MozDef Documentation

Mozilla

Jan 03, 2020

Contents

CHAPTER 1

Overview

1.1 What?

It’s easiest to describe The Mozilla Defense Platform (MozDef) as a set of micro-services you can use as an open source Security Information and Event Management (SIEM) overlay on top of Elasticsearch.

1.2 Why?

The inspiration for MozDef comes from the large arsenal of tools available to attackers. Open source suites like metasploit, armitage, lair, dradis and others are readily available to help attackers coordinate, share intelligence and finely tune their attacks in real time.

Open source options for defenders are usually limited to wikis, ticketing systems and manual tracking databases attached to the end of a commercial SIEM.

The Mozilla Defense Platform (MozDef) seeks to automate the security incident handling process and facilitate the real-time activities of incident handlers.

1.3 Goals

1.3.1 High level

- Provide a platform for use by defenders to rapidly discover and respond to security incidents
- Automate interfaces to other systems like firewalls, cloud protections and anything that has an API
- Provide metrics for security events and incidents
- Facilitate real-time collaboration amongst incident handlers
- Facilitate repeatable, predictable processes for incident handling

- Go beyond traditional SIEM systems in automating incident handling, information sharing, workflow, metrics and response automation

1.3.2 Technical

- Offer micro services that make up an Open Source Security Information and Event Management (SIEM)
- Scalable, should be able to handle thousands of events per second, provide fast searching, alerting, correlation and handle interactions between teams of incident handlers

MozDef aims to provide traditional SIEM functionality including:

- Accepting events/logs from a variety of systems.
- Storing events/logs.
- Facilitating searches.
- Facilitating alerting.
- Facilitating log management (archiving,restoration).

It is non-traditional in that it:

- Accepts only JSON input.
- Provides you open access to your data.
- Integrates with a variety of log shippers including logstash, beaver, nxlog, syslog-ng and any shipper that can send JSON to either rabbit-mq or an HTTP(s) endpoint.
- Provides easy integration to Cloud-based data sources such as CloudTrail or GuardDuty.
- Provides easy python plugins to manipulate your data in transit.
- Provides extensive plug-in opportunities to customize your event enrichment stream, your alert workflow, etc.
- Provides realtime access to teams of incident responders to allow each other to see their work simultaneously.

1.4 Architecture

MozDef is based on open source technologies including:

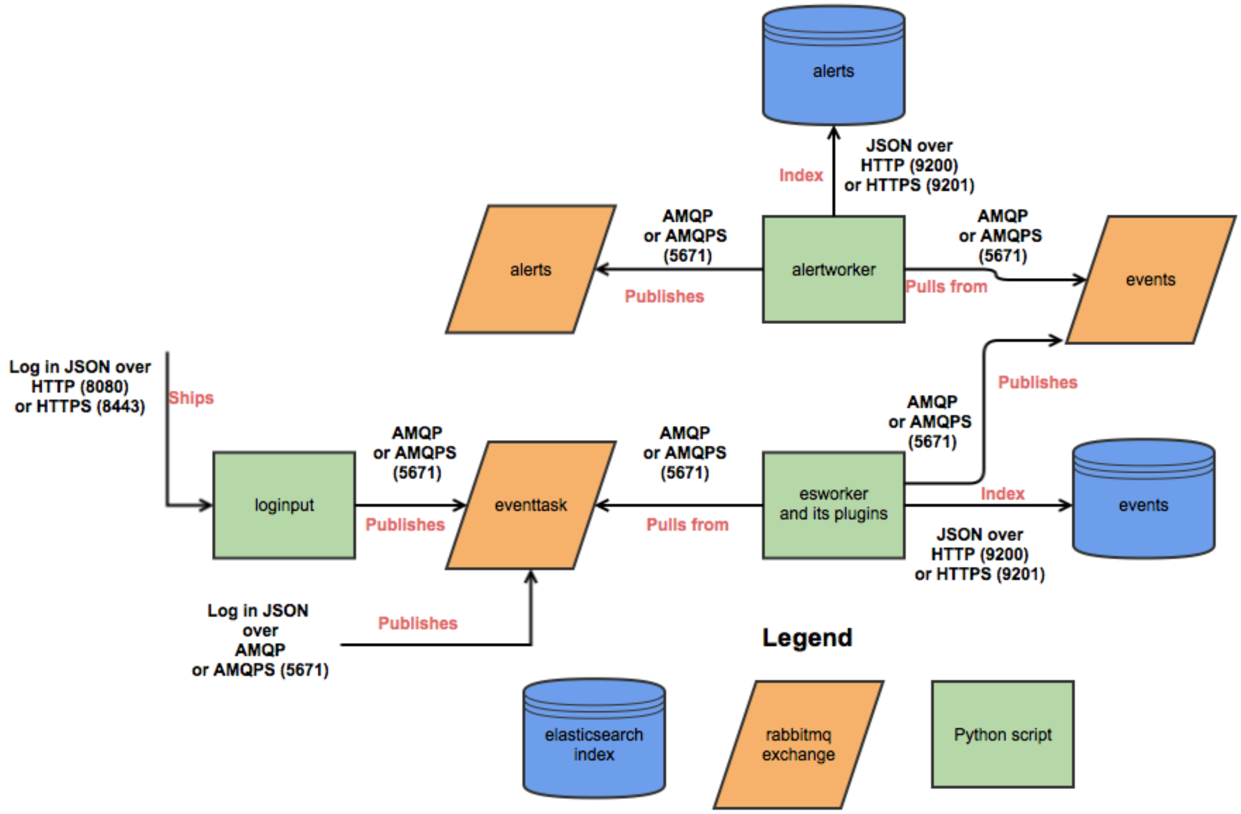
- Nginx (http(s)-based log input)
- RabbitMQ (message queue and amqp(s)-based log input)
- uWSGI (supervisory control of python-based workers)
- bottle.py (simple python interface for web request handling)
- Elasticsearch (scalable indexing and searching of JSON documents)
- Meteor (responsive framework for Node.js enabling real-time data sharing)
- MongoDB (scalable data store, tightly integrated to Meteor)
- VERIS from verizon (open source taxonomy of security incident categorizations)
- d3 (javascript library for data driven documents)
- dc.js (javascript wrapper for d3 providing common charts, graphs)
- three.js (javascript library for 3d visualizations)

- Firefox (a snappy little web browser)

1.4.1 Frontend processing

Frontend processing for MozDef consists of receiving an event/log (in json) over HTTP(S), AMQP(S), or SQS doing data transformation including normalization, adding metadata, etc. and pushing the data to Elasticsearch.

Internally MozDef uses RabbitMQ to queue events that are still to be processed. The diagram below shows the interactions between the python scripts (controlled by uWSGI), the RabbitMQ exchanges and Elasticsearch indices.



1.5 Status

MozDef has been in production at Mozilla since 2014 where we are using it to process over 300 million events per day.

CHAPTER 2

Introduction

2.1 Concept of operations

2.1.1 Event Management

From an event management point of view MozDef relies on Elastic Search for:

- event storage
- event archiving
- event indexing
- event searching

This means if you use MozDef for your log management you can use the features of Elastic Search to store millions of events, archive them to Amazon if needed, index the fields of your events, and search them using highly capable interfaces like Kibana.

MozDef differs from other log management solutions that use Elastic Search in that it does not allow your log shippers direct contact with Elastic Search itself. In order to provide advanced functionality like event correlation, aggregation and machine learning, MozDef inserts itself as a shim between your log shippers (rsyslog, syslog-ng, beaver, nxlog, heka, logstash) and Elastic Search. This means your log shippers interact with MozDef directly and MozDef handles translating their events as they make they're way to Elastic Search.

2.1.2 Event Pipeline

The logical flow of events is:

```
                   +-----------+              +--------------+
                   |    MozDef +--------------+              |
+----------+       |    FrontEnd              | Elastic      |
| shipper  +-------+-----------+              | Search       |
++++++++++++                                  | cluster      |
```

(continues on next page)

(continued from previous page)

```
++++++++++++                                  |              |
| shipper  +-------+-----------+              |              |
+----------+       |    MozDef +--------------+              |
                   |    FrontEnd              |              |
                   +-----------+              |              |
                                              +--------------+
```

Choose a shipper (logstash, nxlog, beaver, heka, rsyslog, etc) that can send JSON over http(s). MozDef uses nginx to provide http(s) endpoints that accept JSON posted over http. Each front end contains a Rabbit-MQ message queue server that accepts the event and sends it for further processing.

You can have as many front ends, shippers and cluster members as you with in any geographic organization that makes sense for your topology. Each front end runs a series of python workers hosted by uwsgi that perform:

- event normalization (i.e. translating between shippers to a common taxonomy of event data types and fields)
- event enrichment
- simple regex-based alerting
- machine learning on the real-time event stream

2.1.3 Event Enrichment

To facilitate event correlation, MozDef allows you to write plugins to populate your event data with consistent metadata customized for your environment. Through simple python plug-ins this allows you to accomplish a variety of event-related tasks like:

- further parse your events into more details
- geoIP tag your events
- correct fields not properly handled by log shippers
- tag all events involving key staff
- tag all events involving previous attackers or hits on a watchlist
- tap into your event stream for ancilary systems
- maintain ‘last-seen’ lists for assets, employees, attackers

2.1.4 Event Correlation/Alerting

Correlation/Alerting is currently handled as a series of queries run periodically against the Elastic Search engine. This allows MozDef to make full use of the lucene query engine to group events together into summary alerts and to correlate across any data source accessible to python.

2.1.5 Incident Handling

From an incident handling point of view MozDef offers the realtime responsiveness of Meteor in a web interface. This allows teams of incident responders the ability to see each others actions in realtime, no matter their physical location.

CHAPTER 3

Installation

MozDef can be run in either Docker containers, or manually on a CentOS 7 machine.

3.1 Docker

Note: When using *docker* on Mac OS X, you may need to tweak Docker to use the aufs filesystem driver (to avoid issues unpacking tar files on overlayfs) Changing Filesystem Driver

Note: MozDef consists of ~10 containers, so it's encouraged to have at least 4GB of memory provided to the Docker daemon.

If you have MozDef source code downloaded locally, you can build the docker containers locally:

```
make build
```

If you want to use pre-built images that are on docker-hub:

```
make build BUILD_MODE=pull
```

Start MozDef:

```
make run
```

You're done! Now go to:

- http://localhost < meteor (main web interface)
- http://localhost:9090/app/kibana < kibana
- http://localhost:8080 < loginput

- http://localhost:514 < syslog input

If you want to stop MozDef:

```
make stop
```

To cleanup all of the existing docker resources used by MozDef:

```
make clean
```

3.2 Manual

These steps outline the process to manually install MozDef in pieces, which can be useful in running MozDef in a distrubuted data center environment.

This installation process has been tested on CentOS 7.

3.2.1 Initial Setup

System Setup

Create the user:

```
adduser mozdef -d /opt/mozdef
mkdir /opt/mozdef/envs
chown -R mozdef:mozdef /opt/mozdef
```

Clone repository:

```
yum install -y git
su mozdef
cd ~/
git clone https://github.com/mozilla/MozDef.git /opt/mozdef/envs/mozdef
```

Python Setup

Setting up a Python 3.6 virtual environment (as root):

```
yum install -y epel-release
yum install -y python36 python36-devel python3-pip libcurl-devel gcc
pip3 install virtualenv
su mozdef
cd /opt/mozdef/envs
/usr/local/bin/virtualenv -p /bin/python3 /opt/mozdef/envs/python
```

Install MozDef python requirements (as mozdef user):

```
source /opt/mozdef/envs/python/bin/activate
cd /opt/mozdef/envs/mozdef
PYCURL_SSL_LIBRARY=nss pip install -r requirements.txt
mkdir /opt/mozdef/envs/mozdef/data
```

Syslog Setup

Copy over mozdef syslog file (as root):

```
cp /opt/mozdef/envs/mozdef/config/50-mozdef-filter.conf /etc/rsyslog.d/50-mozdef-
↪filter.conf
```

Ensure log directory is created:

```
mkdir -p /var/log/mozdef/supervisord
chown -R mozdef:mozdef /var/log/mozdef
```

Restart rsyslog:

```
systemctl restart rsyslog
```

3.2.2 External Services

MozDef uses multiple external open source services to store data. These services can be setup on multiple hosts, allowing for a more distrubuted environment.

Elasticsearch

Elasticsearch is the main data storage of MozDef. It's used to store alerts and event documents, which can then be searched through in a fast and efficient manner. Each day's events is stored in a separate index (events-20190124 for example),

Note: MozDef currently only supports Elasticsearch version 6.8

Elasticsearch requires java, so let's install it:

```
yum install -y java-1.8.0-openjdk
```

Import public signing key of yum repo:

```
rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch
```

Create yum repo file:

```
vim /etc/yum.repos.d/elasticsearch.repo
```

With the following contents:

```
[elasticsearch-6.x]
name=Elasticsearch repository for 6.x packages
baseurl=https://artifacts.elastic.co/packages/6.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
enabled=1
autorefresh=1
type=rpm-md
```

Install elasticsearch:

```
yum install -y elasticsearch
```

Start Service:

```
systemctl start elasticsearch
systemctl enable elasticsearch
```

It may take a few seconds for Elasticsearch to start up, but we can look at the log file to verify when it's ready:

```
tail -f /var/log/elasticsearch/elasticsearch.log
```

Once the services seems to have finished starting up, we can verify using curl:

```
curl http://localhost:9200
```

You should see some information in JSON about the Elasticsearch endpoint (version, build date, etc). This means Elasticsearch is all setup, and ready to go!

Kibana

Kibana is a webapp to visualize and search your Elasticsearch cluster data.

Import public signing key of yum repo:

```
rpm --import https://artifacts.elastic.co/GPG-KEY-elasticsearch
```

Create yum repo file:

```
vim /etc/yum.repos.d/kibana.repo
```

With the following contents:

```
[kibana-6.x]
name=Kibana repository for 6.x packages
baseurl=https://artifacts.elastic.co/packages/6.x/yum
gpgcheck=1
gpgkey=https://artifacts.elastic.co/GPG-KEY-elasticsearch
enabled=1
autorefresh=1
type=rpm-md
```

Install kibana:

```
yum install -y kibana
```

Kibana should work just fine out of the box, but we should take a look at what settings are available:

```
cat /etc/kibana/kibana.yml
```

Some of the settings you'll want to configure are:

- server.name (your server's hostname)
- elasticsearch.url (the url to your elasticsearch instance and port)
- logging.dest (/path/to/kibana.log so you can easily troubleshoot any issues)

Then you can start the service:

```
systemctl start kibana
systemctl enable kibana
```

Now that Kibana and Elasticsearch are setup, we can populate the MozDef indices and Kibana settings:

```
su mozdef
source /opt/mozdef/envs/python/bin/activate
cd /opt/mozdef/envs/mozdef/scripts/setup
python initial_setup.py http://localhost:9200 http://localhost:5601
```

RabbitMQ

RabbitMQ is used on workers to have queues of events waiting to be inserted into the Elasticsearch cluster (storage).

RabbitMQ requires EPEL repos so we need to first install that:

```
yum -y install epel-release
```

Download and install Rabbitmq:

```
wget https://www.rabbitmq.com/releases/rabbitmq-server/v3.6.1/rabbitmq-server-3.6.1-1.noarch.rpm
rpm --import https://www.rabbitmq.com/rabbitmq-signing-key-public.asc
yum install -y rabbitmq-server-3.6.1-1.noarch.rpm
```

COPY docker/compose/rabbitmq/files/rabbitmq.config /etc/rabbitmq/ COPY docker/compose/rabbitmq/files/enabled_plugins /etc/rabbitmq/

Create rabbitmq configuration file:

```
vim /etc/rabbitmq/rabbitmq.config
```

With the following contents:

```
[
  {rabbit,
    [
      {tcp_listeners, [5672]},
      {loopback_users, []}
    ]
  },
  {rabbitmq_management,
    [
      {listener,
        [
          {port, 15672},
          {ip, "127.0.0.1"}
        ]
      }
    ]
  }
].
```

Enable management plugin:

```
vim /etc/rabbitmq/enabled_plugins
```

With the following contents:

```
[rabbitmq_management].
```

Start Service:

```
systemctl start rabbitmq-server
systemctl enable rabbitmq-server
```

MongoDB

Mongodb is the backend database used by Meteor (the web UI).

Note: It’s preferred to run this service on the same host that the Web UI will be running on, so you don’t need to expose this service externally.

Create yum repo file:

```
vim /etc/yum.repos.d/mongodb.repo
```

With the following contents:

```
[mongodb-org-3.4]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/3.4/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-3.4.asc
```

Then you can install mongodb:

```
yum install -y mongodb-org
```

Overwrite config file:

```
cp /opt/mozdef/envs/mozdef/config/mongod.conf /etc/mongod.conf
```

Start Service:

```
systemctl start mongod
systemctl enable mongod
```

Nginx

Nginx is used as a proxy to forward requests to the loginput service.

Install nginx:

```
yum install -y nginx
```

Copy mozdef nginx conf:

```
cp /opt/mozdef/envs/mozdef/config/nginx.conf /etc/nginx/nginx.conf
```

Ensure nginx is started and running:

```
systemctl start nginx
systemctl enable nginx
```

3.2.3 MozDef Services

MozDef services can be broken up into 3 different groups (Alert, Ingest, Web). Each group of services should be run on the same machine, with all of the Ingest services able to run on N number of machines, allowing for a more distrubuted environment.

Note: It's recommended in a distributed environment, to have only 1 Alert Service node, 1 Web Service node, and N Ingest Service nodes.

MozDef Service	Service Type	Required Service(s)
Alerts	Alert	Elasticsearch, RabbitMQ, MozdefRestAPI
Alert Actions	Alert	RabbitMQ
Bot	Alert	RabbitMQ
Loginput	Ingest	RabbitMQ, Nginx
MQ Workers	Ingest	Elasticsearch, RabbitMQ
RestAPI	Web	Elasticsearch, Mongodb, Nginx
Meteor	Web	Mongodb, MozdefRestAPI, Nginx
Kibana	Web	Elasticsearch, Nginx

RestAPI

The MozDef RestAPI service is an HTTP API that works alongside the Mozdef Web Service.

Copy over systemd file:

```
cp /opt/mozdef/envs/mozdef/systemdfiles/web/mozdefrestapi.service /usr/lib/systemd/
↪system/mozdefrestapi.service
```

Start loginput service:

```
systemctl start mozdefrestapi
systemctl enable mozdefrestapi
```

Verify service is working:

```
curl http://localhost:8081/status
```

You should see some json returned!

Alerts

The alerts service searches Elasticsearch for specific terms (defined in specific alerts), and will create an Alert document if any of the alerts found events.

Note: The alerts service depends on Elasticsearch, RabbitMQ, AND the MozdefRestAPI.

Let's edit the configuration file:

```
vim /opt/mozdef/envs/mozdef/alerts/lib/config.py
```

The *ALERTS* dictionary is where we define what alerts are running, and with what schedule they run on. The dictionary *key* consists of 2 fields, the alert filename (excluding the .py extension), and the alert classname. The dictionary *value* is a dictionary containing the schedule. An example:

```
ALERTS = {
    'bruteforce_ssh.AlertBruteforceSsh': {'schedule': crontab(minute='*/1')},
    'unauth_ssh.AlertUnauthSSH': {'schedule': crontab(minute='*/1')},
    'get_watchlist.AlertWatchList': {'schedule': crontab(minute='*/1')},
}
```

Copy over systemd file:

```
cp /opt/mozdef/envs/mozdef/systemdfiles/alert/mozdefalerts.service /usr/lib/systemd/
↪system/mozdefalerts.service
```

Start alerts service:

```
systemctl start mozdefalerts
systemctl enable mozdefalerts
```

Look at logs:

```
tail -f /var/log/mozdef/supervisord/alert_errors.log
```

Alert Actions

The Alert Actions service consists of pluggable modules that perform certain actions if certain alerts are detected. These actions are simply python files, so actions like sending an email or triggering a pagerduty notification are possible.

These actions are stored in */opt/mozdef/envs/mozdef/alerts/actions/*

Let's edit the configuration file:

```
vim /opt/mozdef/envs/mozdef/alerts/lib/config.py
```

The *ALERT_ACTIONS* list is where we define what alert actions are running. Each entry is simply the filename of the alert action to run (excluding the .py extension). An example:

```
ALERT_ACTIONS = [
    'pagerDutyTriggerEvent',
]
```

Copy over systemd file:

```
cp /opt/mozdef/envs/mozdef/systemdfiles/alert/mozdefalertactions.service /usr/lib/
↪systemd/system/mozdefalertactions.service
```

Start alert actions service:

```
systemctl start mozdefalertactions
systemctl enable mozdefalertactions
```

Look at logs:

```
tail -f /var/log/mozdef/alertactions.log
```

Bot

The MozDef Bot service is a method for sending alerts to either an IRC or slack channel(s).

The source code for this service is broken up into multiple directories, depending on if you want to use Slack or IRC. For this example, we're going to be using the Slack functionality.

Let's edit the configuration file and set our channel and secrets accordingly:

```
vim /opt/mozdef/envs/mozdef/bot/slack/mozdefbot.conf
```

Copy over systemd file:

```
cp /opt/mozdef/envs/mozdef/systemdfiles/alert/mozdefbot.service /usr/lib/systemd/
↪system/mozdefbot.service
```

Start bot service:

```
systemctl start mozdefbot
systemctl enable mozdefbot
```

Look at logs:

```
tail -f /var/log/mozdef/mozdefbot.log
```

Cron

Crontab is used to run periodic maintenance tasks in MozDef.

MozDef cron entries can be broken up similarly to the 3 Service Groups (Alerts, Ingest, Web).

Note: You should run the Ingest Related Tasks on each ingest host that you have in your MozDef environment.

Recommended Mozdef Crontab:

```
su mozdef
crontab -e
```

With the following contents:

```
## Alert Related Tasks ##
*/15 * * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/
↪eventStats.sh
0 0 * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/esMaint.sh
0 8 * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/pruneES.sh
0 0 * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/update_
↪geolite_db.sh
0 1 * * 0 /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/
↪closeIndices.sh
```

(continues on next page)

(continued from previous page)

```
## Ingest Related Tasks ##
* * * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/
↪healthAndStatus.sh
0 0 * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/update_
↪geolite_db.sh

## Web Related Tasks ##
* * * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/
↪healthToMongo.sh
* * * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/
↪collectAttackers.sh
* * * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/
↪syncAlertsToMongo.sh
# Uncomment if running multiple Elasticsearch nodes
#0 * * * * /opt/mozdef/envs/mozdef/cron/cronic /opt/mozdef/envs/mozdef/cron/
↪esCacheMaint.sh
```

Loginput

The MozDef Loginput service is an HTTP API to send events to MozDef by external sources.

Copy over systemd file:

```
cp /opt/mozdef/envs/mozdef/systemdfiles/consumer/mozdefloginput.service /usr/lib/
↪systemd/system/mozdefloginput.service
```

Start loginput service:

```
systemctl start mozdefloginput
systemctl enable mozdefloginput
```

Verify service is working:

```
curl http://localhost:8080/status
```

You should see some json returned!

MQ Workers

MQ Workers are the main service to pull events into MozDef. These workers can pull from a queue from RabbitMQ, Cloudtrail, Amazon SQS, Amazon SNS and Papertrail.

The MQ worker files are stored in */opt/mozdef/envs/mozdef/mq/*.

For this example, we're going to focus on configuring and running the *eventtask* worker. This specific workers pull from a RabbitMQ queue, which will be populated by the MozDef Loginput service.

Each MQ worker service has the following associated files:

1. A .ini file used to control certain properties of the worker service (number of processes, logging directory, etc). Modify eventtask ini file:

```
vim /opt/mozdef/envs/mozdef/mq/eventtask.ini
```

Note: The mules key is used to determine how many "processes" the worker service will run. Depending on the amount of incoming messages, you may need to duplicate this line (thus adding more processes to run).

2. A .conf file used to store credentials and other configuration options for the worker process. Modify eventtask conf file:

```
vim /opt/mozdef/envs/mozdef/mq/esworker_eventtask.conf
```

3. A systemd file used to start/stop/enable the specific MQ worker. Copy systemd file into place:

```
cp /opt/mozdef/envs/mozdef/systemdfiles/consumer/mworker-eventtask.service /usr/lib/
↪systemd/system/mworker-eventtask.service
```

Start worker:

```
systemctl start mworker-eventtask
systemctl enable mworker-eventtask
```

Web

Meteor is a javascript framework used for the realtime aspect of the web interface.

Install requirements:

```
export NODE_VERSION=8.11.4
export METEOR_VERSION=1.8

cd /opt/mozdef
gpg="gpg --no-default-keyring --secret-keyring /dev/null --keyring /dev/null --no-
↪option --keyid-format 0xlong"
rpmkeys --import /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7
rpm -qi gpg-pubkey-f4a80eb5 | $gpg | grep 0x24C6A8A7F4A80EB5
curl --silent --location https://rpm.nodesource.com/setup_8.x | bash -
rpmkeys --import /etc/pki/rpm-gpg/NODESOURCE-GPG-SIGNING-KEY-EL
rpm -qi gpg-pubkey-34fa74dd | $gpg | grep 0x5DDBE8D434FA74DD
yum install -y \
   make \
   glibc-devel \
   gcc \
   gcc-c++ \
   libstdc++ \
   zlib-devel \
   nodejs
curl --silent --location https://static-meteor.netdna-ssl.com/packages-bootstrap/
↪$METEOR_VERSION/meteor-bootstrap-os.linux.x86_64.tar.gz \
  | tar --extract --gzip --directory /opt/mozdef .meteor
ln --symbolic /opt/mozdef/.meteor/packages/meteor-tool/*/mt-os.linux.x86_64/scripts/
↪admin/launch-meteor /usr/bin/meteor
install --owner mozdef --group mozdef --directory /opt/mozdef/envs /opt/mozdef/envs/
↪mozdef
chown -R mozdef:mozdef /opt/mozdef/envs/mozdef/meteor
chown -R mozdef:mozdef /opt/mozdef/.meteor
```

Let's edit the configuration file:

```
vim /opt/mozdef/envs/mozdef/meteor/imports/settings.js
```

Note: We'll need to modify the rootURL and kibanaURL variables in settings.js

The default setting will use Meteor Accounts, but you can just as easily install an external provider like Github, Google, Facebook or your own OIDC:

```
mozdef = {
  ...
  authenticationType: "meteor-password",
  ...
}
```

or for an OIDC implementation that passes a header to the nginx reverse proxy (for example using OpenResty with Lua and Auth0):

```
mozdef = {
  ...
  authenticationType: "OIDC",
  ...
}
```

In addition, environment variables can also be set instead of requiring modification of the settings.js file.:

```
OPTIONS_METEOR_ROOTURL is "http://localhost" by default and should be set to the dns␣
↪name of the UI where you will run MozDef
OPTIONS_METEOR_PORT is 80 by default and is the port on which the UI will run
OPTIONS_METEOR_ROOTAPI is http://rest:8081 by default and should resolve to the␣
↪location of the rest api
OPTIONS_METEOR_KIBANAURL is http://localhost:9090/app/kibana# by default and should␣
↪resolve to your kibana installation
OPTIONS_METEOR_ENABLECLIENTACCOUNTCREATION is true by default and governs whether␣
↪accounts can be created
OPTIONS_METEOR_AUTHENTICATIONTYPE is meteor-password by default and can be set to␣
↪oidc to allow for oidc authentication
OPTIONS_REMOVE_FEATURES is empty by default, but if you pass a comma separated list␣
↪of features you'd like to remove they will no longer be available.
```

Install mozdef meteor project:

```
su mozdef
export MONGO_URL=mongodb://localhost:3002/meteor
export ROOT_URL=http://localhost
export PORT=3000

mkdir -p /opt/mozdef/envs/meteor/mozdef
cd /opt/mozdef/envs/mozdef/meteor
meteor npm install
meteor build --server localhost:3002 --directory /opt/mozdef/envs/meteor/mozdef
ln --symbolic /opt/mozdef/envs/meteor/mozdef/node_modules /opt/mozdef/envs/mozdef/
↪meteor/node_modules
cd /opt/mozdef/envs/meteor/mozdef/bundle/programs/server
npm install
```

Copy over systemd file (as root):

```
cp /opt/mozdef/envs/mozdef/systemdfiles/web/mozdefweb.service /usr/lib/systemd/system/
 ↪mozdefweb.service
systemctl daemon-reload
```

Start loginput service:

```
systemctl start mozdefweb
systemctl enable mozdefweb
```

CHAPTER 4

Usage

4.1 Web Interface

MozDef uses the Meteor framework for the web interface and bottle.py for the REST API. For authentication, MozDef supports local account creation. Meteor (the underlying UI framework) supports many authentication options including google, github, twitter, facebook, oath, native accounts, etc.

4.1.1 Events visualizations

Since the backend of MozDef is Elastic Search, you get all the goodness of Kibana with little configuration. The MozDef UI is focused on incident handling and adding security-specific visualizations of SIEM data to help you weed through the noise.

4.2 Sending logs to MozDef

Events/Logs are accepted as json over http(s) with the POST or PUT methods or over rabbit-mq. Most modern log shippers support json output. MozDef is tested with support for:

- heka
- beaver
- nxlog
- logstash
- rsyslog
- native python code
- AWS cloudtrail (via native python)

We have some configuration snippets

4.2.1 What should I log?

If your program doesn't log anything it doesn't exist. If it logs everything that happens it becomes like the proverbial boy who cried wolf. There is a fine line between logging too little and too much but here is some guidance on key events that should be logged and in what detail.

Event	Example	Rationale
Authentication Events	Failed/Success logins	Authentication is always an important event to log as it establishes traceability for later events and allows correlation of user actions across systems.
Authorization Events	Failed attempts to insert/update/delete a record or access a section of an application.	Once a user is authenticated they usually obtain certain permissions. Logging when a user's permissions do not allow them to perform a function helps troubleshooting and can also be helpful when investigating security events.
Account Lifecycle	Account creation/deletion/update	Adding, removing or changing accounts are often the first steps an attacker performs when entering a system.
Password/Key Events	Password changed, expired, reset. Key expired, changed, reset.	If your application takes on the responsibility of storing a user's password (instead of using a centralized source) it is important to note changes to a users credentials or crypto keys.
Account Activations	Account lock, unlock, disable, enable	If your application locks out users after failed login attempts or allows for accounts to be inactivated, logging these events can assist in troubleshooting access issues.
Application Exceptions	Invalid input, fatal errors, known bad things	If your application catches errors like invalid input attempts on web forms, failures of key components, etc creating a log record when these events occur can help in troubleshooting and tracking security patterns across applications. Full stack traces should be avoided however as the signal to noise ratio is often overwhelming. It is also preferable to send a single event rather than a multitude of events if it is possible for your application to correlate a significant exception. For example, some systems are notorious for sending a connection event with source IP, then sending an authentication event with a session ID then later sending an event for invalid input that doesn't include source IP or session ID or username. Correctly correlating these events across time is much more difficult than just logging all pieces of information if it is available.

4.3 JSON format

This section describes the structure JSON objects to be sent to MozDef. Using this standard ensures developers, admins, etc are configuring their application or system to be easily integrated into MozDef.

4.3.1 Background

Mozilla used CEF as a logging standard for compatibility with Arcsight and for standardization across systems. While CEF is an admirable standard, MozDef prefers JSON logging for the following reasons:

- Every development language can create a JSON structure.
- JSON is easily parsed by computers/programs which are the primary consumer of logs.
- CEF is primarily used by Arcsight and rarely seen outside that platform and doesn't offer the extensibility of JSON.
- A wide variety of log shippers (heka, logstash, fluentd, nxlog, beaver) are readily available to meet almost any need to transport logs as JSON.
- JSON is already the standard for cloud platforms like amazon's cloudtrail logging.

4.3.2 Description

As there is no common RFC-style standard for json logs, we prefer the following structure adapted from a combination of the graylog GELF and logstash specifications.

Note all fields are lowercase to avoid one program sending sourceIP, another sending sourceIp, another sending SourceIPAddress, etc. Since the backend for MozDef is elasticsearch and fields are case-sensitive this will allow for easy compatibility and reduce potential confusion for those attempting to use the data. MozDef will perform some translation of fields to a common schema but this is intended to allow the use of heka, nxlog, beaver and retain compatible logs.

4.3.3 Mandatory Fields

Field	Purpose	Sample Value
category	General category/type of event matching the 'what should I log' section below	authentication, authorization, account creation, shutdown, atartup, account deletion, account unlock, zeek
details	Additional, event-specific fields that you would like included with the event. Please completely spell out a field rather an abbreviate: i.e. sourceipaddress instead of srcip.	<see below>
hostname	The fully qualified domain name of the host sending the message	server1.example.com
processid	The PID of the process sending the log	1234
processname	The name of the process sending the log	myprogram.py
severity	RFC5424 severity level of the event in all caps: DEBUG, INFO, NOTICE, WARNING, ERROR, CRITICAL, ALERT, EMERGENCY	INFO
source	Source of the event (file name, system name, component name)	/var/log/syslog/2014.01.02.log
summary	Short human-readable version of the event suitable for IRC, SMS, etc.	john login attempts over threshold, account locked
tags	An array or list of any tags you would like applied to the event	vpn, audit nsm,zeek,intel
timestamp	Full date plus time timestamp of the event in ISO format including the timezone offset	2014-01-30T19:24:43+06:00
utctimestamp	Full UTC date plus time timestamp of the event in ISO format including the timezone offset	2014-01-30T13:24:43+00:00
receivedtimestamp	Full UTC date plus time timestamp in ISO format when mozdef parses the event. This is set by mozdef upon receipt of the event	2014-01-30T13:24:43+00:00

4.3.4 Details substructure (mandatory if such data is sent, otherwise optional)

Field	Purpose	Sample Value
destinationipaddress	Destination IP of a network flow	8.8.8.8
destinationport	Destination port of a network flow	80
sourceipaddress	Source IP of a network flow	8.8.8.8
sourceport	Source port of a network flow	42297
sourceuri	Source URI such as a referer	https://www.mozilla.org/
destinationuri	Destination URI as in "wget this URI"	https://www.mozilla.org/
error	Action resulted in an error or failure	true/false
success	Transaction failed/ or succeeded	true/false
username	Username, email, login, etc.	kang@mozilla.com
useragent	Program agent string	curl/1.76 (Windows; 5.1)

4.3.5 Examples

```
{
    "timestamp": "2014-02-14T11:48:19.035762739-05:00",
    "hostname": "somemachine.in.your.company.com",
    "processname": "/path/to/your/program.exe",
    "processid": 3380,
    "severity": "INFO",
    "summary": "joe login failed",
    "category": "authentication",
    "source": "ldap",
    "tags": [
        "ldap",
        "adminAccess",
            "failure"
    ],
    "details": {
        "username": "joe",
        "task": "access to admin page /admin_secret_radioactiv",
        "result": "10 authentication failures in a row",
        "success": false
    }
}
```

4.4 Simple test

If you want to just post some sample json to Mozdef do something like

```
curl -v --header "Content-Type: application/json"   --request POST   --data '{"tags":␣
↪["test"],"summary": "just a test"}'   http://localhost:8080/events
```

where http://localhost:8080 is whatever is running the 'loginput' service. The 'data' curl option is what gets posted as json to MozDef. If your post is successful, you should then be able to find the event in elastic search/kibana.

4.5 Alert Development Guide

This guide is for someone seeking to write a MozDef alert

4.5.1 How to start developing your new alert

Run:

```
make new-alert
```

This will prompt for information and create two things:

- <The new alert file>
- <The new alert test file>

You can now edit these files in place, but it is recommended that you run unit-tests on the new alert to make sure it passes before editing (instructions below).

4.5.2 How to run tests on your alert

Requirements:

- Make sure you have the latest version of docker installed.
- Known Issues: docker's overlayfs has a known issue with tar files, so you will need to go to Docker => Preferences => Daemon => Advanced and add the following key pair ("storage-driver" : "aufs"). You may also need to allow more than 2GB for docker depending on which containers you run.

```
make build-tests
make run-tests TEST_CASE=tests/alerts/[YOUR ALERT TEST FILE].py
```

This test should pass and you will have confirmed you have a working environment.

At this point, begin development and periodically run your unit-tests locally with the following commands:

```
make build-tests
make run-tests TEST_CASE=tests/alerts/[YOUR ALERT TEST FILE].py
```

4.5.3 Background on concepts

- Logs - These are individual log entries that are typically emitted from systems, like an Apache log.
- Events - The entry point into MozDef, a log parsed into JSON by some log shipper (syslog-ng, nxlog) or a native JSON data source like GuardDuty, CloudTrail, most SaaS systems, etc.
- Alerts - These are either a 1:1 events to alerts (this thing happens and alert) or a M:1 events to alerts (N of these things happen and alert).

Alerts in MozDef are mini python programs. Most of the work is done by the alert library so the portions you will need to code fall into two functions:

- main - This is where the alert defines the criteria for the types of events that will trigger the alert.
- onAggregation/onEvent - This is where the alert defines what happens when it sees those events, such as post processing of events and making them into a useful summary to emit as an alert.

In both cases the alert is simple python, and you have all the facility of python at your disposal including any of the python libraries you might want to add to the project.

It's important to note that when you iterate on the alert to regularly test to ensure that the alert is still firing. Should you run into a situation where it's not firing, the best way to approach this is to backout the most recent change and review the alert and tests to ensure that the expectations are still in sync.

4.5.4 Example first alert

Let's step through creating a simple alert you might want to verify a working deployment. For this sub-section it is assumed that you have a working MozDef instance which resides in some MozDefDir and is receiving logs.

First move to to your MozDefDir and issue

```
make new-alert
```

You will be asked for a string to name a new alert and the associated test. For this example we will use the string "foo"

```
make new-alert
Enter your alert name (Example: proxy drop executable): foo
Creating alerts/foo.py
Creating tests/alerts/test_foo.py
```

These will be created as above in the alerts and tests/alerts directories. There’s a lot to the generated code, but a class called “AlertFoo” is of immediate interest and will define when and how to alert. Here’s the head of the auto generated class.

```
class AlertFoo(AlertTask):
  def main(self):
      # Create a query to look back the last 20 minutes
      search_query = SearchQuery(minutes=20)

      # Add search terms to our query
      search_query.add_must([
          TermMatch('category', 'helloworld'),
          ExistsMatch('details.sourceipaddress'),
      ])
      ...
```

This code tells MozDef to query the collection of events for messages timestamped within 20 minutes from time of query execution which are of category “helloworld” and also have a source IP address. If you’re pumping events into MozDef odds are you don’t have any which will be tagged as “helloworld”. You can of course create those events, but lets assume that you have events tagged as “syslog” for the moment. Change the TermMatch line to

```
TermMatch('category', 'syslog'),
```

and you will create alerts for events marked with the category of ‘syslog’. Ideally you should edit your test to match, but it’s not strictly necessary.

4.5.5 Scheduling your alert

Next we will need to enable the alert. Alerts in MozDef are scheduled via the celery task scheduler. The schedule passed to celery is in the config.py file:

Open the file

```
docker/compose/mozdef_alerts/files/config.py
```

or simply

```
alerts/files/config.py
```

if you are not working from the docker images and add your new foo alert to the others with a crontab style schedule

```
ALERTS = {
  'foo.AlertFoo': {'schedule': crontab(minute='*/1')},
  'bruteforce_ssh.AlertBruteforceSsh': {'schedule': crontab(minute='*/1')},
  'unauth_ssh.AlertUnauthSSH': {'schedule': crontab(minute='*/1')},
}
```

The format is _‘pythonfilename.classname’: {‘schedule’: crontab(timeunit=’*/x’)}_ and you can use any celery time unit (minute, hour) along with any schedule that makes sense for your environment. Alerts don’t take many resources to execute, typically finishing in sub second times, so it’s easiest to start by running them every minute.

4.5.6 How to run the alert in the docker containers

Once you've got your alert passing tests, you'd probably like to send in events in a docker environment to further refine, test, etc.

There are two ways to go about integration testing this with docker: 1) Use 'make run' to rebuild the containers each time you iterate on an alert 2) Use docker-compose with overlays to instantiate a docker environment with a live container you can use to iterate your alert

In general, the 'make run' approach is simpler, but can take 5-10mins each iteration to rebuild the containers (even if cached).

To use the 'make run' approach, you edit your alert. Add it to the docker/compose/mozdef_alerts/files/config.py file for scheduling as discussed above and simply:

```
make run
```

This will rebuild any container that needs it, use cache for any that haven't changed and restart mozdef with your alert.

To use a live, iterative environment via docker-compose:

```
docker-compose -f docker/compose/docker-compose.yml -f docker/compose/dev-alerts.yml -
↪p mozdef up
```

This will start up all the containers for a mozdef environment and in addition will allow you an adhoc alerts container to work in that loads the /alerts directory as a volume in the container. To run the alert you are developing you will need to edit the alerts/lib/config.py file as detailed above to schedule your alert. You will also need to edit it to reference the container environment as follows

```
RABBITMQ = {
    'mqserver': 'rabbitmq',
...
ES = {
  'servers': ['http://elasticsearch:9200']
}
```

Once you've reference the containers, you can shell into the alerts container:

```
docker exec -it mozdef_alerts_1 bash
```

Next, start celery

```
celery -A lib.tasks worker --loglevel=info --beat
```

If you need to send in adhoc events you can usually do it via curl as follows:

```
curl -v --header "Content-Type: application/json" --request POST --data '{"tags": [
↪"test"],"category": "helloworld","details":{"sourceipaddress":"1.2.3.4"}}' http://
↪loginput:8080/events
```

4.5.7 How to get the alert in a release of MozDef?

If you'd like your alert included in the release version of Mozdef, the best way is to propose a pull request and ask for a review from a MozDef developer. They will be able to help you get the most out of the alert and help point out pitfalls. Once the alert is accepted into MozDef master, there is a process by which MozDef installations can make use or 'enable' that alert. It's best to work with that MozDef instance's maintainer to enable any new alerts.

4.5.8 Customizing the alert summary

On the alerts page of the MozDef web UI each alert is given a quick summary and for many alerts it is useful to have contextual information displayed here. Looking at the example foo alert we see

```
def onAggregation(self, aggreg):
    # aggreg['count']: number of items in the aggregation, ex: number of failed login␣
↪attempts
    # aggreg['value']: value of the aggregation field, ex: toto@example.com
    # aggreg['events']: list of events in the aggregation
    category = 'My first alert!'
    tags = ['Foo']
    severity = 'NOTICE'
    summary = "Foo alert"

    # Create the alert object based on these properties
    return self.createAlertDict(summary, category, tags, aggreg['events'], severity)
```

This is where the alert object gets created and returned. In the above code the summary will simply be "Foo Alert", but say we want to know how many log entries were collected in the alert? The aggreg object is here to help.

```
summary = "Foo alert " +  aggreg['count']
```

Gives us an alert with a count. Similarly

```
summary = "Foo alert " +  aggreg['value']
```

Will append the aggregation field to the summary text. The final list aggreg['events'] contains the full log entries of all logs collected and is in general the most useful. Suppose we want one string if the tag 'foo' exists on these logs and another otherwise

```
if 'foo' in aggreg['events'][0]['_source']['tags']:
  summary = "Foo alert"
else:
  summary = "Bar alert"
```

All source log data is held within the ['_source'] and [0] represents the first log found. Beware that no specific ordering of the logs is guaranteed and so [0] may be first, last, or otherwise chronologically.

4.5.9 Questions?

Feel free to file a github issue in this repository if you find yourself with a question not answered here. Likely the answer will help someone else and will help us improve the docs.

4.5.10 Resources

Python for Beginners <https://www.python.org/about/gettingstarted/>

CHAPTER 5

MozDef for AWS

What is MozDef for AWS

Cloud based MozDef is an opinionated deployment of the MozDef services created in 2018 to help AWS users ingest CloudTrail, GuardDuty, and provide security services.

5.1 Feedback

MozDef for AWS is new and we'd love your feedback. Try filing GitHub issues here in the repository or connect with us in the Mozilla Discourse Security Category.

https://discourse.mozilla.org/c/security

You can also take a short survey on MozDef for AWS after you have deployed it. https://goo.gl/forms/JYjTYDK45d3JdnGd2

5.2 Dependencies

MozDef requires the following:

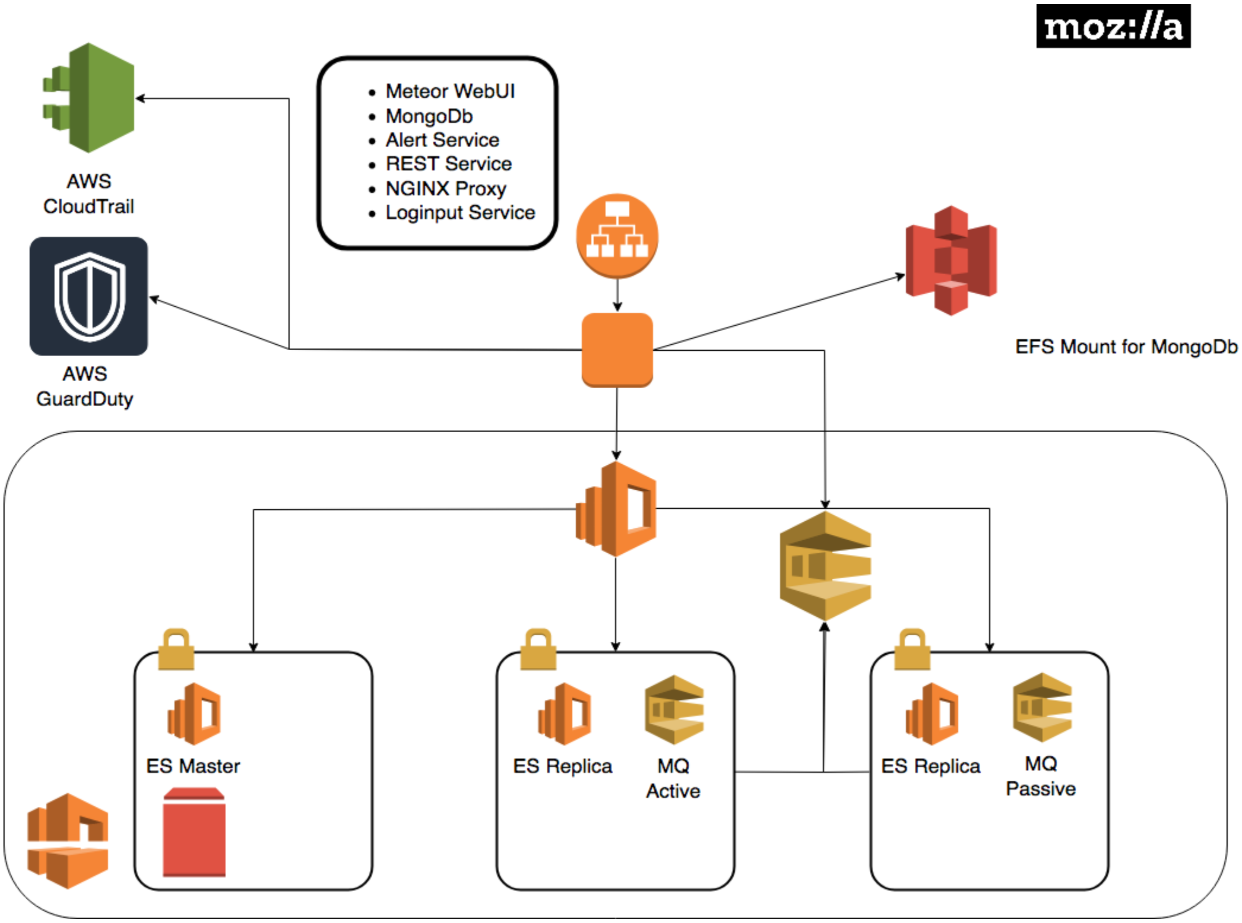
- A DNS name (e.g. cloudymozdef.security.allizom.org) which you will need to point at the IP address of the Application Load Balancer
- An OIDC Provider with ClientID, ClientSecret, and Discovery URL
  - Mozilla uses Auth0 but you can use any OIDC provider you like: Shibboleth, KeyCloak, AWS Cognito, Okta, Ping (etc.).
  - You will need to configure the redirect URI of `/redirect_uri` as allowed in your OIDC provider configuration.

- An ACM Certificate in the deployment region for your DNS name
- A VPC with three public subnets available
  - It is advised that this VPC be dedicated to MozDef or used solely for security automation.
  - The three public subnets must all be in different availability zones and have a large enough number of IP addresses to accommodate the infrastructure.
  - The VPC must have an internet gateway enabled on it so that MozDef can reach the internet.
- An SQS queue receiving GuardDuty events - At the time of writing this is not required but may be required in future.

5.3 Supported Regions

MozDef for AWS is currently only supported in us-west-2 but additional regions will be added over time.

5.4 Architecture



5.5 Deployment Process

1. Launch the one click stack and provide the requisite values.
2. Wait for the stack to complete. You'll see several nested stacks in the CloudFormation console. Once the EC2 instance is running there are still provisioning steps taking place on the instance. *Note: This may take a while*
3. Configure your DNS name to point to the application load balancer
4. Navigate to the URL you set up for MozDef. It should redirect you to the single sign on provider. If successful you'll see the MozDef UI.
5. Try navigating to ElasticSearch https://your_base_url:9090

You should see the following:

```
{
  "name" : "SMf4400",
  "cluster_name" : "656532927350:mozdef-mozdef-yemjpbnpw8xb",
  "cluster_uuid" : "_yBEIsFkQH-nEZfrFgj7mg",
  "version" : {
    "number" : "5.6.8",
    "build_hash" : "688ecce",
    "build_date" : "2018-09-11T14:44:40.463Z",
    "build_snapshot" : false,
    "lucene_version" : "6.6.1"
  },
  "tagline" : "You Know, for Search"
}
```

5. Test out Kibana at https://your_base_url:9090/_plugin/kibana/app/kibana#/discover?_g=()

5.6 Troubleshooting

To view logs on the ec2 instance

1. Determine the name/IP of the autoscaled EC2 instance via the command line or web console
2. SSH into that EC2 instance as the `ec2-user` user using the SSH keypair that you set as the `KeyName` parameter in CloudFormation
3. List out all the containers with

   ```
   sudo docker container ls
   ```

4. Tail logs from the container you'd like to examine with

   ```
   # show both the access logs and the error logs
   sudo docker logs --follow NAME_OF_CONTAINER
   # show only the error logs
   docker logs --follow NAME_OF_CONTAINER >/dev/null
   ```

   where `NAME_OF_CONTAINER` is the container name or ID that you found in the step above

5. To enter the environment for that container run

   ```
   sudo docker exec --interactive --tty NAME_OF_CONTAINER /bin/bash
   ```

6. To view the environment variables being made available to the containers view the file `/opt/mozdef/docker/compose/cloudy_mozdef.env`

5.7 Using MozDef

Refer back to our other docs on how to use MozDef for general guidance. Cloud specific instructions will evolve here. If you saw something about MozDef for AWS at re: Invent 2018 and you want to contribute we'd love your PRs.

CHAPTER 6

Development

6.1 Code

6.1.1 Plugins

Plugins are supported in several places: Event Processing and the REST api.

Event Processing

The front-end event processing portion of MozDef supports python mq plugins to allow customization of the input chain. Plugins are simple python modules than can register for events with a priority, so they only see events with certain dictionary items/values and will get them in a predefined order.

To create a plugin, make a python class that presents a registration dictionary and a priority as follows:

```
class message(object):
    def __init__(self):
        '''register our criteria for being passed a message
           as a list of lower case strings or values to match with an event's␣
↪dictionary of keys or values
           set the priority if you have a preference for order of plugins to run.
           0 goes first, 100 is assumed/default if not sent
        '''
        self.registration = ['sourceipaddress', 'destinationipaddress']
        self.priority = 20
```

Message Processing

To process a message, define an onMessage function within your class as follows:

```
def onMessage(self, message, metadata):
    #do something interesting with the message or metadata
    return (message, metadata)
```

The plugin will receive a copy of the incoming event as a python dictionary in the ‘message’ variable. The plugin can do whatever it wants with this dictionary and return it to MozDef. Plugins will be called in priority order 0 to 100 if the incoming event matches their registration criteria. i.e. If you register for sourceipaddress you will only get events containing the sourceipaddress field.

If you return the message as None (i.e. message=None) the message will be dropped and not be processed any further. If you modify the metadata the new values will be used when the message is posted to elastic search. You can use this to assign custom document types, set static document _id values, etc.

Plugin Registration

Simply place the .py file in the plugins directory where the esworker.py is located, restart the esworker.py process and it will recognize the plugin and pass it events as it sees them.

REST Plugins

The REST API for MozDef also supports rest plugins which allow you to customize your handling of API calls to suit your environment. Plugins are simple python modules than can register for REST endpoints with a priority, so they only see calls for that endpoint and will get them in a predefined order.

To create a REST API plugin simply create a python class that presents a registration dictionary and priority as follows:

```
class message(object):
    def __init__(self):
        '''register our criteria for being passed a message
           as a list of lower case strings to match with an rest endpoint
           (i.e. blockip matches /blockip)
           set the priority if you have a preference for order of plugins
           0 goes first, 100 is assumed/default if not sent

           Plugins will register in Meteor with attributes:
           name: (as below)
           description: (as below)
           priority: (as below)
           file: "plugins.filename" where filename.py is the plugin code.

           Plugin gets sent main rest options as:
           self.restoptions
           self.restoptions['configfile'] will be the .conf file
           used by the restapi's index.py file.

        '''

        self.registration = ['blockip']
        self.priority = 10
        self.name = "Banhammer"
        self.description = "BGP Blackhole"
```

The registration is the REST endpoint for which your plugin will receive a copy of the request/response objects to use or modify. The priority allows you to order your plugins if needed so that they operate on data in a defined pattern. The name and description are passed to the Meteor UI for use in dialog boxes, etc so the user can make choices when needed to include/exclude plugins. For example the /blockip endpoint allows you to register multiple methods of blocking an IP to match your environment: firewalls, BGP tables, DNS blackholes can all be independently implemented and chosen by the user at run time.

Message Processing

To process a message, define an onMessage function within your class as follows:

```
def onMessage(self, request, response):
    '''
    request: https://bottlepy.org/docs/dev/api.html#the-request-object
    response: https://bottlepy.org/docs/dev/api.html#the-response-object

    '''
    response.headers['X-PLUGIN'] = self.description
```

It's a good idea to add your plugin to the response headers if it acts on a message to facilitate troubleshooting. Other than that, you are free to perform whatever processing you need within the plugin being sure to return the request, response object once done:

```
return (request, response)
```

Plugin Registration

Simply place the .py file in the rest/plugins directory, restart the REST API process and it will recognize the plugin and pass it events as it sees them.

Alert Plugins

The alert pipeline also supports alert plugins which allow you to modify an alert's properties while the alert is "firing" (before it is saved into Elasticsearch/sent to alert actions).

Create a sample plugin in alerts/plugins:

```
class message(object):
    def __init__(self):
        '''
        adds a new field 'testing'
        to the alert if sourceipaddress is 127.0.0.1
        '''

        self.registration = "sourceipaddress"
        self.priority = 1
```

This plugin's onMessage function will get executed every time an alert has "sourceipaddress" as either a key or a value.

Message Processing

To process a message, define an onMessage function within your class as follows:

```
def onMessage(self, message):
    if 'sourceipaddress' in message && message['sourceipaddress'] == '127.0.0.1':
        message['testing'] = True
    return message
```

It's worth noting that this is a blocking mechanism, so if this function is reaching out to external resources, the alert will not "fire" until it's execution has finished. It may be preferred to use an alert action instead in cases where you don't need to modify the alert, but want to trigger an API somewhere.

Plugin Registration

Simply place the .py file in the alerts/plugins directory, restart the alerts process and it will recognize the plugin and pass it alerts based on registration.

6.1.2 Actions

Actions are currently supported at the end of the alert pipeline.

Alert Action Processing

Alert actions run at the very end of the alert pipeline after the alert is already created, and are non blocking (meaning they also don't have the ability to modify alerts inline).

```
class message(object):
    def __init__(self):
        '''
        triggers when a geomodel alert is generated
        '''
        self.alert_classname = 'AlertGeomodel'
        self.registration = 'geomodel'
        self.priority = 1
```

Alert Trigger

```
def onMessage(self, message):
    print(message)
    return message
```

Plugin Registration

Simply place the .py file in the alert actions directory.

6.1.3 Tests

Our test suite builds and runs entirely in *docker*, so a Docker daemon is required to be running locally. The test suite requires a local Elasticsearch and RabbitMQ service to run, but will be started automatically as containers as part of the tests make targets.

Run tests

To run our entire test suite, simply run:

```
make tests
```

If you want to only run a specific test file/directory, you can specify the *TEST_CASE* parameter:

```
make tests TEST_CASE=tests/mq/plugins/test_github_webhooks.py
```

Note: If you end up with a clobbered ES index, or anything like that which might end up in failing tests, you can clean the environment with *make clean*. Then run the tests again.

6.2 Mozdef_util Library

We provide a library used to interact with MozDef components.

6.2.1 Installation

If you're using Mac OS X:

```
git clone https://github.com/mozilla/mozdef mozdef
cd ./mozdef
export PYCURL_SSL_LIBRARY=openssl
export LDFLAGS=-L/usr/local/opt/openssl/lib;export CPPFLAGS=-I/usr/local/opt/openssl/
↪include
pip install -r requirements.txt
```

If you're using CentOS:

```
git clone https://github.com/mozilla/mozdef mozdef
cd ./mozdef
PYCURL_SSL_LIBRARY=nss pip install -r requirements.txt
```

6.2.2 Usage

Connecting to Elasticsearch

```
from mozdef_util.elasticsearch_client import ElasticsearchClient
es_client = ElasticsearchClient("http://127.0.0.1:9200")
```

Creating/Updating Documents

Create a new Event

```
event_dict = {
    "example_key": "example value"
}
es_client.save_event(body=event_dict)
```

Update an existing event

```
event_dict = {
    "example_key": "example new value"
}
# Assuming 12345 is the id of the existing entry
es_client.save_event(body=event_dict, doc_id="12345")
```

Create a new alert

```
alert_dict = {
    "example_key": "example value"
}
es_client.save_alert(body=alert_dict)
```

Update an existing alert

```
alert_dict = {
    "example_key": "example new value"
}
# Assuming 12345 is the id of the existing entry
es_client.save_alert(body=alert_dict, doc_id="12345")
```

Create a new generic document

```
document_dict = {
    "example_key": "example value"
}
es_client.save_object(index='randomindex', body=document_dict)
```

Update an existing document

```
document_dict = {
    "example_key": "example new value"
}
# Assuming 12345 is the id of the existing entry
es_client.save_object(index='randomindex', body=document_dict, doc_id="12345")
```

Bulk Importing

```
1 from mozdef_util.elasticsearch_client import ElasticsearchClient
2 es_client = ElasticsearchClient("http://127.0.0.1:9200", bulk_amount=30, bulk_refresh_time=5)
3 es_client.save_event(body={'key': 'value'}, bulk=True)
```

- Line 2: bulk_amount (defaults to 100), specifies how many messages should sit in the bulk queue before they get written to elasticsearch
- Line 2: bulk_refresh_time (defaults to 30), is the amount of time that a bulk flush is forced
- Line 3: bulk (defaults to False) determines if an event should get added to a bulk queue

Searching for documents

Simple search

```
from mozdef_util.query_models import SearchQuery, TermMatch, ExistsMatch

search_query = SearchQuery(hours=24)
must = [
    TermMatch('category', 'brointel'),
    ExistsMatch('seenindicator')
]
search_query.add_must(must)
results = search_query.execute(es_client, indices=['events','events-previous'])
```

SimpleResults

When you perform a “simple” search (one without any aggregation), a SimpleResults object is returned. This object is a dict, with the following format:

Key	Description
hits	Contains an array of documents that matched the search query
meta	Contains a hash of fields describing the search query (Ex: if the query timed out or not)

Example simple result:

```
{
 'hits': [
   {
     '_id': u'cp5ZsOgLSu6tHQm5jAZW1Q',
     '_index': 'events-20161005',
     '_score': 1.0,
     '_source': {
       'details': {
         'information': 'Example information'
       },
       'category': 'excategory',
       'summary': 'Test Summary',
       'type': 'event'
     }
   }
```

(continues on next page)

(continued from previous page)

```
 ],
 'meta': {'timed_out': False}
}
```

Aggregate search

```
from mozdef_util.query_models import SearchQuery, TermMatch, Aggregation

search_query = SearchQuery(hours=24)
search_query.add_must(TermMatch('category', 'brointel'))
search_query.add_aggregation(Aggregation('source'))
results = search_query.execute(es_client)
```

AggregatedResults

When you perform an aggregated search (Ex: give me a count of all different ip addresses are in the documents that match a specific query), a AggregatedResults object is returned. This object is a dict, with the following format:

Key	Description
aggregations	Contains the aggregation results, grouped by field name
hits	Contains an array of documents that matched the search query
meta	Contains a hash of fields describing the search query (Ex: if the query timed out or not)

```
{
 'aggregations': {
    'ip': {
      'terms': [
        {
          'count': 2,
          'key': '1.2.3.4'
        },
        {
          'count': 1,
          'key': '127.0.0.1'
        }
      ]
    }
  },
  'hits': [
    {
      '_id': u'LcdS2-koQWeICOpbOT__gA',
      '_index': 'events-20161005',
      '_score': 1.0,
      '_source': {
        'details': {
          'information': 'Example information'
        },
        'ip': '1.2.3.4',
        'summary': 'Test Summary',
        'type': 'event'
      }
    },
```

(continues on next page)

(continued from previous page)

```
    {
      '_id': u'F1dLS66DR_W3v7ZWlX4Jwg',
      '_index': 'events-20161005',
      '_score': 1.0,
      '_source': {
        'details': {
          'information': 'Example information'
        },
        'ip': '1.2.3.4',
        'summary': 'Test Summary',
        'type': 'event'
      }
    },
    {
      '_id': u'G1nGdxqoT6eXkL5KIjLecA',
      '_index': 'events-20161005',
      '_score': 1.0,
      '_source': {
        'details': {
          'information': 'Example information'
        },
        'ip': '127.0.0.1',
        'summary': 'Test Summary',
        'type': 'event'
      }
    }
  ],
  'meta': {
    'timed_out': False
  }
}
```

Match/Query Classes

ExistsMatch

Checks to see if a specific field exists in a document

```
from mozdef_util.query_models import ExistsMatch

ExistsMatch("randomfield")
```

TermMatch

Checks if a specific field matches the key

```
from mozdef_util.query_models import TermMatch

TermMatch("details.ip", "127.0.0.1")
```

TermsMatch

Checks if a specific field matches any of the keys

```
from mozdef_util.query_models import TermsMatch

TermsMatch("details.ip", ["127.0.0.1", "1.2.3.4"])
```

WildcardMatch

Allows regex to be used in looking for documents that a field contains all or part of a key

```
from mozdef_util.query_models import WildcardMatch

WildcardMatch('summary', 'test*')
```

PhraseMatch

Checks if a field contains a specific phrase (includes spaces)

```
from mozdef_util.query_models import PhraseMatch

PhraseMatch('summary', 'test run')
```

BooleanMatch

Used to apply specific "matchers" to a query. This will unlikely be used outside of SearchQuery.

```
from mozdef_util.query_models import ExistsMatch, TermMatch, BooleanMatch

must = [
    ExistsMatch('details.ip')
]
must_not = [
    TermMatch('type', 'alert')
]

BooleanMatch(must=must, should=[], must_not=must_not)
```

MissingMatch

Checks if a field does not exist in a document

```
from mozdef_util.query_models import MissingMatch

MissingMatch('summary')
```

RangeMatch

Checks if a field value is within a specific range (mostly used to look for documents in a time frame)

```python
from mozdef_util.query_models import RangeMatch

RangeMatch('utctimestamp', "2016-08-12T21:07:12.316450+00:00", "2016-08-13T21:07:12.316450+00:00")
```

QueryStringMatch

Uses a custom query string to generate the "match" based on (Similar to what you would see in kibana)

```python
from mozdef_util.query_models import QueryStringMatch

QueryStringMatch('summary: test')
```

SubnetMatch

Checks if an IP field is within the bounds of a subnet

```python
from mozdef_util.query_models import SubnetMatch

SubnetMatch('details.sourceipaddress', '10.1.1.0/24')
```

Aggregation

Used to aggregate results based on a specific field

```python
from mozdef_util.query_models import Aggregation, SearchQuery, ExistsMatch

search_query = SearchQuery(hours=24)
must = [
    ExistsMatch('seenindicator')
]
search_query.add_must(must)
aggr = Aggregation('details.ip')
search_query.add_aggregation(aggr)
results = search_query.execute(es_client, indices=['events','events-previous'])
```

6.3 Continuous Integration and Continuous Deployment

6.3.1 Overview

Each git commit to the *master* branch in GitHub triggers both the TravisCI automated tests as well as the AWS CodeBuild building. Each git tag applied to a git commit triggers a CodeBuild build.

6.3.2 Travis CI

Travis CI runs tests on the MozDef code base with each commit to *master*. The results can be seen on the Travis CI MozDef dashboard

The Test Sequence

- Travis CI creates webhooks when first setup which allow commits to the MozDef GitHub repo to trigger Travis.
- When a commit is made to MozDef, Travis CI follows the instructions in the .travis.yml file.
- *.travis.yml* installs *docker-compose* in the *before_install* phase.
- In the *install* phase, Travis runs the build-tests make target which calls *docker-compose build* on the docker/compose/docker-compose-tests.yml file which builds a few docker containers to use for testing.
- In the *script* phase, Travis runs the tests make target which
  - calls the *build-tests* make target which again runs *docker-compose build* on the docker/compose/docker-compose-tests.yml file.
  - calls the run-tests make target which.
    - calls the run-tests-resources make target which starts the docker containers listed in docker/compose/docker-compose-tests.yml.
    - runs *flake8* with the .flake8 config file to check code style.
    - runs *py.test tests* which runs all the test cases.

6.3.3 AWS CodeBuild

Enabling GitHub AWS CodeBuild Integration

Onetime Manual Step

The steps to establish a GitHub CodeBuild integration unfortunately require a onetime manual step be done before using CloudFormation to configure the integration. This onetime manual step **need only happen a single time for a given AWS Account + Region**. It need **not be performed with each new CodeBuild project or each new GitHub repo**

1. Manually enable the GitHub integration in AWS CodeBuild using the dedicated, AWS account specific, GitHub service user.
   1. A service user is needed as AWS CodeBuild can only integrate with GitHub from one AWS account in one region with a single GitHub user. Technically you could use different users for each region in a single AWS account, but for simplicity limit yourself to only one GitHub user per AWS account (instead of one GitHub user per AWS account per region)
   2. To do the one time step of integrating the entire AWS account in that region with the GitHub service user
      1. Browse to CodeBuild in AWS and click Create Project
      2. Navigate down to `Source` and set `Source Provider` to `GitHub`
      3. For `Repository` select `Connect with a GitHub personal access token`
      4. Enter the persona access token for the GitHub service user. If you haven't created one do so and grant it `repo` and `admin:repo_hook`

5. Click `Save Token`
6. Abort the project setup process by clicking the `Build Projects` breadcrumb at the top. This "Save Token" step was the only thing you needed to do in that process

Grant the GitHub service user access to the GitHub repository

1. As an admin of the GitHub repository go to that repositories settings, select Collaborators and Teams, and add the GitHub service user to the repository
2. Set their access level to `Admin`
3. Copy the invite link, login as the service user and accept the invitation

Deploy CloudFormation stack creating CodeBuild project

Deploy the `mozdef-cicd-codebuild.yml` CloudFormation template to create the CodeBuild project and IAM Role

The Build Sequence

- A branch is merged into *master* in the GitHub repo or a version git tag is applied to a commit.
- GitHub emits a webhook event to AWS CodeBuild indicating this.
- AWS CodeBuild considers the Filter Groups configured to decide if the tag or branch warrants triggering a build. These Filter Groups are defined in the `mozdef-cicd-codebuild.yml` CloudFormation template. Assuming the tag or branch are acceptable, CodeBuild continues.
- AWS CodeBuild reads the buildspec.yml file to know what to do.
- The *install* phase of the *buildspec.yml* fetches packer and unzips it.
  - *packer* is a tool that spawns an ec2 instance, provisions it, and renders an AWS Machine Image (AMI) from it.
- The *build* phase of the *buildspec.yml* runs the cloudy_mozdef/ci/deploy script in the AWS CodeBuild Ubuntu 14.04 environment.
- The *deploy* script calls the build-from-cwd target of the *Makefile* which calls *docker-compose build* on the docker-compose.yml file, building the docker images in the AWS CodeBuild environment. These are built both so they can be consumed later in the build by packer and also for use by developers and the community.
- *deploy* then calls the docker-push-tagged make target which calls
  - the tag-images make target which calls the cloudy_mozdef/ci/docker_tag_or_push tag script which applies a docker image tag to the local image that was just built by AWS CodeBuild.
  - the hub-tagged make target which calls the cloudy_mozdef/ci/docker_tag_or_push push script which
    - Uploads the local image that was just built by AWS CodeBuild to DockerHub. If the branch being built is *master* then the image is uploaded both with a tag of *master* as well as with a tag of *latest*.
    - If the branch being built is from a version tag (e.g. *v1.2.3*) then the image is uploaded with only that version tag applied.
- The *deploy* script next calls the packer-build-github make target in the cloudy_mozdef/Makefile which calls the ci/pack_and_copy script which does the following steps.
  - Calls packer which launches an ec2 instance, executing a bunch of steps and and producing an AMI

  - Shares the resulting AMI with the AWS Marketplace account
  - Copies the resulting AMI to a list of additional AWS regions
  - Copies the tags from the original AMI to these copied AMIs in other regions
  - Shares the AMIs in these other regions with the AWS Marketplace account
  - Creates a blob of YAML which contains the AMI IDs. This blob will be used in the CloudFormation templates

- When *ci/pack_and_copy* calls packer, packer launches an ec2 instance based on the configuration in cloudy_mozdef/packer/packer.json
  - Within this ec2 instance, packer clones the MozDef GitHub repo and checks out the branch that triggered this build.
  - Packer replaces all instances of the word *latest* in the docker-compose-cloudy-mozdef.yml file with either the branch *master* or the version tag (e.g. *v1.2.3*).
  - Packer runs *docker-compose pull* on the docker-compose-cloudy-mozdef.yml file to pull down both the docker images that were just built by AWS CodeBuild and uploaded to Dockerhub as well as other non MozDef docker images.
- After packer completes executing the steps laid out in *packer.json* inside the ec2 instance, it generates an AMI from that instance and continues with the copying, tagging and sharing steps described above.
- Now back in the AWS CodeBuild environment, the *deploy* script continues by calling the publish-versioned-templates make target which runs the ci/publish_versioned_templates script which
  - injects the AMI map yaml blob produced earlier into the mozdef-parent.yml CloudFormation template so that the template knows the AMI IDs of that specific branch of code.
  - uploads the CloudFormation templates to S3 in a directory either called *master* or the tag version that was built (e.g. *v1.2.3*).

CHAPTER 7

References

7.1 Screenshots

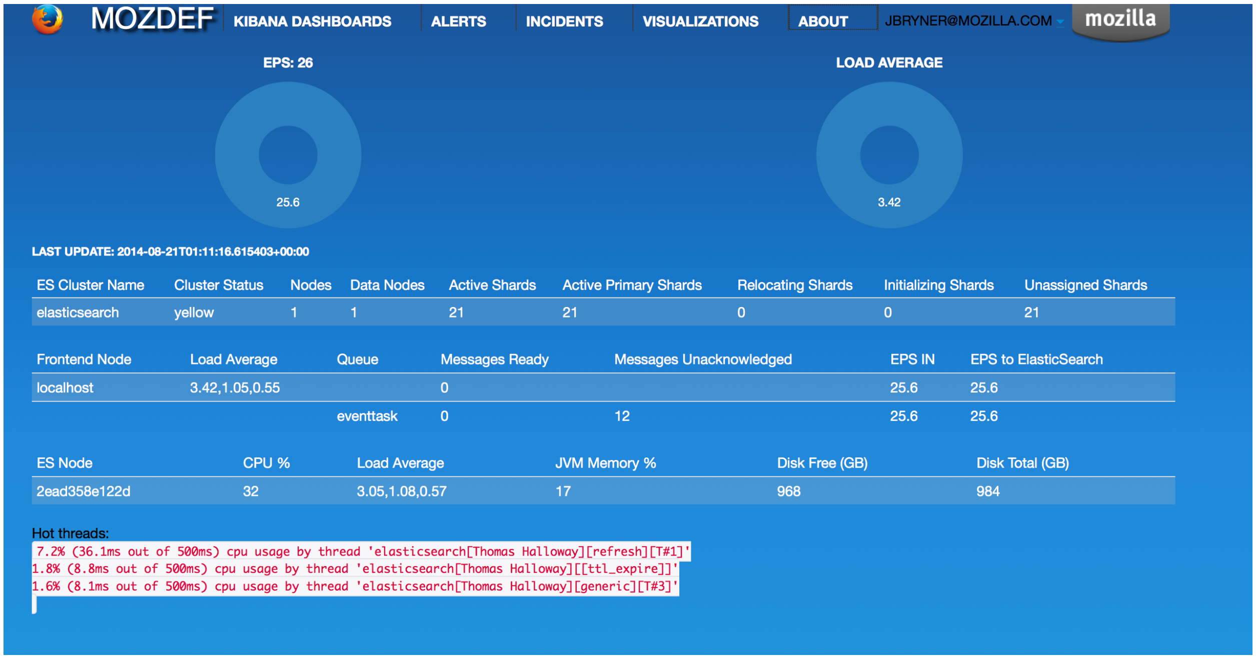
Here are a few screen captures of key portions of the MozDef user interface.

7.1.1 Health and Status

MozDef includes an integrated health and status screen under the 'about' menu showing key performance indicators like events per second from rabbit-mq and elastic search cluster health.

You can have as many front-end processors running rabbit-mq as you like in whatever geographic distribution makes sense for your environment. The hot threads section shows you what your individual elastic search nodes are up to.

The entire display updates in real time as new information is retrieved.



LAST UPDATE: 2014-08-21T01:11:16.615403+00:00

ES Cluster Name	Cluster Status	Nodes	Data Nodes	Active Shards	Active Primary Shards	Relocating Shards	Initializing Shards	Unassigned Shards
elasticsearch	yellow	1	1	21	21	0	0	21

Frontend Node	Load Average	Queue	Messages Ready	Messages Unacknowledged	EPS IN	EPS to ElasticSearch
localhost	3.42,1.05,0.55		0		25.6	25.6
		eventtask	0	12	25.6	25.6

ES Node	CPU %	Load Average	JVM Memory %	Disk Free (GB)	Disk Total (GB)
2ead358e122d	32	3.05,1.08,0.57	17	968	984

Hot threads:

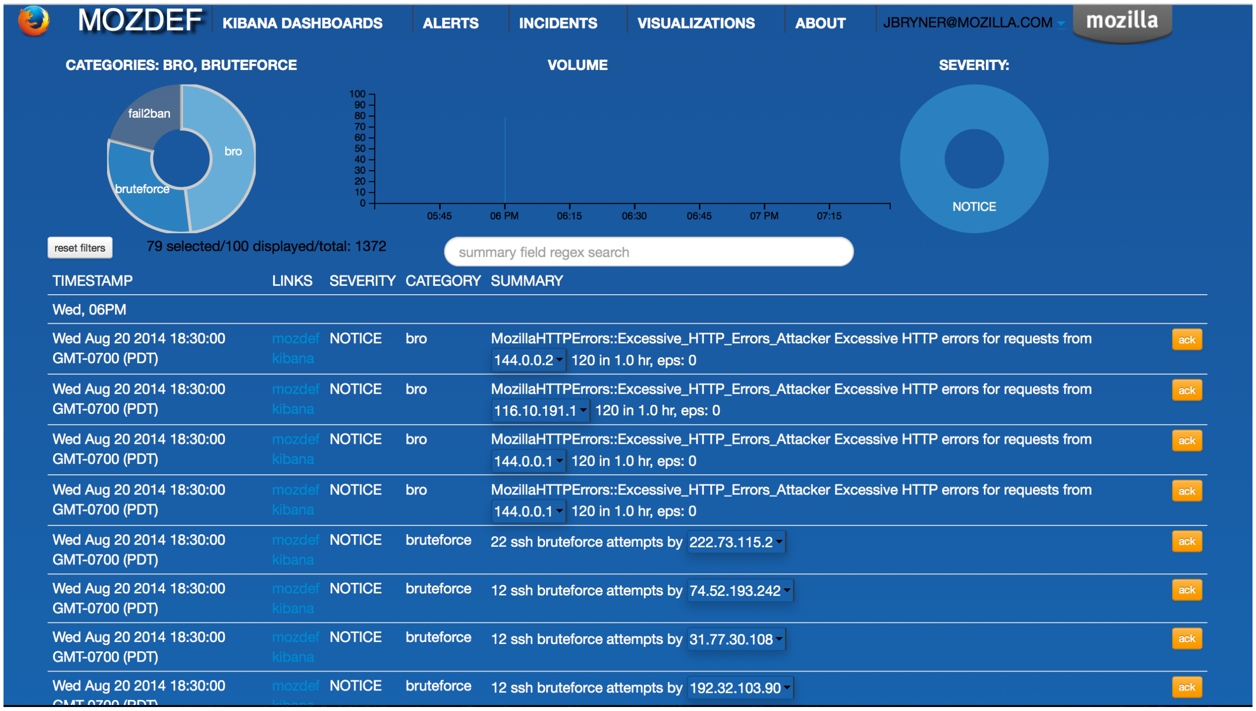
```
7.2% (36.1ms out of 500ms) cpu usage by thread 'elasticsearch[Thomas Halloway][refresh][T#1]'
1.8% (8.8ms out of 500ms) cpu usage by thread 'elasticsearch[Thomas Halloway][[ttl_expire]]'
1.6% (8.1ms out of 500ms) cpu usage by thread 'elasticsearch[Thomas Halloway][generic][T#3]'
```

7.1.2 Alerts

Alerts are simply python jobs run as celery tasks that query elastic search for either individual events, or correlate multiple events into an alert.

The alerts screen shows the latest 100 alerts and allows interactive filtering by category, severity, time frame and free-form regex.

The display updates in real time as new alerts are received and any IP address in an alert is decorated with a menu allowing you to query whois, dshield, CIF, etc to get context on the item. If your facilities include blocking, you can also integrate that into the menu to allow you to block an IP directly from this screen.



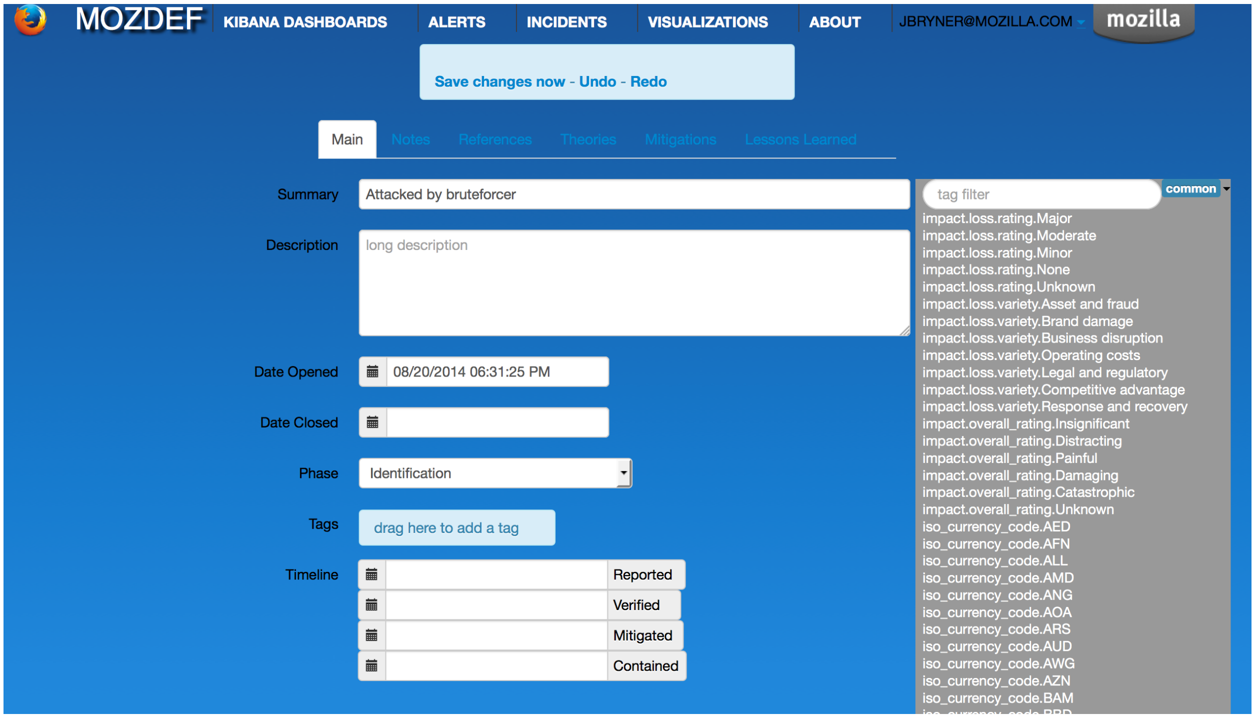
79 selected/100 displayed/total: 1372

TIMESTAMP	LINKS	SEVERITY	CATEGORY	SUMMARY	
Wed, 06PM					
Wed Aug 20 2014 18:30:00 GMT-0700 (PDT)	mozdef kibana	NOTICE	bro	MozillaHTTPErrors::Excessive_HTTP_Errors_Attacker Excessive HTTP errors for requests from 144.0.0.2 120 in 1.0 hr, eps: 0	ack
Wed Aug 20 2014 18:30:00 GMT-0700 (PDT)	mozdef kibana	NOTICE	bro	MozillaHTTPErrors::Excessive_HTTP_Errors_Attacker Excessive HTTP errors for requests from 116.10.191.1 120 in 1.0 hr, eps: 0	ack
Wed Aug 20 2014 18:30:00 GMT-0700 (PDT)	mozdef kibana	NOTICE	bro	MozillaHTTPErrors::Excessive_HTTP_Errors_Attacker Excessive HTTP errors for requests from 144.0.0.1 120 in 1.0 hr, eps: 0	ack
Wed Aug 20 2014 18:30:00 GMT-0700 (PDT)	mozdef kibana	NOTICE	bro	MozillaHTTPErrors::Excessive_HTTP_Errors_Attacker Excessive HTTP errors for requests from 144.0.0.1 120 in 1.0 hr, eps: 0	ack
Wed Aug 20 2014 18:30:00 GMT-0700 (PDT)	mozdef kibana	NOTICE	bruteforce	22 ssh bruteforce attempts by 222.73.115.2	ack
Wed Aug 20 2014 18:30:00 GMT-0700 (PDT)	mozdef kibana	NOTICE	bruteforce	12 ssh bruteforce attempts by 74.52.193.242	ack
Wed Aug 20 2014 18:30:00 GMT-0700 (PDT)	mozdef kibana	NOTICE	bruteforce	12 ssh bruteforce attempts by 31.77.30.108	ack
Wed Aug 20 2014 18:30:00 GMT-0700 (PDT)	mozdef kibana	NOTICE	bruteforce	12 ssh bruteforce attempts by 192.32.103.90	ack

7.1.3 Incident Handling

MozDef includes an integrated, real time incident handling facility that allows multiple responders to work collaboratively on a security incident. As they add information to the incident they are able to see each others changes as they happen, in real time.

MozDef includes integration into the VERIS classification system to quickly tag incidents with metadata by dragging tags onto the incident which allows you to aggregate metrics about your incidents.


MOZDEF
KIBANA DASHBOARDS
ALERTS
INCIDENTS
VISUALIZATIONS
ABOUT
JBRYNER@MOZILLA.COM
mozilla
Save changes now - Undo - Redo
Main
Notes
References
Theories
Mitigations
Lessons Learned
tag filter
common
impact.loss.rating.Major
impact.loss.rating.Moderate
impact.loss.rating.Minor
impact.loss.rating.None
impact.loss.rating.Unknown
impact.loss.variety.Asset and fraud
impact.loss.variety.Brand damage
impact.loss.variety.Business disruption
impact.loss.variety.Operating costs
impact.loss.variety.Legal and regulatory
impact.loss.variety.Competitive advantage
impact.loss.variety.Response and recovery
impact.overall_rating.Insignificant
impact.overall_rating.Distracting
impact.overall_rating.Painful
impact.overall_rating.Damaging
impact.overall_rating.Catastrophic
impact.overall_rating.Unknown
iso_currency_code.AED
iso_currency_code.AFN
iso_currency_code.ALL
iso_currency_code.AMD
iso_currency_code.ANG
iso_currency_code.AOA
iso_currency_code.ARS
iso_currency_code.AUD
iso_currency_code.AWG
iso_currency_code.AZN
iso_currency_code.BAM

Summary: Attacked by bruteforcer

Description: long description

Date Opened: 08/20/2014 06:31:25 PM

Date Closed:

Phase: Identification

Tags: drag here to add a tag

Timeline: Reported, Verified, Mitigated, Contained

7.2 GeoModel Version 0.1 Specification

The first release version of GeoModel will be a minimum viable product (MVP) containing features that replace the functionality of the existing implementation along with a few new requirements.

7.2.1 Terminology

Locality

The locality of a user is a geographical region from which most of that user's online activity originates.

Authenticated Actions

An event produced as a result of any user taking any action that required they be authenticated e.g. authenticating to a service with Duo, activity in the AWS web console etc.

7.2.2 Primary Interface

GeoModel v0.1 is an alert built into MozDef that:

1. Processes authentication-related events.
2. Updates user locality information.
3. Emits alerts when some specific conditions are met.

Data Stores

GeoModel interacts with MozDef to both query for events as well as store new alerts.

GeoModel also maintains its own user locality information. Version 0.1 will store this information in the same ElasticSearch instance that MozDef uses, under a configured index.

Functional Components

GeoModel v0.1 can be thought of as consisting of two core "components" that are each responsible for a distinct set of responsibilities. These two components interact in a pipeline.

Because GeoModel v0.1 is implemented as an Alert in MozDef, it is essentially a distinct Python program run by MozDef's `AlertTask` scheduler.

Analysis Engine

The first component handles the analysis of events pertaining to authenticated actions made by users. These events are retrieved from MozDef and analyzed to determine locality of users which is then persisted in a data store.

This component has the following responsibilities:

1. Run configured queries to retrieve events describing authenticated actions taken by users from MozDef.
2. Load locality state from ElasticSearch.
3. Remove outdated locality information.
4. Update locality state with information from retrieved events.

Alert Emitter

The second component handles the creation of alerts and communicating of those alerts to MozDef.

This component has the following responsibilities:

1. Inspect localities produced by the Analysis Engine to produce alerts.
2. Store alerts in MozDef's ElasticSearch instance.

The Alert Emitter will, given a set of localities for a user, produce an alert if and only if both:

1. User activity is found to originate from a location outside of all previously known localities.
2. It would **not** be possible for the user to have travelled to a new locality from the one they were last active in.

Data Models

The following models describe what data is required to implement the features that each component is responsible for. They are described using a JSON-based format where keys indicidate the names of values and values are strings containing those values' types, which are represented using TypeScript notation. We use this notation because configuration data as well as data stored in ElasticSearch are represented as JSON and JSON-like objects respectively.

General Configuration

The top-level configuration for GeoModel version 0.1 must contain the following.

```
{
  "localities": {
    "es_index": string,
    "valid_duration_days": number,
    "radius_kilometres": number
  },
  "events": {
    "search_window": object,
    "lucene_query": string,
  },
  "whitelist": {
    "users": Array<string>,
    "cidrs": Array<string>
  }
}
```

Using the information above, GeoModel can determine:

- What index to store locality documents in.
- What index to read events from.
- What index to write alerts to.
- What queries to run in order to retrieve a complete set of events.
- When a user locality is considered outdated and should be removed.
- The radius that localities should have.
- Whitelisting rules to apply.

In the above, note that `events.queries` describes an array of objects. Each of these objects are expected to contain a query for ElasticSearch using Lucene syntax. The `username` field is expected to be a string describing the path into the result dictionary your query will return that will produce the username of the user taking an authenticated action.

The `search_window` object can contain any of the keywords passed to Python's timedelta constructor.

So for example the following:

```
{
  "events": [
    {
      "search_window": {
        "minutes": 30
      },
      "lucene_query": "tags:auth0",
      "username_path": "details.username"
    }
  ]
}
```

would query ElasticSearch for all events tagged `auth0` and try to extract the `username` from `result["details"]["username"]` where `result` is one of the results produced by executing the query.

The `alerts.whitelist` portion of the configuration specifies a couple of parameters for whitelisting acitivity:

1. From any of a list of users (based on `events.queries.username`).

2. From any IPs within the range of any of a list of CIDRs.

For example, the following whitelist configurations would instruct GeoModel **not** to produce alerts for actions taken by “testuser” **or** for any users originating from an IP in either the ranges `1.2.3.0/8` and `192.168.0.0/16`.

```
{
  "alerts": {
    "whitelist": {
      "users": ["testuser"],
      "cidrs": ["1.2.3.0/8", "192.168.0.0/16"]:
    }
  }
}
```

Note however that GeoModel **will still retain locality information for whitelisted users and users originating from whitelisted IPs**.

User Locality State

GeoModel version 0.1 uses one ElasticSearch Type (similar to a table in a relational database) to represent locality information. Under this type, one document exists per user describing that user’s locality information.

```
{
  "type_": "locality",
  "username": string,
  "localities": Array<{
    "sourceipaddress": string,
    "city": string,
    "country": string,
    "lastaction": date,
    "latitude": number,
    "longitude": number,
    "radius": number
  }>
}
```

Using the information above, GeoModel can determine:

- All of the localities of a user.
- Whether a locality is older than some amount of time.
- How far outside of any localities a given location is.

Alerts

Alerts emitted to the configured index are intended to cohere to MozDef’s preferred naming scheme.

```
{
  "username": string,
  "hops": [
    {
      "origin": {
        "ip": string,
        "city": string,
        "country": string,
```

(continues on next page)

(continued from previous page)

```
        "latitude": number,
        "longitude": number,
        "geopoint": GeoPoint
      }
      "destination": {
        "ip": string,
        "city": string,
        "country": string,
        "latitude": number,
        "longitude": number,
        "geopoint": GeoPoint
      }
    }
  ]
}
```

Note in the above that the `origin.geopoint` field uses ElasticSearch's GeoPoint type.

7.2.3 User Stories

User stories here make references to the following categories of users:

- An **operator** is anyone responsible for deploying or maintaining a deployment of MozDef that includes GeoModel.
- An **investigator** is anyone responsible for viewing and taking action based on alerts emitted by GeoModel.

Potential Compromises Detected

As an investigator, I expect that if a user is found to have performed some authenticated action in one location and then, some short amount of time later, in another that an alert will be emitted by GeoModel.

Realistic Travel Excluded

As an investigator, I expect that if someone starts working somehwere, gets on a plane and continues working after arriving in their destination that an alert will **not** be emitted by GeoModel.

Diversity of Indicators

As an operator, I expect that GeoModel will fetch events pertaining to authenticated actions from new sources (Duo, Auth0, etc.) after I deploy MozDef with GeoModel configured with queries targeting those sources.

Old Data Removed Automatically

As an operator, I expect that GeoModel will forget about localities attributed to users that have not been in those geographic regions for a configured amount of time.

7.3 AWS re:invent 2018 SEC403 Presentation

- Watch our presentation on MozDef in AWS at AWS re:Invent 2018
- Read the slides

CHAPTER 8

Contributors

Here is the list of the awesome contributors helping us or that have helped us in the past:

Contributors

CHAPTER 9

License

license

CHAPTER 10

Contact

- mozdef INSERTAT mozilla.com
- #mozdef