# CyberTop v1.0 manual

Politecnico di Torino

## Contents

## Forewords

CyberTop (CYBERsecurity TOPologies) is the tool in charge of creating a set of countermeasures against an attack in the SHIELD framework.

This document is split into two pars: the *user manual* is useful to who wants to use the application, while the *developer manual* is useful if you want to extend it. Before delving deep into them, it is a good idea to read the introduction (Section 1), since it explains several fundamental concepts.

# 1. Introduction

The basic work-flow behind CyberTop is illustrated in Figure 1.

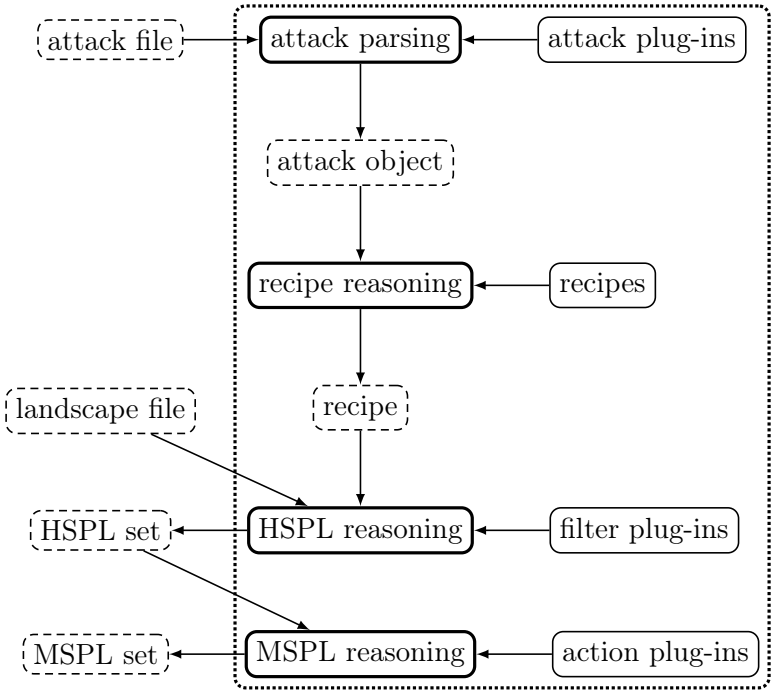


Figure 1: CyberTop architecture.

From a very high overview, CyberTop requires two input files to work with: an *attack file*, that is a CSV (Comma Separated Values) file describing the attack to mitigate, and a *landscape file*, an XML (eXtensible Markup Language) file describing the vNSF for the counterattack. The output are two sets: an HSPL (High-level Security Policy Language) and MSPL (Medium-level Security Policy Language) sets, two groups of policies at different granularity.

CyberTop can work as a background daemon, reacting to the creation of files in a specific folder by producing the HSPL and MSPL sets and saving them to a file or sending them to a server. Otherwise you can use it directly by calling its internal classes in your application.

When a new attack file arrives, CyberTop generates the remediation using the following work-flow:

1. CyberTop searches for a suitable attack parser plug-in[1] for parsing the attack file and generates an *attack object*, an internal representation that is essentially a list of *attack events*, which are the lines of the CSV file;

[1] You can write your own attack parser plug-in to allow CyberTop handling new attack types. More information is available in Section 6.

2. the attack object is analyzed by a *recipe reasoner* that picks the best recipe (countermeasure) from a set of recipes[2];

3. the recipe is feed to an *HSPL* reasoner that translates it into an HSPL set and deploys it to a vNSF using the information contained in the landscape file — CyberTop can optionally ignore some events in an attack by using a set of filter plug-ins[3];

4. the HSPL set is read by an *MSPL reasoner* that performs the final translation into an MSPL set by using a set of plug-ins[4]

[2] You can easily add your own recipe or change the existing ones to modify the behavior of CyberTop. More information is available in Section 7.

[3] You can write your own filter plug-in to customize the attack event selection. More information is available in Section 8.

[4] You can write your own action plug-in to produce your own ad-hoc MSPL set. More information is available in Section 9.

# Part I.
# User manual

## 2. Requirements

CyberTop requires Python 3 and the following dependencies:

- `setuptools`;
- `pyinotify`;
- `yapsy`;
- `lxml`;
- `python-dateutil`;
- `pika`.

The previous dependencies can also be installed automatically by launching the `setuptools` script.

## 3. Installation

To start the installation simply launch `python setup.py install`.

You can launch the built-in test suite by issuing the command `python setup.py test`. This step is optional, and if you launch it all the tests must be passed.

CyberTop can also be installed as a `systemd` service by running the script `daemon/cybertop_systemd_install.sh`.

## 4. Configuration

Before using CyberTop you need to edit a couple of configuration files in order to adapt it to your system.

First, you need to create a proper configuration file, usually named `cybertop.cfg`. An example is shown in Listing 1.

```
[global]
watchedDirectory = attacks
landscapeFile = landscape.xml
#dashboardHost = localhost
#dashboardPort = 123
#dashboardExchange = dashboard-exchange
#dashboardTopic = remediation
#dashboardAttempts = 20
#dashboardRetryDelay = 5
```

```
dashboardContent = MSPL
hsplsFile = hspls.dump
msplsFile = mspls.dump
hsplMergeInclusions = on
hsplMergeWithAnyPorts = on
hsplMergeWithSubnets = on
hsplMergingMinBits = 31
hsplMergingMaxBits = 24
hsplMergingThreshold = 10

[limit]
maxConnections = 25
rateLimit = 150kbit/s
```

Listing 1: Example of configuration file.

There are two sections: `[global]`, containing some general information for CyberTop, and `[limit]`, used to set the default parameters for the traffic rate limiting remediation.

Empty lines and lines starting with a `#` are ignored.

The `[global]` section supports the following fields:

- `watchedDirectory`: the path of the directory to watch for the creation of attack files;
- `landscapeFile`: the path of the landscape file to use;
- `dashboardHost`, `dashboardPort`, `dashboardExchange` and `dashboardTopic`: respectively the address, port, exchange name and topic of an AMQP (Advanced Message Queuing Protocol) server where the HSPL and MSPL sets will be sent — remove or comment these lines to disable the remote sending of the HSPL and MSPL sets;
- `dashboardAttempts` and `dashboardRetryDelay`: specifies how many attempts, and their temporal distance in seconds, CyberTop will perform when connecting to the AMQP server — remove or comment these lines to disable the remote sending of the HSPL and MSPL sets;
- `dashboardContent`: indicates what to send to the AMPQ server — it can be `HSPL`, `MSPL` or `HSPL+MSPL`;
- `hsplsFile` and `msplsFile`: respectively the name of two log files that will contain the generated HSPL and MSPL sets — remove or comment these lines to disable the HSPL and MSPL logging;
- `hsplMergeInclusions`: a flag (it can be `on` or `off`) that toggle the removal of the HSPLs included in other, more generic HSPLs;
- `hsplMergeWithAnyPorts`: a flag (it can be `on` or `off`) that toggle the substitution of multiple HSPLs with another one having any as a port value;
- `hsplMergeWithSubnets`: a flag (it can be `on` or `off`) that toggle the substitution of multiple HSPLs with another one having a subnet as a source/destination address value;

- `hsplMergingMinBits` and `hsplMergingMaxBits`: respectively the minimum and maximum size in bits of the generated subnets for the `hsplMergeWithSubnets` option;
- `hsplMergingThreshold`: an integer value stating the threshold that will trigger the HSPL merging (see the above options) to reduce the number of HSPLs — in short this is the maximum number of desired HSPLs, that is CyberTop will try to produce at most `hsplMergingThreshold` HSPLs.

The `[limit]` section supports the following fields:

- `maxConnections`: the default number of connections per host when a recipe does not set it;
- `rateLimit`: the default rate limit per host when a recipe does not set it — its unit of measure can be `bit/second`, `kbit/second`, `mbit/second`, `bit/minute`, `kbit/minute`, `mbit/minute`, `bit/hour`, `kbit/hour`, `mbit/hour`, `bit/day`, `kbit/day`, `mbit/day`, `bit/s`, `kbit/s`, `mbit/s`, `bit/m`, `kbit/m`, `mbit/m`, `bit/h`, `kbit/h`, `mbit/h`, `bit/d`, `kbit/d` or `mbit/d`.

Second, you need to create a landscape file to inform CyberTop of your vNSFs. The landscape file is an XML file whose schema file is located in `cybertop/xsd/landscape.xsd`. An example is shown in Listing 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<landscape xmlns="http://security.polito.it/shield/landscape"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://security.polito.it/shield/landscape␣../xsd/landscape.xsd
            ␣">

        <it-resource id="vNSF1">
                <capability>filtering.basic</capability>
        </it-resource>

        <it-resource id="vNSF2">
                <capability>filtering.basic</capability>
                <capability>filtering.limit</capability>
        </it-resource>

</landscape>
```

Listing 2: Example of landscape file.

The root tag is `landscape` and it contains a set of `it-resource` tags, specifying the *IT resources* (i.e. security controls) that CyberTop can configure. Each IT resource is uniquely identified by a name specified by the attribute `id` and supports the capabilities listed by the `capability` tag. The capabilities understood by CyberTop are:

- `filtering.basic`, indicating that an IT resource supports basic filtering facilities such as IP/port filtering and the accept/drop actions;
- `filtering.limit`, indicating that an IT resource supports the rate limit target for the traffic filtering.

Finally, you need to create a logging configuration file to tell CyberTop what to record. For more info you can look at https://docs.python.org/3/howto/logging.html.

## 5. Execution

You can launch CyberTop as a daemon launching the `daemon/daemon.py` script. It will listen when a file is created into a directory and react accordingly, sending the results to the dashboard.

It supports the following command line parameters:

- `-v` or `--version`: prints the version number and exits;
- `-c` or `--conf`: sets the configuration file to use, otherwise it will look for `/etc/cybertop.cfg`;
- `-l` or `--logging`: sets the logging configuration file to use, otherwise it will look for `logging.ini`;
- `-i` or `--interactive`: do not send CyberTop in background.

For instance, you can launch CyberTop in foreground using the command: `daemon.py -c myconfig.cfg -l mylogging.ini`.

# Part II.
# Developer manual

## 6. Adding new attack parser plug-ins

An attack parser plug-in is a special plug-in that can be used to read a CSV file detailing an attack.

Let's suppose that we want to add the support of a new attack called `myAttack`. An example CSV file is reported in Listing 3.

```
timestamp,source,destination,type
12345678,1.2.3.4,5.6.7.8,A
12345679,1.2.3.5,5.6.7.9,B
12345680,1.2.3.6,5.6.7.10,C
12345681,1.2.3.7,5.6.7.11,A
```

Listing 3: Example of an attack file.

The first row contains the column names, which are a timestamp, the attack source (the attacker), the attack destination (the attack target) and the attack type (`A`, `B` or `C`).

The name of the CSV file is also important since it stores some information. CyberTop supports two formats for the file names:

- `<severity>-<type>-<id>.csv` (e.g. `high-myAttack-123.csv`), where it specifies the attack `<severiy>` (it can be `very high`, `high`, `low` and `very low`), the attack type and a numerical id (actually ignored by CyberTop);
- `<type>.csv` (e.g. `myAttack.csv`), where the severity is by default very high.

### 6.1. Create a parser class

We want to make this file readable by CyberTop. The first thing to do is to create a Python file, for instance `ParserMyAttack.py`, in the directory `cybertop/plugins`. An example is shown in Listing 4.

```
from cybertop.plugins import ParserPlugin
from cybertop.attacks import AttackEvent
import re

class ParserMyAttack(ParserPlugin):
        def parse(self, fileName, count, line):
                if count == 1: # Skip the header line.
                        return None

                parts = re.split("\s*,\s*|\s+", line.rstrip())

                timestamp = int(parts[0])
                source = parts[1]
                destination = parts[2]
```

```
        kind = parts[3]

        attackEvent = AttackEvent(timestamp, source, destination)
        attackEvent.fields["type"] = kind

        return attackEvent
```

Listing 4: Example of an attack parser plug-in.

Your class must inherit from `cybertop.plugins.ParserPlugin` and must implement the `parse()` method. This method is called for each line in the CSV file and must return the corresponding attack event or `None` if the line should be ignored. In input it receives the file name, the current line number and the line to parse itself.

To create an attack event you must use the `AttackEvent` class. Its constructor requires a timestamp (in any format), a source and a destination (they can be IP addresses, URL and so on). The `AttackEvent.fields` attribute is a dictionary that can be used to store additional information, such as the attack type in this case.

## 6.2. Create a descriptor file

The final step is to create a descriptor file in the `cybertop/plugins` folder. This file contains some metadata about the plug-in and must have a `.yapsy-plugin` extension and must be called as the Python module, that is `ParserMyAttack.yapsy-plugin` in this case. An example is shown in Listing 5.

```
[Core]
Name = My attack event parser
Module = ParserMyAttack
Attack = myAttack
FileName = myAttack | myattack

[Documentation]
Description = My attack event parser plug-in.
Version = 0.1
Author = Somebody
```

Listing 5: Example of an attack parser descriptor.

The `[Core]` section is mandatory and must contain the following variables:

- `Name`: a human readable name of the plug-in;
- `Module`: the class name of the plug-in;
- `Attack`: the name of the attack (used by the recipes);
- `FileName`: a regular expression used to match the file names for this parser — in this case, the attack file names `myAttack.csv`, `myattack.csv`, `high-myAttack-123.csv` and `low-myattack-456.csv` matches.

The `[Documentation]` is optional and can contain the following fields:

- `Description`: a human readable description of the plug-in;
- `Version`: the version number of the plug-in;
- `Author`: the author(s) of the plug-in.

# 7. Adding new recipes

To add a new recipe, simply create a new file in the `cybertop/recipes` folder. It must be a valid XML file which *must* pass the validation against the XSD schema file `cybertop/xsd/recipe.xsd`. Invalid recipes are ignored, but logged in the log files.

An example of a recipe for an attack `myAttack` is given in Listing 6.

```
<?xml version="1.0" encoding="UTF-8"?>
<recipe-set
        minSeverity="1"
        maxSeverity="2"
        type="myAttack"
        xmlns="http://security.polito.it/shield/recipe" xmlns:xsi="http://www.w3.org/2001/
            XMLSchema-instance"
        xsi:schemaLocation="http://security.polito.it/shield/recipe ../xsd/recipe.xsd">

        <recipe>
                <name>limit packets</name>
                <action>limit</action>
                <object-constraints>
                        <any-port>true</any-port>
                </object-constraints>
                <traffic-constraints>
                        <max-connections>20</max-connections>
                        <rate-limit>100kbit/s</rate-limit>
                </traffic-constraints>
                <filters evaluation="or">
                        <my-filter>A</my-filter>
                        <my-filter>B</my-filter>
                </filters>
        </recipe>

        <recipe>
                <name>drop packets</name>
                <action>drop</action>
                <object-constraints>
                        <any-port>true</any-port>
                </object-constraints>
                <filters evaluation="or">
                        <my-filter>A</my-filter>
                        <my-filter>C</my-filter>
                </filters>
        </recipe>

</recipe-set>
```

Listing 6: Example of a recipe.

The root tag is `recipe-set`, a collection of recipes for the same attack. The `minSeverity`[5], `maxSeverity`[5] and `type` attributes are used to specify the severity range and the attack type of the recipes.

Inside a recipe set you can specify multiple recipes for the same attack, but keep in mind that only one will be chosen (most likely due to capability constraints of the landscape). You must use a `recipe` tag that can contain the following tags:

- `name`: a (mandatory) human readable description of the recipe;
- `action`: the (mandatory) action to perform on the offending traffic — the supported actions are `limit` (rate limit the traffic) and `drop` (drop the traffic);
- `object-constraints`: a set of (optional) constraints on the attack target:
  - `any-port`: a boolean value indicating that the generated HSPLs must contain an any port value in the destination;
- `traffic-constraints`: a set of (optional) constraints on the traffic:
  - `type`: a boolean value indicating that the generated HSPLs are valid only for some protocol — it can be `TCP`, `UDP` or `TCP+UDP`;
  - `max-connections`: the maximum number of connections per host if the action is `limit` — if omitted, the default value in the CyberTop configuration file will be used;
  - `rate-limit`: the rate limit per host if the action is `limit` — if omitted, the default value in the CyberTop configuration file will be used;
- `filters`: a set of (optional) filters that can be used to ignore some attack events from the remediation — the `evaluation` attribute can be `and` or `or` to specify if all the filters or at least one must match to trigger the filtering (see Section 8 for more information about the filters.)

## 8. Adding new attack filter plug-ins

Usually, when a recipe is selected, it is used to generate a list of HSPLs by replicating it for each attack event. In some cases, however, this may not be the right choice since some events might not be real attacks or must be handled with a different way. In this cases you can use a filter to ignore some events and apply a recipe only to a sub-set of the attack events in an attack.

The following filters are already available:

- `input-bytes`: restrict a recipe to the attack events with some particular size (in bytes);

[5]The values of 1, 2, 3 and 4 respectively correspond to very low, low, high and very high severity as specified in the attack file names.

- `input-packets`: restrict a recipe to the attack events with some particular size (in packets);
- `query-length`: restrict a recipe to the attack events with a specific DNS query size (in bytes);
- `query-digits`: restrict a recipe to the attack events when a DNS query has a certain amount of digits.

You use your own filter in a recipe, as shown in Listing 6. Note that a filter can receive in input a custom string as a parameter.

Thanks to the plug-in nature of CyberTop, you can easily add new filters using the following steps.

## 8.1. Create a filter class

First, we need to create a Python file, for instance `FilterMyFilter.py`, in the directory `cybertop/plugins`. An example is shown in Listing 7.

```
from cybertop.plugins import FilterPlugin

class FilterMyFilter(FilterPlugin):
        def filter(self, value, attackEvent):
                kind = value

                if kind == attackEvent.fields["type"]:
                        return True
                else:
                        return False
```

Listing 7: Example of a filter plug-in.

Your class must inherit from `cybertop.plugins.FilterPlugin` and must implement the `filter()` method. This method is called for each attack event and must return `True` if the event must be kept or `False` if it must be ignored. In input it receives the string specified in the recipe and the attack event to analyze.

## 8.2. Create a descriptor file

The second step is to create a descriptor file in the `cybertop/plugins` folder. This file contains some metadata about the plug-in and must have a `.yapsy-plugin` extension and must be called as the Python module, that is `FilterMyFilter.yapsy-plugin` in this case. An example is shown in Listing 8.

```
[Core]
Name = My filter
Module = FilterMyFilter
Tag = my-filter

[Documentation]
Description = My filter.
```

```
Version = 0.1
Author = Somebody
```

Listing 8: Example of a filter descriptor.

The `[Core]` section is mandatory and must contain the following variables:

- `Name`: a human readable name of the plug-in;
- `Module`: the class name of the plug-in;
- `Tag`: the XML tag associated to this filter, used in the recipe files.

The `[Documentation]` is optional and can contain the following fields:

- `Description`: a human readable description of the plug-in;
- `Version`: the version number of the plug-in;
- `Author`: the author(s) of the plug-in.

### 8.3. Edit the recipe schema file

The final step is to edit the recipe schema file located in `cybertop/xsd/recipe.xsd`. This is mandatory since each recipe is validated and using an unknown tag will make the validation fail. For instance, you can add the simple lines shown in Listing 9 in the type `filters` to the schema file to add the support for the `my-filter` tag.

```
<element name="my-filter" maxOccurs="1" minOccurs="0">
        <annotation>
                <documentation>My filter is pretty cool.</documentation>
        </annotation>
        <simpleType>
                <restriction base="string">
                        <pattern value="A|B|C" />
                </restriction>
        </simpleType>
</element>
```

Listing 9: Snippet for the recipe schema file.

You can also add some restrictions to make sure that the input string passed to filter is in the right format.

## 9. Adding new action plug-ins

Action plug-ins are in charge of translating an HSPL into an MSPL. They can be quite complex and, as usual, they are located in the `cybertop/plugins` directory.

Let's suppose that we want to add a `myAction` action for using it in the recipes.

## 9.1. Create an action class

As usual, the first thing to do is to create a Python file, for instance `ActionMyAction.py`, in the directory `cybertop/plugins`. An example is shown in Listing 10.

```
from cybertop.plugins import ActionPlugin
from cybertop.util import getHSPLNamespace

class ActionDrop(ActionPlugin):
        def configureITResource(self, itResource, hsplSet):
                # An FMR firewall with accept as the default action.
                c = self.createFilteringConfiguration(itResource, "accept", "FMR")

                # Creates an MSPL for each HSPL.
                count = 0
                for i in hsplSet:
                        if i.tag == "{%s}hspl" % getHSPLNamespace():
                                s = i.findtext("{%s}subject" % getHSPLNamespace())
                                o = i.findtext("{%s}object" % getHSPLNamespace()))
                                count += 1
                                # Creates a drop rule.
                                self.createFilteringRule(c, count, "drop",
                                        direction = "inbound",
                                        sourceAddress = o, sourcePort = "*",
                                        destinationAddress = s, destinationPort = "*",
                                        protocol = "TCP")
```

Listing 10: Example of an action plug-in.

Your class must inherit from `cybertop.plugins.ActionPlugin` and must implement the `configureITResource()` method. This method must configure the IT resource passed as its first parameter translating into a set of HSPLs the MSPLs passed as the second parameter.

Since a lot of MSPLs are very similar, you can use some methods that can simplify your job, instead of creating an MSPL from scratches. In particular you can used the following methods to create a filtering MSPL:

- `createFilteringConfiguration()`: initializes a filtering IT resource configuration — it as three inputs: the IT resource to initialize, the default action and the resolution strategy:
- `createFilteringRule()`: adds a new filtering rule to an initialized IT resource — it has three mandatory inputs (the IT resource configuration, the rule priority number and the MSPL action) and various optional parameters.

If you want to fully customize your MSPL set, you will need to use the `lxml.etree` class to create an MSPL XML document (see http://lxml.de/tutorial.html). Your best option for starting with this task is to look at the implementation of the aforementioned methods in the `cybertop/plugins.py` file.

Your action plug-in can translate an HSPL action into a different MSPL action such as in this case, where the HSPL action `myAction` is translated into a set of `drop` rules. If you

want to add new MSPL actions or resolution strategies you will have to edit the MSPL schema file located in `cybertop/xsd/mspl.xsd`, otherwise the XML validation will fail.

## 9.2. Create a descriptor file

Finally, you have to create a descriptor file in the `cybertop/plugins` folder. This file contains some metadata about the plug-in and must have a `.yapsy-plugin` extension and must be called as the Python module, that is `ActionMyAction.yapsy-plugin` in this case. An example is shown in Listing 11.

```
[Core]
Name = My action
Module = ActionMyAction
Capabilities = filtering.basic
Action = myAction
Score = 1

[Documentation]
Description = My action plug-in.
Version = 0.1
Author = Somebody
```

Listing 11: Example of a action descriptor.

The `[Core]` section is mandatory and must contain the following variables:

- `Name`: a human readable name of the plug-in;
- `Module`: the class name of the plug-in;
- `Capabilities`: the comma separated list of capabilities needed to implement the MSPL set produced by the translation;
- `Action`: the action name, used in the recipes;
- `Score`: a numeric value used to select this plug-in when several suitable plug-ins for the same action are present — CyberTop will pick the plug-in with the highest score.

The `[Documentation]` is optional and can contain the following fields:

- `Description`: a human readable description of the plug-in;
- `Version`: the version number of the plug-in;
- `Author`: the author(s) of the plug-in.