

Authors:
Tommy B. Ingdal
Halvor Mydske Thoresen
Victor Rudulfsson

February 13, 2015

Contents

1 Goals And Limitations

1.1 Background

The IT department at Gjøvik University College has been working on implementing improved detection of security related events in their network.
The systems that are going to be implemented to achieve this, are complex and has a tendency to become difficult to manage.

1.2 Project Goal

The goal of the project is to simplify the distribution and management aspect of the new intrusion detection and prevention, web application firewall and system audit policy systems by developing tools that keeps track of the ruleset and policy versions on the different sensors. These tools will also ease the process of deploying and editing those rules and policies.
The end product will be a fully functional and modular framework which is accessable by both a commandline interface and a simple web interface. The product should be made in such a way that it can be extended by user-made modules with relative ease. To achieve this, all code will follow well known standards and be sufficiently documentet.

1.3 Boundary

The tools developed during this project should be of interest for other institutions than the current employer, but because of limited resources and time, the primary focus will be on developing the best possible product tailored for the IT department at GUC.
The project will also be limited to the use of PHP, Python, C/C++ and HTML, CSS and Jacascript as development languages because of the already existing systems at GUC. The application must also be able to function on CentOS 6.5 or newer, or Debian 7.x, and the use of databases are limited to PostgresSQL 9.2.x.

2 Scope

2.1 Limitations

This application will not be able to handle all the different HIDS, NIDS and WAFs that exist out-of-the-box. The application will however be modular, which means that IT-tjenesten will be able to add their own modules if they wish to extend the application.

The use of closed-source libraries will be kept at a minimum or avoided all together. The reason for this is to avoid the problems that may arrise if a closed-source system or library is updated or changed, and potentially causing problems for our application.

2.2 Project Description

The IT department at Gjøvik University College is in need of a centralized and modular tool to ease the distribution and management of HIDS, NIDS, IPS, WAF and system audit policy rulesets. The tools should be controlled by the use of command line and a simple web interface.
During the course of the project, the group members will get increased experience in software development, system engeneering, distribution systems and working with the security aspect of open-source systems.

3 Project Organization

3.1 Responsibilities And Roles

For the Bachelor Thesis our employer is Gjøvik University College / IT Departement and we have two supervisors, namely Stewart Kowalski and Thomas Kemmerich.
Our group consists of three members: Tommy B. Ingdal, Halvor Mydske Thoresen and Victor Rudolfsson. All of which are students in Information Security.

Tommy B. Ingdal is our elected group leader and will communicate with the supervisors and employer on behalf of the group.
Even though we have elected a group leader, we do have a democratic group system and the group has to vote if there's a disagreement. The group leader is only there to ensure that the supervisors, employer and group members can have a easy way of communicating.
All the group members have worked together on projects before and know each other very well. This is an advantage since we know the work flow of each other and trust each member to finish the task he has been given.
Each member have also signed a group contract to ensure that each member know the rules in the group. You can find this contract under Appendix A - Group Contract.

3.2 Group Procedures And Rules

See Appendix A - Group Contract

4 Planning, Follow-Up And Reporting

4.1 Division Of Project

We have to decided to divivde the project in two parts. The first, and most important part, is to finish the development of the core component(s) of the application (i.e. the API which other systems and the web front-end can communicate with). The second part of the project is to develop the front-end, or the web application, the administrators will use to manage the rule sets for the different sensors on the network.

The most difficult, and time consuming part, is the development of the API. We expect to be working on this part until the end of April, but hopefully we will finish this task sooner than expected. As soon as we finish this part, all group members will be focused on the front-end development, as well as the report it self.

Even though this is a development project, we also have to work on the project report during the development. During the meetings we will have each week, we will discuss what needs to be added to the report, as well as delegating different tasks related to the project report.

4.2 Status Meetings And Decision Points

To ensure consistant feedback and that we are doing everything correct, we wish to have status meetings with our supervisor(s) every week. Preferably at the start of the week.

At the status meetings we will show where we are in the development lifecycle by using the Gantt diagram, and what we have done with the report since last time. We will also present a plan for the coming week (i.e. which tasks we hope to complete, both in regards of the application it self and the report).

5 Quality Assurance

5.1 Documentation, Standards And Source Code

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Figure 1: PEP-20 - The Zen of Python

Quality should be assured by means of four basic practices:

- *Documentation*,
- *Testing*,
- *Modularity* and
- consistent use of defined *Standards*.

"A style guide is about consistency. Consistency with this style guide is important. ***Consistency within a project is more important.*** *Consistency within one module or function is most important."*

Figure 2: PEP-8 - Style Guide for Python Code

5.1.1 Documentation

To make the code readable and easy to modify in the future, it's not **only** a good idea, or a requirement, but an incredibly bad choice to **not** document code. Therefore, all code written should be documented both using Python's built-in, standardized *docstrings* practice.

These can then be accessed either through the REPL, using the magic **function.__doc__** but also by using automated documentation tools like *Sphinx*.
Using docstrings with Python in conjunction with Sphinx allows for a native replacement to regular javadoc.

This only makes sense if used consistently - and as mentioned previously, this is one of the core pillars of Python.

Documenting source code at the same time as it's written is required, not only to reach the desired quality level in the end product, but because even if the code is just a draft, and it's *"intended to be rewritten later"*, if the code in itself is not clear enough to other people than the author, it must be commented to allow refactoring, modification or improvement by others.

5.1.2 Standards

Python standards are very clear, providing common guidelines in the PEP-8 (Style Guide) and PEP-20 (The Zen of Python). These shall be adhered to to as high degree as possible.

Although it's a common practice to use CamelCase and name functions according to the programmers own abbreviations (such as *bob.getUID()* for a function that returns the user ID) this can end up messy when such abbreviations aren't obvious (is bob.getUID()

returning the users ID? or is UID some inherited attribute like ***U**nique **Id**entifier*, inherited by all objects of the same base class?) to other programmers without double-checking. Therefore, function names should be named ***according to what they do** in a **readable manner***, in lowercase with words separated by underscores. That one extra second you may have saved by making the name shorter is paid for with reduced clarity - instead, a function to retrieve the ID from a user object would be unmistakingly named *bob.get_user_id()*.

Now you might be thinking *"You fool! Why use getters and setters when nothing is private in Python?"*, in which case you may rest assured it was merely for explanatory purposes; which brings us to *attributes* and *variables*. Similarly to methods, these should **always** be named by the same principle: snake_case, with clarity and reabability in mind.

"Don't build a house on sand"

Furthermore, there's **no shame** in refactoring. On the contrary, refactoring is *always* welcome once code becomes messy and big - *even though this is not necessarily part of SCRUM*. Although it may take up precious time, it's often necessary to maintain a good quality as the code base grows. A house needs to be built on a firm foundation; continuing coding on a loose foundation may end up costing more time in the end than necessary, and just like with writing a book, *don't be too attached to the first draft!*.

5.1.3 Testing

Tests should **always** be written on the *system/functional* level, and *integration* level, meaning tests that probe functionality and interaction between units (classes). Unit testing is not mandatory (but recommended!) as the internals of a class may change without breaking functionality (but causing a failing test). Therefore, tests should probe functions exposed to and accessed by other classes to ensure functionality remains intact.

Furthermore, it's generally a good idea to remember that if a function deserves and/or requires an elaborate explanation in its block comment, then a test should be written for it.

5.2 Configuration Management

Certain administrative overhead is necessary for maintaining, reviewing/analysing and storing the source code in a good manner. As such, use of a version control system is mission critical, and it's been agreed unanimously that *git* (and *github*) will be used for this purpose.

This comes with a big responsibility that is easy (or convenient) to ignore, and because of it's importance it will be repeated: Commit messages ***must** be provided*, ***must** be descriptive*, and ***must** be less than 80 characters*.

5.2.1 Git workflow

Developing with git requires certain mutually agreed-upon practices, and a good workflow is required. All code written will be developed in their respective *branches*, and **only** tested, functional code will make it into the master branch.

Three default branches will be used, together with additional branches created as the need arises:

- **Master**: Production branch containing *latest version* of functional, tested, production-ready code.
- **Hotfixes**: Contains quick bug-fixes for previously undetected bugs discovered in Master branch.
- **Release branches**: Not a branch in itself, but a branch for each release. This is useful to access the code for each individual release as it was when it was merged into Master, even if it's not the most recent release.
- **Development**: Contains code that is being worked on, with each new feature being developed in..
- **Feature branches**: A branch for each new feature, that is then merged back into the development branch and subsequently into its release branch, until it's ready to go into Master as the latest version.

Code that has been merged into the current Release branch is ready to be tested, after which it can be merged into Master. If this sounds confusing, this diagram by *Vincent Driessen* may help visualize the process:

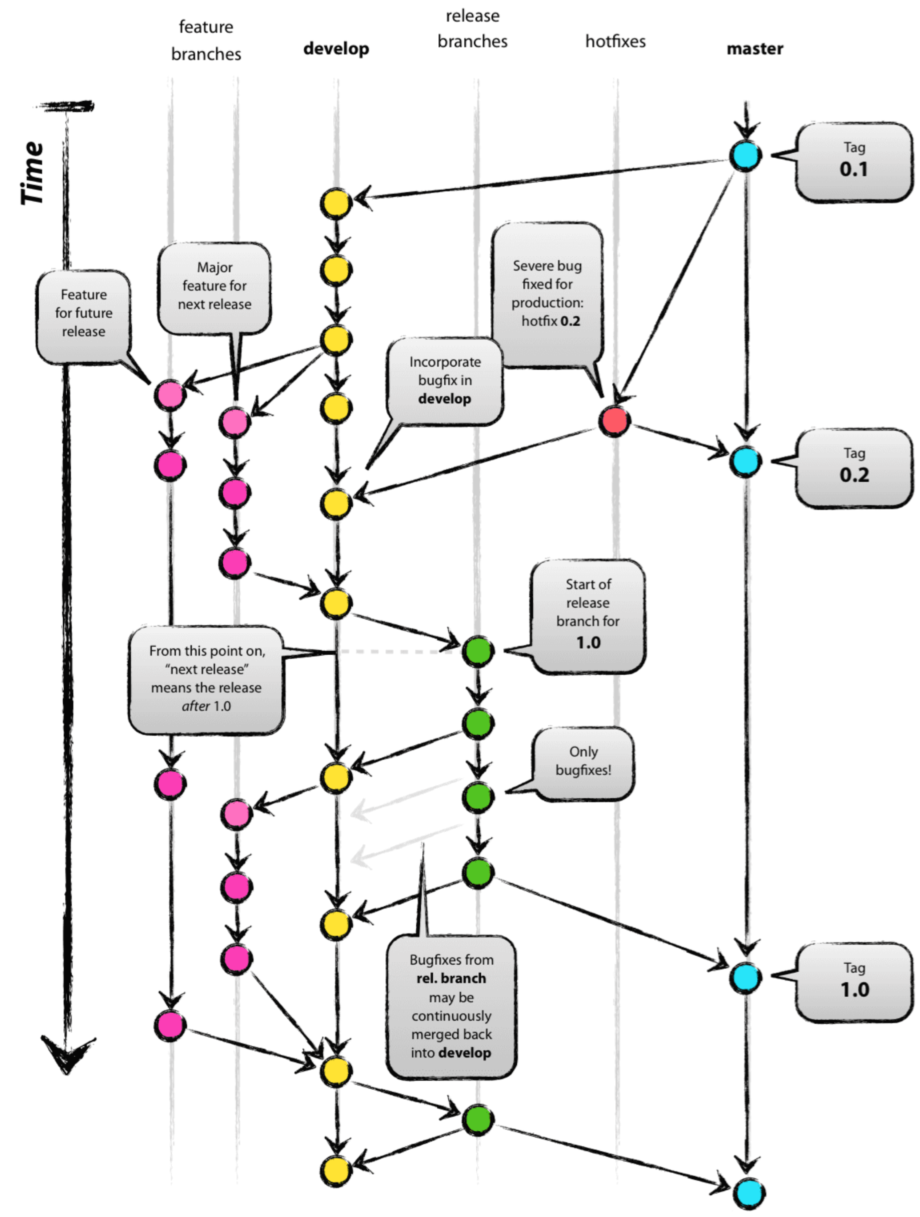
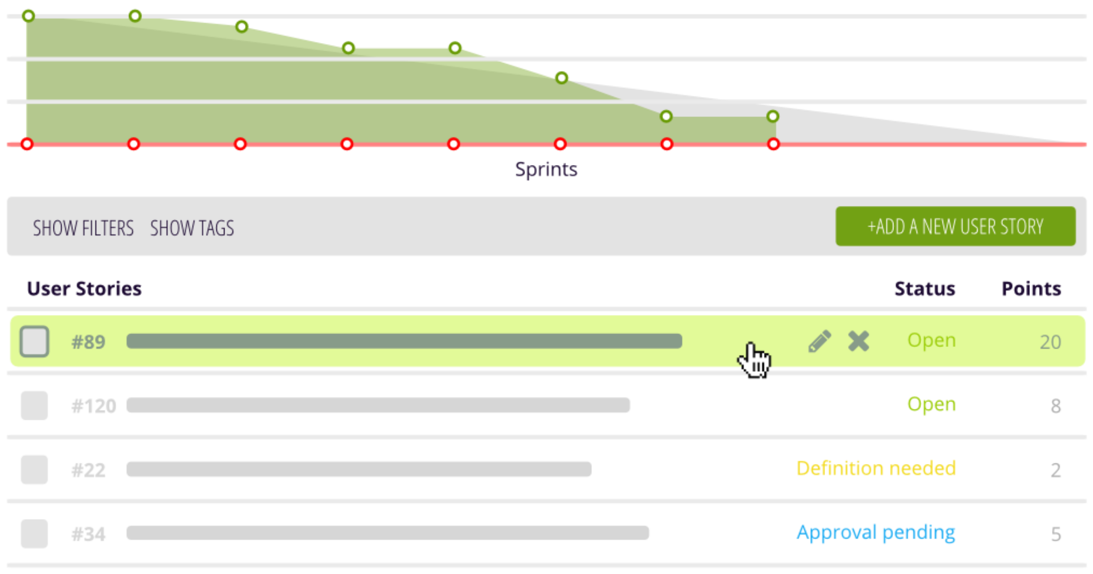

feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Figure 3: Git Workflow

5.2.2 Backlog & Administration



User Stories	Status	Points
#89	Open	20
#120	Open	8
#22	Definition needed	2
#34	Approval pending	5

Figure 4: Taiga.io - Agile Project Management

To manage sprints, keep track of user stories and features, and get an overview of how the project is moving along for the rest of the group, we'll be using a relatively new Project Management System called *Taiga.io*. Taiga is built by developers who simply got tired of bloated, overcomplicated project management systems that tried to meet every possible need and was chosen instead of the previously used *TeamWork.com* because of its slimmed down, minimalistic interface, and functional simplicity in regards to our requirements.

Taiga is elegant, easy to navigate, and has ***just*** the features we need - nothing more, nothing less. It's built for SCRUM or KANBAN, and because the SCRUM version of Taiga contains a KANBAN-like backlog, it meets our requirements perfectly.

5.3 Risk Analysis

There are a lot of factors that can go wrong during a four month long project.
By analysing the threats and risks, we are able to mitigate potential problems that may arrise during the project period.

Weighted Risk Analysis Bachelor Thesis				
Threat	**Probability**	**Impact**	**Time Consumption**	**Total**
Weighted Score:	**30**	**40**	**30**	**100**
Project Too Difficult	0.2	1	0.9	73
	We have underestimated the skills needed to implement this application and we have to work on a different project.			
Group Disbanding	0.1	1	0.8	67
	This is a group project, and because of this there will always be a possibility for one or more of the group members not being able to continue contributing due to various reasons.			
Loss Of Data	0.4	0.7	0.8	64
	Loss of data or information could occur due to various reasons like user faults or hardware trouble.			
Unable To Access Material	0.3	0.2	0.8	41
	All materials and project files are stored digitally and online. If the group are unable to access this information, a lot of time will be wasted.			
Loss Of Communication	0.2	0.1	0.8	34
	If the group members are unable to contact each other, the supervisor or the employer, a lot of time will be wasted.			

6 Implementation Plan

6.1 Gantt Chart

See Appendix B - Gantt Chart

6.2 Comments On Gantt Chart

In our Gantt Chart we have included three main parts: Scrum+Kanban, Project Report and Presentation.
Scrum+Kanban shows the different sprints we have to go through in the development life cycle. Each sprint is 7 days, and during those 7 days we have to implement the tasks chosen for that specific sprint. If we for some reason don't finish a specific task on time, this task carries over to the next sprint.
Sprint #10 however is optional. The reason we included an optional sprint is beacuse of potential bugs and problems that may arrise during the end of the development fase. If the application is finished, tested and working on sprint #9, we skip sprint #10 and continue writing on the project report, if not, we will use this sprint to finish the application.

The Project Report part of the Gantt Chart shows how much time we have dedicated to actually writing on the report. We will start working on the main report as soon as the preliminary project is delivered, and continue working on this report throughout this project.

The last task of the Bachelor Thesis is to work on the presentation. We have dedicated 12 days to the presentation part. This includes making the presentation, the poster, as well as practicing the presentation and trying out potential demoes we will show to the participants.

6.3 Ideas

As of now, we have established more or less how the network communication should work in practice, but the inner workings of the program is not yet decided, as we have two ideas - both of which would work. Some things are common for both ideas and thus these common practices will be explained first, followed by the differences in design.

6.3.1 Application Flow

When the application is started, it's started with an argument that specifies whether it's a slave-node or a master-node. Slave nodes have the option to specify either an IP for the primary Master to connect to, or be run in automatic detection mode.

When a slave is run in automatic mode, it will broadcast a HELLO-packet triggering the master to respond and exchange public keys with the node. When a Master has been found, it's IP and hostname will be displayed to the user, which can confirm that these are correct.

When a slave has specified the IP to connect to, it establishes a connection with the master, and exchanges public keys. After these have been exchanged, communication can continue encrypted.

The slave node sends a hash of all its current rules to the Master, and the date these were last updated (or 0 if never). The master compares this hash against the hash it has for this nodes rules in its database, and proceeds to transfer the rules to be applied by the slave. If the hash from the node didn't match what the master had on record, but the timestamp was 0 for the node, the master can request to import rules from the Node and assign it (to **only**) that node. This would prevent wiping out the Nodes rules, and would allow the Master to synchronize with previously unlinked ID(P)S nodes.

Nodes are allowed to query the master for updated rules by providing the date it was last updated, the hash of its current rules and its unique identifier (generated using hardware parameters); the Master updates the field for the nodes last update and sets it to the current time. If the master hasn't received a query for new rules in more than twice the default interval for the nodes, it will proceed to query the node and initiate an update if the rule-hash has changed.

Apart from this, the Master also runs a RESTful API, which allows an authenticated user to do everything that can be done via the CLI, but in a more sophisticated manner:

- Manage nodes

  - Assign nickname
  - Assign rules to multiple nodes
  - Blacklist nodes from communicating with the master
  - Edit node information (but NOT unique identifier)
  - Query the slave for logs
  - Enable or disable a specific node
  - Get status for a specific node (Online/Not online?)
  - Force update

and **Manage Rules:**

- Add rule
- Add rule feeds (for automatic rule updates from different vendors)
- View rules
  - by software (+ version)
  - by node
  - by subnet (IP-range)
  - by direction (outgoing/incoming)
- Edit rule
- Remove rules

This RESTful API would be used only locally to allow a web-application running on the local machine to communicate with the daemon. This gives some flexibility as it makes the application rather agnostic as to what language or platform the interface is written on, and as long as the requests are only within the local machine and authenticated, it would be possible to support different interfaces for managing the application graphically.

Now, when it comes to the difference in ideas, these are mainly related to the program structure, the differences of which are described below.

6.3.2 Idea #1: Program States

The idea here is that the program is written as one, single program, which can be run in different modes. This means that all nodes would essentially have the same database layout (but nodes wouldn't use as much of it, and would only keep a single node in its database, linked to itself), and the RESTful API would be disabled by default.

This would be based mainly on two existing Python libraries: *Twisted* and *SQLAlchemy*. Twisted would be the basis, which manages network communications, and daemonization; and both the node and the master would build on this for communication. Twisted has a vast library of protocols it can utilize, as well as good logging functionality, and provides good performance and scalability because it is built on something called *deferreds*, which lays the foundation for its asynchronous architecture.

There are also other alternatives here, when large scalability comes into play: Cyclone. Cyclone implements another library, called *Tornado* as a Twisted plugin, making Tornado usable *through* Twisted as a protocol, and Tornado uses non-blocking network I/O to scale to tens of thousands of open connections. However, although we may want to be able to scale to a lot of nodes, it's fairly unlikely that the amount would be in the thousands.

Twisted is also actively maintained, which while it leaves the responsibility up to the Twisted developers, the networking part becomes something that doesn't have to be actively developed on our part now or in the future, as we can pass this off to Twisted.

A downside of this approach would be packaging more than is necessary for slaves that will never run in Master-mode, containing a few hundred kilobytes of extra code that won't be run, and the Master containing some extra classes (such as those used to import/export rules on slaves).

6.3.3 Idea #2: Separate Programs

This idea takes a different approach, in that the program is not written as one large program running in different states and thus behaving slightly differently, but instead the program is written as two separate daemons, a master and a slave.

The master node would only run the master daemon, which contains all of the Master-functionality previously described. The slave would thus only run the slave daemon, which would contain the slave functionality. While the master would be built upon Twisted and SQLAlchemy, the *slaves* *would not*. These would instead manage daemonizing and networking functionality manually and be written from scratch as this is fairly easily done in Python, and the database for each node would be a stripped down version containing only what's absolutely necessary for the node. A slave node could thus never be changed to run as a master and vice versa, their roles are defined by the program they run.

A downside with this is that additional networking functionality that *may* be implemented in the future would require developing it from scratch on the slave node (this could arguably classify as *YAGNI*) or changing it to build on Twisted; and the package would be split in two.

6.3.4 Code Structure

Common for both ideas is the structure of the Master, which would consist of a few basic classes inheriting SQLAlchemy's SQLObject, which makes each object associated to a row in its SQL table, and containing relationships to its related objects - a Node containing a set of Rule objects, for example.

A module loader loads all enabled modules from the module-directory, and a set of functions for various events (such as on_rule_save, on_node_connect, and so on) which allows modules to hook into these events and execute desired functionality when these events are triggered. This paves the way for a highly modular design where events by default do nothing unless a module reacts to it.

If the program is based on separate programs, these events may be slightly different for a slave-node and a master-node.

6.4 Features

As previously mentioned, apart from the basic classes, all actual functionality and making use of the classes happen in the modules. There should be one module responsible for distributing rules, hooking into i.e *on_rule_save()* and then checking all nodes that should have this rule, and updating these.

Running Node.save(), for example, would save the node and be responsible for making sure data is sent to it, and a callback from this would trigger the *on_node_save()* event hook which could be used to alert the administrator of the result.

In general, actions performed by the user trigger events which the modules hook into to perform different actions, ensuring a highly modular design of as low coupling and high cohesion as possible.

Available events:

- Software:
  - on_software_create Dispatched when a new software + version has been added
  - on_software_change Dispatched when an existing software + version has been changed
  - on_software_destroy Dispatched when a software + version is removed from the database
  - on_software_save Dispatched on save regardless of change or if its new
- Key:
  - on_key_create Dispatched when a new key is added
  - on_key_generate Dispatched when a new key is generated
  - on_key_change Dispatched when a key is changed
  - on_key_destroy Dispatched when a key is destroyed
  - on_key_save Dispatched when a key is saved
- Node:
  - on_node_create Dispatched when a new node is added to the database
  - on_node_change Dispatched when an existing node has been updated with new information
  - on_node_connect Dispatched when a node attempts to connect
  - on_node_disconnect Dispatched when a node disconnects
  - on_node_auth_success Dispatched when a node has successfully authenticated
  - on_node_auth_fail Dispatched when a node fails to authenticate
  - on_node_detect Dispatched when a non-connected node is detected on the network
  - on_node_save Dispatched everytime a node is saved
  - on_node_enable Dispatched when a node becomes enabled
  - on_node_disable Dispatched when a node becomes disabled

  - on_node_rule_change Dispatched when rules are added, or removed from a node
  - on_node_ban Dispatched when a node is banned
  - on_node_unban Dispatched when a node is unbanned

- Rule:
  - on_rule_create Dispatched when a rule is created
  - on_rule_change Dispatched when an existing rule is modified
  - on_rule_destroy Dispatched when a rule is destroyed
  - on_rule_save Dispatched everytime a rule is saved

6.4.1 Under-the-Hood

Some functions that are automatic are the handshake and the communication process between the nodes. Apart from optional but recommended parameters, such as IP, this should happen automatically - and so should the generation of a unique ID based on hard drive serial number, CPU ID and CPU vendor.

Slave nodes should be able to manage importing and exporting rules from the ID(P)S systems running on their host automatically once an update has been completed.

6.4.2 Administrating Nodes

As with all administration, the RESTful API (described later) provides a way to integrate a web interface for doing what can already be done on the command line more efficiently.

6.4.2.1 Command-line administration of nodes begin with specifying the resource, in this case *node*, followed by the ID of the node, and the requested action. If no id or action is specified, a list of nodes will be presented.

Actions for nodes are:

- enable Starts the ID(P)S on a node. Slaves can only enable themselves.
- disable Stops the ID(P)S on a node. Slaves can only disable themselves.
- ban Only available for master. Ignore messages from the node.

- unban Only available for master. Allow messages from the node (default).
- add-feed Adds a rule feed from a vendor to a node
- status Checks the current status of a node and gets **all** sub-command information
  - last-seen Gets only the last-seen info for a node
  - last-update Gets the last-update time for a node
  - ipaddress Gets only the ip-address for a node (only master available for slaves)
  - hostname Gets only the hostname of a node
- set Allows modifying host information (not unique ID)
  - nickname Sets the nickname for a node.
  - public-key Sets the path to the public key on the node
- rule Performs actions on rules for that specific node (see rule actions below)
- log Retrieves the most recent log information from a node.

6.4.2.2 Web interface administration provides a dashboard with an overview of active nodes, and is not available for slaves. On the master running the web-interface, three different views are available to get an oversight: Overviewm, Nodes, and Rules. Each object has its own detail page, but as this section is about nodes, that's what the focus will be on here.

The Node page displays a table of all slaves and their status (enabled, disabled, unable to connect), number of rules active, host information, and what software and version it's running. When clicking a node, a detailed page of information is displayed and allows the administrator to modify the information. By clicking the Rules link, rules specific for that node will be displayed, which brings us to Rule administration.

6.4.3 Administrating Rules

Similar to Nodes, Rules are another type of resource that can be edited both via command-line and web interface.

6.4.3.1 Command-line

administration of rules begin with specifying the rule resource followed by the ID of the rule (if the action is on an existing rule - only add acts on a non-existing rule) and any action to take on that rule.

Actions for rules are:

- add Adds a new rule
- update Modifies a rule
- drop Removes a rule
- show Shows all rules
  - id Shows a specific rule by ID.
  - software Shows rules by software.
  - ip Shows rules by IP-address or IP-range.
  - outgoing Shows rules for outgoing direction
  - incoming Shows rules for incoming direction

6.4.3.2 Web interface

provides a page for Rules where all rules of all nodes can be seen, and where information for each rule is sorted in a table. Here, rules can be mass-assigned (or un-assigned) to nodes, and applied instantly.

6.4.4 RESTful API

The Master runs a RESTful API providing access to the command-line arguments for authenticated users (and requires HMAC or certificate based authentication for each request) via Flask.

A web-interface connects to the RESTful API and is thus able to perform the same actions as is available on the Master's commandline through the web-interface. If the implementation is based on idea #1, whether the program is run as master or slave, this means each slave will also have a (by default) inactive RESTful API, and a web-interface. Should this be activated, then the only node that exists to administrate for the slave is itself, and thus it would work only for configuring that host.

A Group Contract

Gruppekontrakt

B.Sc-oppgave

Gruppemedlemmer

- Tommy B. Ingdal
tommy.ingdal@hig.no
(+47) 92016861

- Halvor Thoresen
halvor.thoresen@hig.no
(+47) 99593818

- Victor Rudolfsson
victor.rudulfsson@hig.no
(+47) 47141618

Arbeidsperiode

01. januar 2015 – 15. mai 2015

Arbeidsfordeling

Arbeid skal fordeles jevnt og rettferdig mellom gruppemedlemmene. Hvert gruppemedlem plikter å fortelle gruppen om ev. forhold som gjør at man ikke kan jobbe så mye som antatt. Om en slik situasjon skulle oppstå må arbeid fordeles på nytt mellom medlemmene slik at man ikke blir hengende etter skjema.

Møtetider

Siden flere av gruppemedlemmene har jobb, så er det vanskelig å forholde seg 100% nøyaktig til hvert tidspunkt. Vi har derfor bestemt at tidene under skal overholdes så fremt det lar seg gjøre, men man kan møtes andre dager/tidspunkt om nødvendig.

Mandag:	10.00 – 15.00
Tirsdag:	10.00 – 15.00
Onsdag:	10.00 – 15.00

Uenigheter

Om det skulle oppstå interne uenigheter skal veileder kontaktes. Om man likevel ikke skulle komme til enighet vil veileder ta en avgjørelse på vegne av gruppen.

Avskjedighet

Om et gruppemedlem ikke møter til fastsatt tid, uten gyldig grunn, eller bryter andre interne regler vil det bli gitt en advarsel.

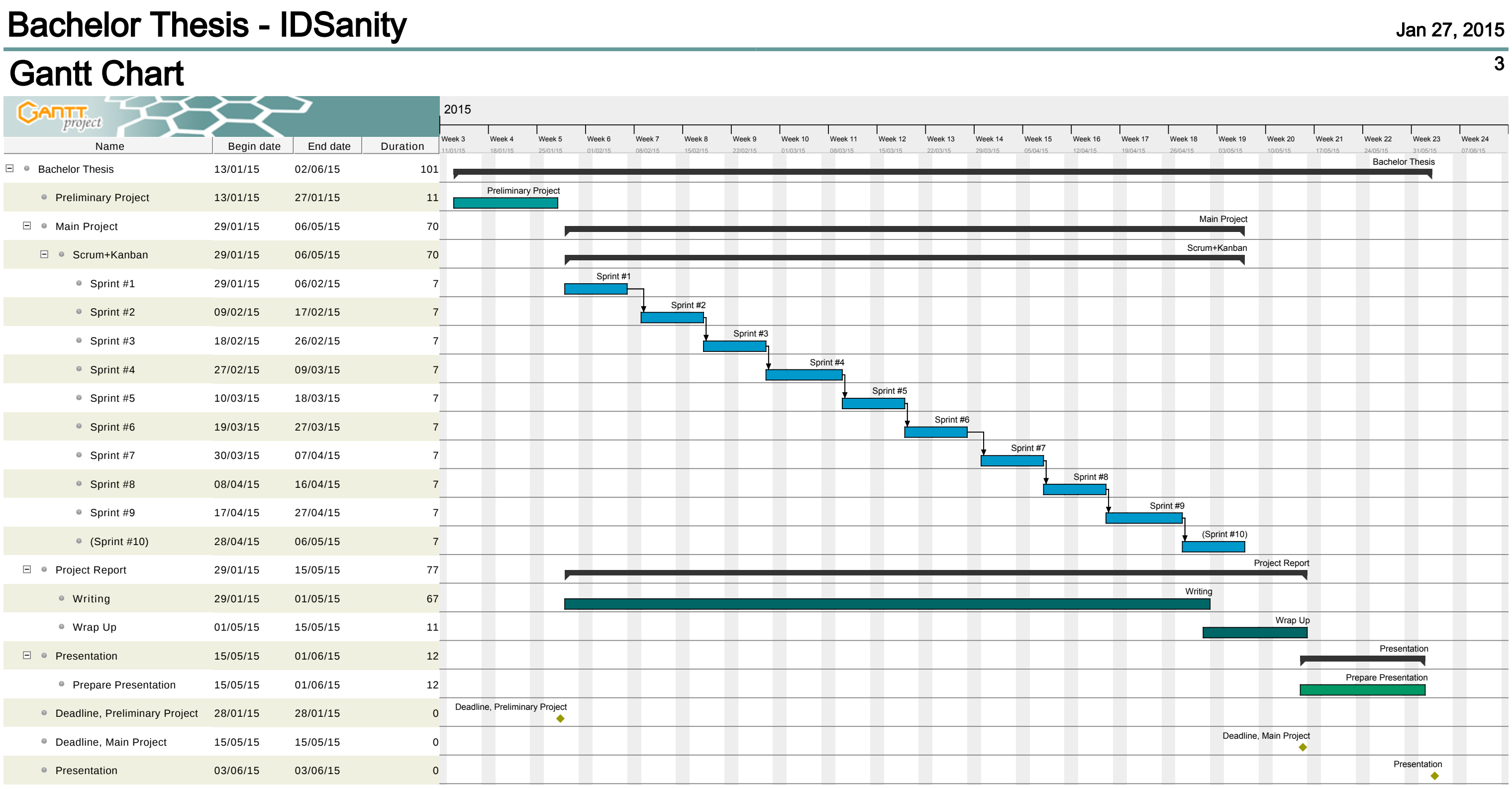
Ved gjentatte regelbrudd vil gruppemedlemmet risikere å bli avskjediget fra gruppen. Veileder vil da bli kontakt for videre håndtering av situasjonen.

Underskrifter

Ved å signere sier jeg meg enig i kontraktens innhold og er klar over at gjentatte regelbrudd kan føre til eksklusjon fra gruppen.

_________________________	_________________________	_________________________
Tommy B. Ingdal	Halvor Thoresen	Victor Rudulfsson

B Gantt Chart

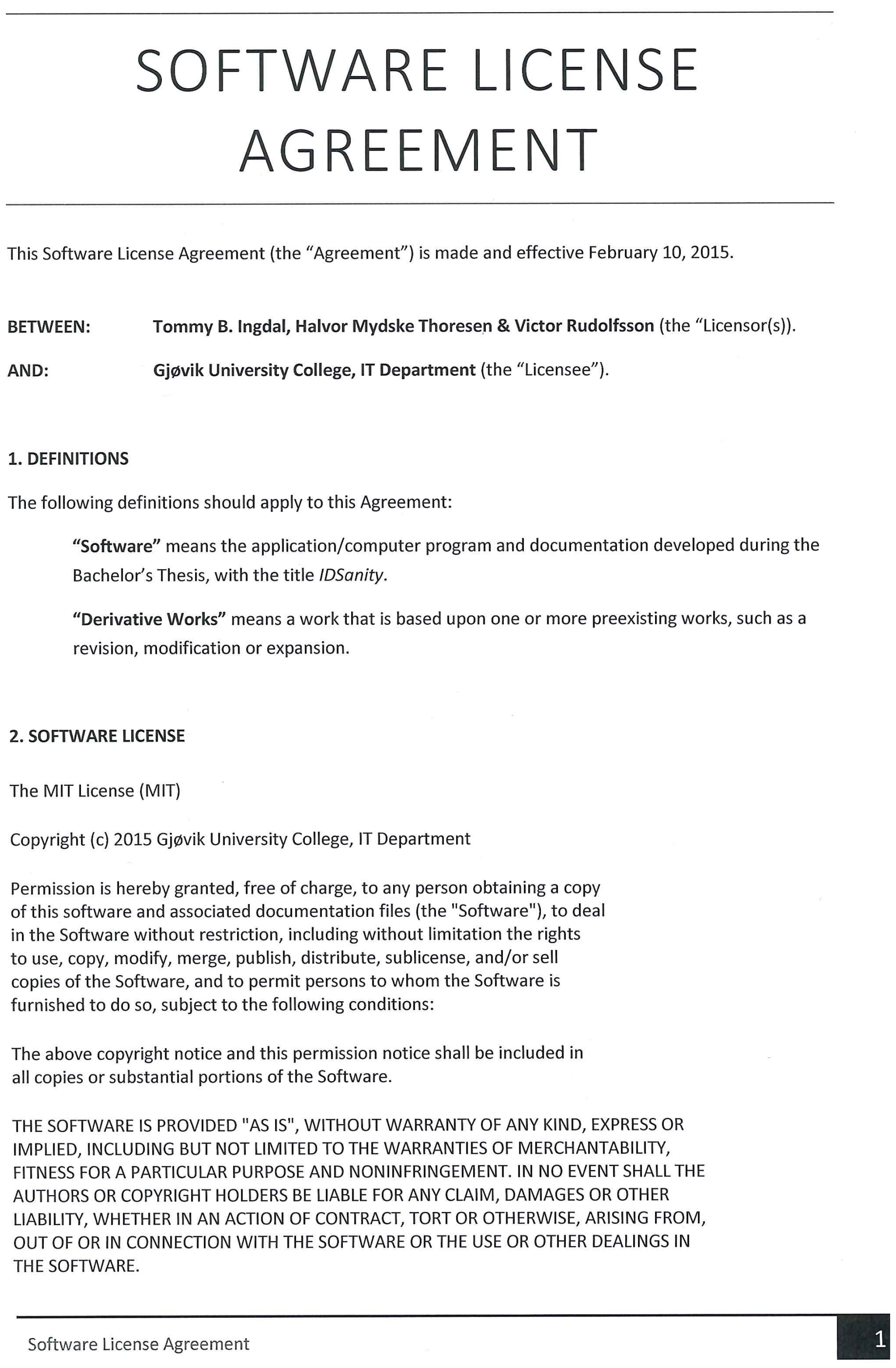


Bachelor Thesis - IDSanity Jan 27, 2015

Gantt Chart 3

Name	Begin date	End date	Duration
Bachelor Thesis	13/01/15	02/06/15	101
Preliminary Project	13/01/15	27/01/15	11
Main Project	29/01/15	06/05/15	70
Scrum+Kanban	29/01/15	06/05/15	70
Sprint #1	29/01/15	06/02/15	7
Sprint #2	09/02/15	17/02/15	7
Sprint #3	18/02/15	26/02/15	7
Sprint #4	27/02/15	09/03/15	7
Sprint #5	10/03/15	18/03/15	7
Sprint #6	19/03/15	27/03/15	7
Sprint #7	30/03/15	07/04/15	7
Sprint #8	08/04/15	16/04/15	7
Sprint #9	17/04/15	27/04/15	7
(Sprint #10)	28/04/15	06/05/15	7
Project Report	29/01/15	15/05/15	77
Writing	29/01/15	01/05/15	67
Wrap Up	01/05/15	15/05/15	11
Presentation	15/05/15	01/06/15	12
Prepare Presentation	15/05/15	01/06/15	12
Deadline, Preliminary Project	28/01/15	28/01/15	0
Deadline, Main Project	15/05/15	15/05/15	0
Presentation	03/06/15	03/06/15	0

C Software License Agreement



SOFTWARE LICENSE AGREEMENT

This Software License Agreement (the "Agreement") is made and effective February 10, 2015.

BETWEEN:	**Tommy B. Ingdal, Halvor Mydske Thoresen & Victor Rudolfsson** (the "Licensor(s)).
AND:	**Gjøvik University College, IT Department** (the "Licensee").

1. DEFINITIONS

The following definitions should apply to this Agreement:

"Software" means the application/computer program and documentation developed during the Bachelor's Thesis, with the title *IDSanity*.

"Derivative Works" means a work that is based upon one or more preexisting works, such as a revision, modification or expansion.

2. SOFTWARE LICENSE

3. MODIFICATIONS AND ENHANCEMENTS

Licensee may enhance and make modifications to the software without Licensor(s) express written consent.

4. THE SOFTWARE

The Software shall consist of the modules or components, shall perform the functions and shall comply with the proposals and specifications, identified or set forth by the Requirement Specification.

5. DOCUMENTATION

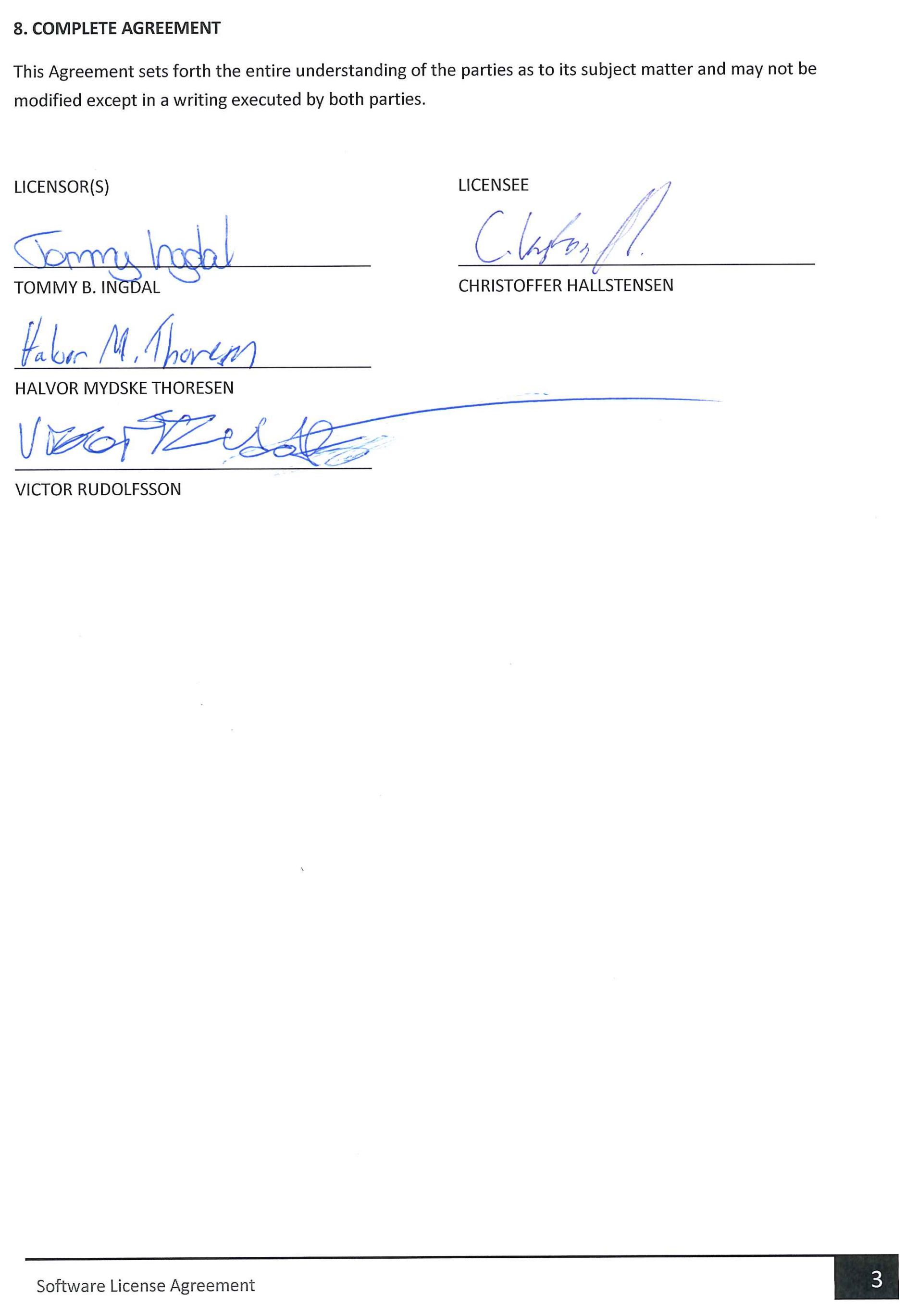
The Documentation shall consist of all user manuals, source code documentation, specifications, and other materials for use in conjunction with the Software. Licensee shall have the right, as part of the license granted herein, to make as many additional copies of the Documentation for its own use as it may determine.

6. SOURCE CODE

The Software shall include its Source Code, and all relevant explanations and documentation of the Source Code (collectively, "Commentary"). Licensor(s) is required to deliver to Licensee, 1 (one) copy of the complete Source Code and a complete listing of the Source Code and Commentary.

7. OWNERSHIP OF THE WORK

Licensor(s) acknowledges that Licensee is the sole and exclusive owner of the Work and of all associated intellectual property. Licensor(s) further agrees that it will not claim ownership right to the Work or any derivative.



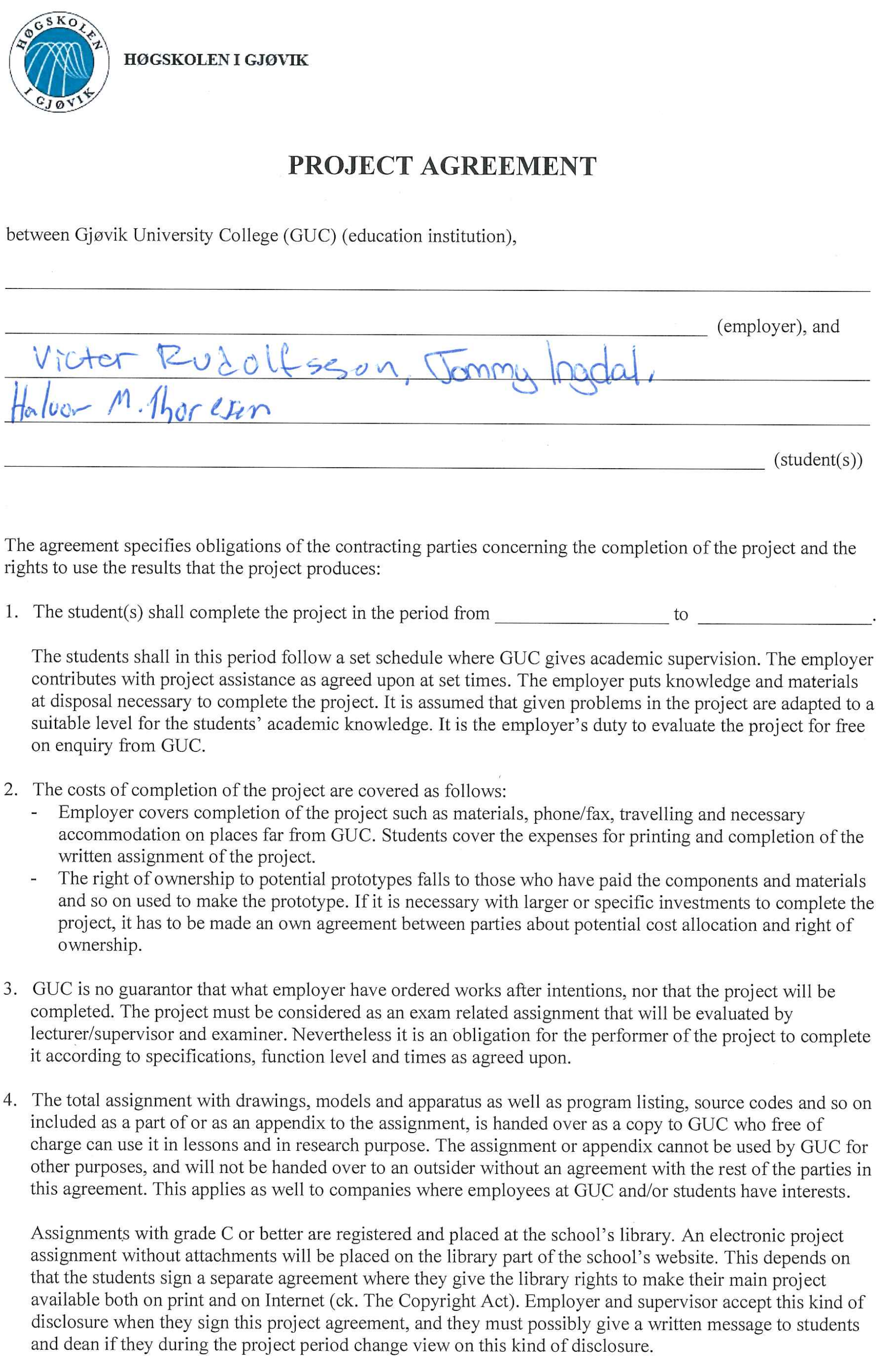
8. COMPLETE AGREEMENT

This Agreement sets forth the entire understanding of the parties as to its subject matter and may not be modified except in a writing executed by both parties.

LICENSOR(S)	LICENSEE
Tommy Ingdal	[illegible]
TOMMY B. INGDAL	CHRISTOFFER HALLSTENSEN
Halvor M. Thoresen	
HALVOR MYDSKE THORESEN	
[illegible]	
VICTOR RUDOLFSSON	

Software License Agreement 3

D Project Agreement



HØGSKOLEN I GJØVIK

PROJECT AGREEMENT

between Gjøvik University College (GUC) (education institution),

____

____________________________________ (employer), and

Victor Rudolfsson, Tommy Ingdal,
Halvor M. Thoresen

_______________________________________ (student(s))

The agreement specifies obligations of the contracting parties concerning the completion of the project and the rights to use the results that the project produces:

1. The student(s) shall complete the project in the period from ______________ to ______________.

   The students shall in this period follow a set schedule where GUC gives academic supervision. The employer contributes with project assistance as agreed upon at set times. The employer puts knowledge and materials at disposal necessary to complete the project. It is assumed that given problems in the project are adapted to a suitable level for the students' academic knowledge. It is the employer's duty to evaluate the project for free on enquiry from GUC.

2. The costs of completion of the project are covered as follows:
   - Employer covers completion of the project such as materials, phone/fax, travelling and necessary accommodation on places far from GUC. Students cover the expenses for printing and completion of the written assignment of the project.
   - The right of ownership to potential prototypes falls to those who have paid the components and materials and so on used to make the prototype. If it is necessary with larger or specific investments to complete the project, it has to be made an own agreement between parties about potential cost allocation and right of ownership.

3. GUC is no guarantor that what employer have ordered works after intentions, nor that the project will be completed. The project must be considered as an exam related assignment that will be evaluated by lecturer/supervisor and examiner. Nevertheless it is an obligation for the performer of the project to complete it according to specifications, function level and times as agreed upon.

4. The total assignment with drawings, models and apparatus as well as program listing, source codes and so on included as a part of or as an appendix to the assignment, is handed over as a copy to GUC who free of charge can use it in lessons and in research purpose. The assignment or appendix cannot be used by GUC for other purposes, and will not be handed over to an outsider without an agreement with the rest of the parties in this agreement. This applies as well to companies where employees at GUC and/or students have interests.

   Assignments with grade C or better are registered and placed at the school's library. An electronic project assignment without attachments will be placed on the library part of the school's website. This depends on that the students sign a separate agreement where they give the library rights to make their main project available both on print and on Internet (ck. The Copyright Act). Employer and supervisor accept this kind of disclosure when they sign this project agreement, and they must possibly give a written message to students and dean if they during the project period change view on this kind of disclosure.

5. The assignment's specifications and results can be used by the employer's own work. If the student(s) in its assignment or while working with it, makes a patentable invention, relations between employer and student(s) applies as described in *Act respecting the right to employees' inventions* of 17th of April 1970, §§ 4-10.

6. Beyond the publicising mentioned in item 4, the student(s) have no right to publicise his/hers/theirs assignment, fully or partly or as a part of another work, without consensus from the employer. Equivalent consent must be made between student(s) and lecturer/supervisor regarding the material placed at disposal by the lecturer/supervisor.

7. The students shall hand in the assignment with attachments electronic (PDF) in Fronter. In addition the students shall hand in a copy to the employer.

8. This agreement is drawn up with one copy to each party. On behalf of GUC it is dean/vice dean who approves the agreement.

9. In each case it is possible to enter separate agreement between employer, student(s) and GUC who closer regulate conditions regarding issues such as ownership, further use, confidentiality, cost coverage, and economic utilisation of the results.

   If employer and student(s) wish an additional or new agreement, this will occur without GUC as a party.

10. When GUC also act as employer, GUC accede to the agreement both as education institution and as employer.

11. Possible disagreements concerning understanding of this agreement are solved by negotiations between the parties. If consensus is not achieved, the parties agree that the disagreement is solved by arbitration, according to provision in Civil Procedure Act of 13th of August 1915, no 6, chapter 32.

12. Participants by project implementation:

GUCs supervisor (name):	stewart kowalski	
Employers contact person (name):	Christoffer Halstensen	
Student(s) (signature):	Tommy Ingdal	date 16/01-15
	[illegible]	date 16/01 -15
	Victor Rudolfsson	date 16/01 -15
		date
Employer (signature):	[illegible]	date 17/1 -15
IMT Dean/Vice Dean (signature):	[illegible]	date 28/01/15

Revised 25th of November, 2010, Hilde Bakke

E Meetings

In this chapter we will go outline the meetings we have had during this project periode. Most of the meetings have happened online, either via Skype or Teamspeak, but we have also had meetings with our supervisors.

16th of January

Where: Gjøvik University College.

Who: Tommy B. Ingdal, Halvor M. Thoresen and Victor Rudolfsson.

What: Discussed different coding styles, which software development model to use and how to organize and manage the project.

All members signed the group contract and the project website was created with Wordpress.

17th of January

Where: Gjøvik University College.

Who: Tommy B. Ingdal, Halvor M. Thoresen, Victor Rudolfsson and the employer.

What: Talked about the planned infrastructure, if we get access to a testing environment, how we should develop the requirement specification and who "owns" the finished product. Employer will give us requirement specification proposal.

19th of January

Where: Gjøvik University College.

Who: Tommy B. Ingdal, Halvor M. Thoresen, Victor Rudolfsson and the supervisors.

What: Tasks related to the preliminary project are delegated between the group members. We also talked about what we should include and got a few tips from the supervisors.

20th of February

Where: Gjøvik University College.

Who: Tommy B. Ingdal, Halvor M. Thoresen, Victor Rudolfsson and the supervisors.

What: Presented the preliminary project for the supervisors and got some quick feedback. The employer gave us an overview of the project, what we need to include etc.

5th of March

Where: Gjøvik University College.

Who: Tommy B. Ingdal, Halvor M. Thoresen and Victor Rudolfsson.

What: Delegated tasks related to the actual writing of the report. We also had a few decisions to make related to the application regarding encryption and database storage.

27th of March

Where: Gjøvik University College.

Who: Tommy B. Ingdal, Halvor M. Thoresen and Victor Rudolfsson.

What: Software License Agreement is now finished and signed by both parties. Main focus is on the application itself.

13th of April

Where: Gjøvik University College.

Who: Tommy B. Ingdal, Halvor M. Thoresen and Victor Rudolfsson.

What: Delegated new tasks in regards of the report. The development of the application is still ongoing, so the work has been diveded; Victor focuses on the application while Tommy and Halvor works on the report.