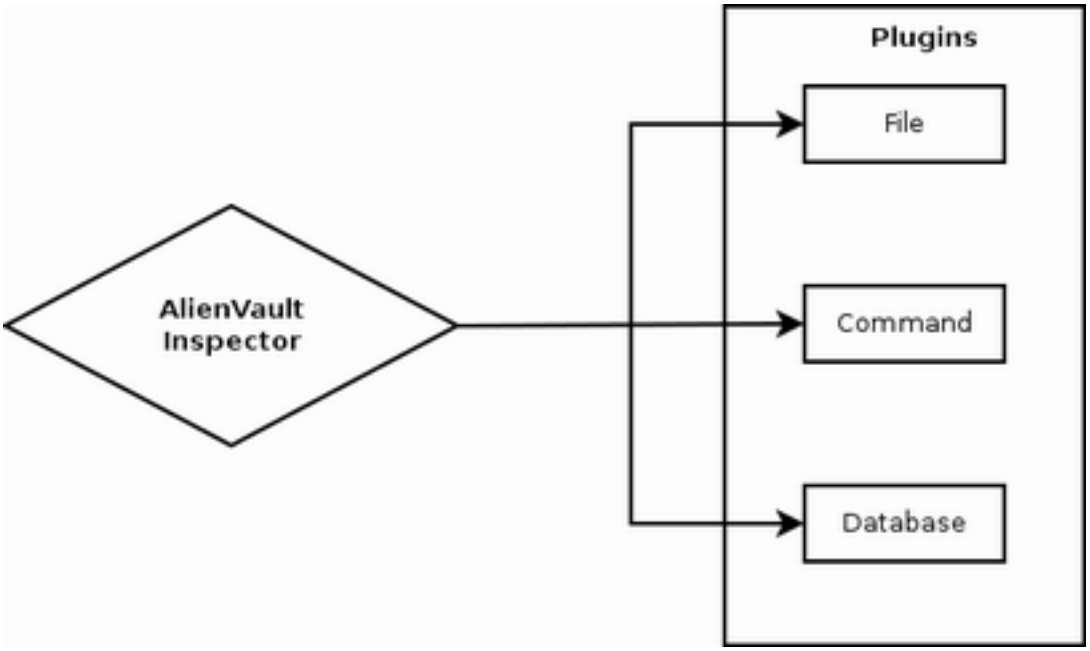
AlienVault Doctor

This document describes at a glance how the AlienVault Doctor tool works and how people can write their own plugins.

The tool

AlienVault Doctor is a command line tool that enables you to detect misconfiguration and performance issues within an AlienVault box. Is a very simple tool and all its power relies on the plugin collection available.



Plugins are all stored in a single directory. The Doctor should be able to list files in that directory, parse them and use them to check for anomalies.

As said, this tool is very simple. In fact it only has three command line options available for now. Running `alienvault-doctor -h` gives us the full list:

```
mabeledo@waterhouse ~ $ ./alienvault-doctor -h
Usage: alienvault-doctor [options]

Cleans the system from your mistakes

Options:
```

```
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -P PLUGIN_DIR, --plugin-dir=PLUGIN_DIR
                        Directory where plugins are stored [default:
                        /etc/ossim/doctor/plugins]

  Output options:
    -o OUTPUT_TYPE, --output=OUTPUT_TYPE
                        Output type [default: none]
    -p OUTPUT_PATH, --path=OUTPUT_PATH
                        Output file path [default: /var/ossim/doctor]
    -f OUTPUT_FILE_PREFIX, --file-prefix=OUTPUT_FILE_PREFIX
                        Output file prefix [default: data]
```

All general options are self explanatory. Omitting the `-P` option enforces a default directory, in this case `/etc/ossim/doctor/plugins`.
Output options may be used only for interprocess communication, as data will be written in JSON format. `-o` accepts two types, `none` and `file`. Choosing `file` allows to specify a path using `-p`, where the file is written. `-f` is for the output file name prefix.

The plugins

There are three plugin types: `file`, `command` and `db`. The first two types are similar, as they both rely on regular expression matching, while the third interacts directly with a MySQL database and parses data in a different way. Plugins are divided in two or more sections, where `properties` contains more general options and the rest are testing rules. Data retrieved by these plugins can be of three types: `@int` for integers, `@char` for single characters and `@string` for any string, and rules can be made to find values equal, less or greater than, or event those which match to a pattern written as a regular expression.

In this naive example, the Doctor tool checks for excessive IO waiting using vmstat.

```
# VMStat plugin
# This plugin checks for excessive swap memory usage.
[properties]
name=vmstat
type=command
profiles=Server:>4.0
command=/usr/bin/vmstat

[wait]
pattern=([0-9]+)$
conditions=@int:0:20
```

The properties section has three mandatory fields, `name`, `type` and `profiles`, and an optional one, `cutoff`. `name` is the name of the plugin and is only used to print output data. `type` is the type of the plugin, whether `file`, `command` or `db`. `profiles` is the profile set where this particular plugin can be run. These profiles are usually found in `/etc/ossim/ossim_setup.conf`. `cutoff` is used to cut the plugin execution if a single check has failed.

The `file` type plugin

`file` and `command` plugins are, as said, quite similar. However, file plugins have an extra condition property named `checksum`. This property allows checking file integrity using any checksum supported by the OpenSSL library installed.

```
# MySQL logrotate plugin
# This plugin checks for misconfiguration of the logrotate daemon for MySQL
# log files.
[properties]
name=mysql_logrotate
type=file
filename=/etc/logrotate.d/percona-server-server-5.5
cutoff=True
profiles=Database:>4.0

[logrotate_checksum]
checksum=@sha1:068e75ca15bc4abb2bb93734c34d6403e7fab441
actions=@command:echo "This digest is fine" | mailx -s "Digest result" root

[mysql.err]
pattern=\s(/.*?mysql.err)
conditions=@string:=="/var/log/mysql/mysql.err"@or@=="/var/log/mysql.err"

[mysql.log]
pattern=\s(/.*?mysql.log)
conditions=@string:=="/var/log/mysql/mysql.log"@or@=="/var/log/mysql.log"
```

This plugin shows how `checksum` and `actions` could be set up: a correct `sha1` checksum triggers a mail action that notifies user root about the state of this logrotate configuration file. Note how the `@or@` logical connective is used, linking two or more conditions. `@string` type values are always enclosed by quotes, while relational operators such as `==` in this case are not.

The `command` type plugin

The `command` type is nearly identical to the `file` type, but rules can't have a `checksum` property.

```
# Disk usage plugin
# Check for disk usage in some important partitions.
[properties]
name=disk_usage
type=command
profiles=Server:>4.0;Database:>4.0
command=df

[root]
pattern=([0-9][0-9]?|100)%\s+/$
conditions=@int:>0@and@<80

[var]
pattern=([0-9][0-9]?|100)%\s+/var$
conditions=@int:<100

[logger]
pattern=([0-9][0-9]?|100)%\s+/var/ossim/logs$
conditions=@int:<100
```

Note how ranges are implemented using an `@and@` connective. In this case `Server` and `Database` profiles are specified, as they're the most IO intensive.

Some `command` plugins depend on Debian packages, files or kernel modules. If they're not present, the plugin won't work. This is the reason why the option `requires` is available under the `properties` section: the plugin will check for dependencies before running. In the following example, IPMI queries are only allowed if the IPMI driver is loaded.

```
# Chassis plugin
# Various rules for Chassis sensor monitoring.
[properties]
name=chassis
type=command
command=ipmi chassis status
requires=@modules:ipmi_watchdog,ipmi_msghandler,ipmi_devintf,ipmi_si

[Power check]
pattern=System Power\s+:\s(on|off)\sPower Overload\s+:\s(true|false)\sPower
Interlock\s+:\s\S+\sMain Power Fault\s+:\s(true|false)\sPower Control
Fault\s+:\s(true|false)
conditions=@string:=="on";@string:=="true";@string:=="true";@string:=="true"
```

The requires option can be used for packages, files and kernel modules using the `@pkg`, `@files` and `@modules` modifiers respectively.

The **db** type plugin

`db` plugin type queries a MySQL database and uses columns in a fashion similar to matches in a regular expression for a `file` or `command` plugin. If there is more than one result, all conditions are checked against each row.

```
# Check for empty required fields in a database.
[properties]
name=null_fields
type=db
host=localhost
user=root
password=@pass
database=alienvault
profiles=Database:!=3.1

[server_config]
query=select value from config where conf == 'server_id' or conf
== 'server_address' or conf == 'ossim_server_version'
conditions=@string:!=""
actions=@command:echo "There are null fields in your config table" | mailx -
s "Null config fields alert" root

[event_sensor]
query=select count(id) from event where sensor_id is null
conditions=@int:==0
```

This example shows another operator, `!=`, both for `profiles` and `conditions` fields. The plugin checks for null fields in `alienvault.config` and `alienvault.event` tables, triggering a mail action for the first one. Note the `@pass` keyword. This acts as a wildcard, and the Doctor will replace every ocurrence with the MySQL password. Wildcards are named using an `@` and the appropriate option from the `ossim_setup.conf` file. Therefore, a wildcard for the MySQL user account would be `@user`, and the corresponding one for the MySQL listening port would be `@db_port`.

Another interesting property of the `db` plugin is that you can invert query results by “pivoting”, using rows as columns and vice versa. Pivoting is useful when a query returns a lot of rows for a very few columns, so the previous approach won’t work. You can pivot a query by appending the option `pivot:` at the beginning of the query. For example:

```
query=pivot:select conf, value from config;
```

TODO: ‘warning’ field