
Cuckoo Sandbox Book

Release 2.0.0

Cuckoo Sandbox

Apr 18, 2017

Contents

1	Using the new Cuckoo Package?	3
2	Having troubles?	5
2.1	FAQ	5
3	Contents	13
3.1	Introduction	13
3.2	Installation	17
3.3	Usage	53
3.4	Customization	96
3.5	Development	112
3.6	Final Remarks	116

Cuckoo Sandbox is an *open source* software for automating analysis of suspicious files. To do so it makes use of custom components that monitor the behavior of the malicious processes while running in an isolated environment.

This guide will explain how to set up Cuckoo, use it, and customize it.

CHAPTER 1

Using the new Cuckoo Package?

There are various big improvements related to usability in the newly released Cuckoo Package. To get the most out of it, start reading on the different subjects related to it. Following are some of the highlights:

- *Cuckoo Working Directory*
- *Cuckoo Working Directory Usage*
- *Installing Cuckoo*
- *Upgrading from a previous release*
- *Cuckoo Feedback*

CHAPTER 2

Having troubles?

If you're having troubles you might want to check out the [FAQ](#) as it may already have the answers to your questions.

FAQ

Here you can find answers for various Frequently Asked Questions:

- *General Questions*
 - *Can I analyze URLs with Cuckoo?*
 - *Can I use Volatility with Cuckoo?*
 - *What do I need to use Cuckoo with VMware ESXi?*
- *Troubleshooting*
 - *After upgrade Cuckoo stops to work*
 - *Cuckoo stumbles and produces some error I don't understand*
 - *Check and restore current snapshot with KVM*
 - *Check and restore current snapshot with VirtualBox*
 - *Unable to bind result server error*
 - *Error during template rendering*
 - *501 Unsupported Method ('GET')*
 - *Permission denied for tcpdump*
 - *DistributionNotFound / No distribution matching the version..*
 - *IOError: [Errno 24] Too many open files*

- *pkg_resources.ContextualVersionConflict*
- *Troubleshooting VM network configuration*

General Questions

Can I analyze URLs with Cuckoo?

New in version 0.5: Native support for URL analysis was added to Cuckoo.

Changed in version 2.0-rc1: Cuckoo will not only start the browser (i.e., Internet Explorer) but will also attempt to actively instrument it in order to extract interesting results such as executed Javascript, iframe URLs, etc. See also our [2.0-rc1 blogpost](#).

Additional details on URL submissions is documented at [Submit an Analysis](#), but it boils down to:

```
$ cuckoo submit --url http://www.example.com
```

Can I use Volatility with Cuckoo?

New in version 0.5: Cuckoo introduces support for optional full memory dumps, which are created at the end of the analysis process. You can use these memory dumps to perform additional memory forensic analysis with [Volatility](#).

Please also consider that we don't particularly encourage this: since Cuckoo employs some rootkit-like technologies to perform its operations, the results of a forensic analysis would be polluted by the sandbox's components.

What do I need to use Cuckoo with VMware ESXi?

To run with VMware vSphere Hypervisor (or ESXi) Cuckoo leverages on libvirt or pyVmomi (the Python SDK for the VMware vSphere API). VMware API are used to take control over virtual machines, though these APIs are available only in the licensed version. In VMware vSphere free edition these APIs are read only, so you will be unable to use it with Cuckoo. For the minimum license needed, please have a look at VMware website.

Troubleshooting

After upgrade Cuckoo stops to work

Probably you upgraded it in a wrong way. It's not a good practice to rewrite the files due to Cuckoo's complexity and quick evolution.

Please follow the upgrade steps described in [Upgrading from a previous release](#).

Cuckoo stumbles and produces some error I don't understand

Cuckoo is a mature but always evolving project, it's possible that you encounter some problems while running it, but before you rush into sending emails to everyone make sure you read what follows.

Cuckoo is not meant to be a point-and-click tool: it's designed to be a highly customizable and configurable solution for somewhat experienced users and malware analysts.

It requires you to have a decent understanding of your operating systems, Python, the concepts behind virtualization and sandboxing. We try to make it as easy to use as possible, but you have to keep in mind that it's not a technology meant to be accessible to just anyone.

That being said, if a problem occurs you have to make sure that you did everything you could before asking for time and effort from our developers and users. We just can't help everyone, we have limited time and it has to be dedicated to the development and fixing of actual bugs.

- We have extensive documentation, read it carefully. You can't just skip parts of it.
- We have a mailing list archive, search through it for previous threads where your same problem could have been already addressed and solved.
- We have a [Community](#) platform for asking questions, use it.
- We have lot of users producing content on Internet, [Google](#) it.
- Spend some of your own time trying fixing the issues before asking ours, you might even get to learn and understand Cuckoo better.

Long story short: use the existing resources, put some efforts into it and don't abuse people.

If you still can't figure out your problem, you can ask help on our online communities (see [Final Remarks](#)). Make sure when you ask for help to:

- Use a clear and explicit title for your emails: "I have a problem", "Help me" or "Cuckoo error" are **NOT** good titles.
- Explain **in details** what you're experiencing. Try to reproduce several times your issue and write down all steps to achieve that.
- Use no-paste services and link your logs, configuration files and details on your setup.
- Eventually provide a copy of the analysis that generated the problem.

Check and restore current snapshot with KVM

If something goes wrong with virtual machine it's best practice to check current snapshot status. You can do that with the following:

```
$ virsh snapshot-current "<Name of VM>"
```

If you got a long XML as output your current snapshot is configured and you can skip the rest of this chapter; anyway if you got an error like the following your current snapshot is broken:

```
$ virsh snapshot-current "<Name of VM>"
error: domain '<Name of VM>' has no current snapshot
```

To fix and create a current snapshot first list all machine's snapshots:

```
$ virsh snapshot-list "<Name of VM>"
Name                               Creation Time                       State
-----
1339506531                         2012-06-12 15:08:51 +0200         running
```

Choose one snapshot name and set it as current:

```
$ snapshot-current "<Name of VM>" --snapshotname 1339506531
Snapshot 1339506531 set as current
```

Now the virtual machine state is fixed.

Check and restore current snapshot with VirtualBox

If something goes wrong with virtual it's best practice to check the virtual machine status and the current snapshot. First of all check the virtual machine status with the following:

```
$ VBoxManage showvminfo "<Name of VM>" | grep State
State:                powered off (since 2012-06-27T22:03:57.000000000)
```

If the state is “powered off” you can go ahead with the next check, if the state is “aborted” or something else you have to restore it to “powered off” before:

```
$ VBoxManage controlvm "<Name of VM>" poweroff
```

With the following check the current snapshots state:

```
$ VBoxManage snapshot "<Name of VM>" list --details
Name: s1 (UUID: 90828a77-72f4-4a5e-b9d3-bb1fdd4cef5f)
Name: s2 (UUID: 97838e37-9ca4-4194-a041-5e9a40d6c205) *
```

If you have a snapshot marked with a star “*” your snapshot is ready, anyway you have to restore the current snapshot:

```
$ VBoxManage snapshot "<Name of VM>" restorecurrent
```

Unable to bind result server error

At Cuckoo startup if you get an error message like this one:

```
2014-01-07 18:42:12,686 [root] CRITICAL: CuckooCriticalError: Unable to bind result_
↪server on 192.168.56.1:2042: [Errno 99] Cannot assign requested address
```

It means that Cuckoo is unable to start the result server on the IP address written in cuckoo.conf (or in machinery.conf if you are using the resultserver_ip option inside). This usually happen when you start Cuckoo without bringing up the virtual interface associated with the result server IP address. You can bring it up manually, it depends from one virtualization software to another, but if you don't know how to do, a good trick is to manually start and stop an analysis virtual machine, this will bring virtual networking up.

In the case of VirtualBox the hostonly interface *vboxnet0* can be created as follows:

```
# If the hostonly interface vboxnet0 does not exist already.
$ VBoxManage hostonlyif create

# Configure vboxnet0.
$ VBoxManage hostonlyif ipconfig vboxnet0 --ip 192.168.56.1 --netmask 255.255.255.0
```

Error during template rendering

Changed in version 2.0-rc1.

In our 2.0-rc1 release a bug was introduced that looks as follows in the screenshot below. In order to resolve this issue in your local setup, please open the `web/analysis/urls.py` file and modify the 21st line by adding an underscore as follows:

```
-      "/(P<ip>[\d\.]+)?/(P<host>[a-zA-Z0-9-\.\.])?"
+      "/(P<ip>[\d\.]+)?/(P<host>[ a-zA-Z0-9-_\.\.])?"
```

The official fixes for this issue can be found in the [following commits](#).

Error during template rendering

In template /srv/cuckoo/web/templates/analysis/report.html, error at line 9

Reverse for 'analysis.views.moloch' with arguments '()' and keyword arguments '{u'host': u' _VLMCS._TCP}' not found. 1 pattern(s) tried: ['analysis/moloch/(?P<ip>[\\d\\.]+)?/(?P<host>[a-zA-Z0-9\\-\\.]+)?/(?P<src_ip>[a-zA-Z0-9\\-\\.]+)?/(?P<src_port>\\d+|None)?/(?P<dst_ip>[a-zA-Z0-9\\-\\.]+)?/(?P<dst_port>\\d+|None)?/(?P<sid>\\d+)?']

```

1 {% extends "base.html" %}
2 {% load staticfiles %}
3 {% load analysis_tags %}
4 {% block content %}
5 <div class="row">
6   <div class="col-md-6"><p style="margin-bottom: 10px;"></p></div>
7   <div class="col-md-6" style="text-align: right;">
8     <a class="btn btn-primary" href="{% url "compare.views.left" analysis.info.id %}">Compare this analysis to...</a>
9     <a class="btn btn-primary" href="{% url "submission.views.resubmit" analysis.info.id %}">Resubmit this sample</a>
10   </div>
11 </div>
12 <ul class="nav nav-tabs">
13   <li class="active"><a href="#overview" data-toggle="tab">Quick Overview</a></li>
14   <li><a href="#static" data-toggle="tab">Static Analysis</a></li>
15   {% if analysis.behavior.processes %}
16     <li><a href="#behavior" data-toggle="tab" id="graph_hook">Behavioral Analysis ({{ analysis.behavior.processes|filter_key_if_has:"track"|length }})</a></li>
17   {% endif %}
18   {% with suricata=analysis.suricata|custom_length:"alerts tls" snort=analysis.snort|custom_length:"alerts" %}
19     {% if analysis.network.http_ex or analysis.network.https_ex %}

```

501 Unsupported Method ('GET')

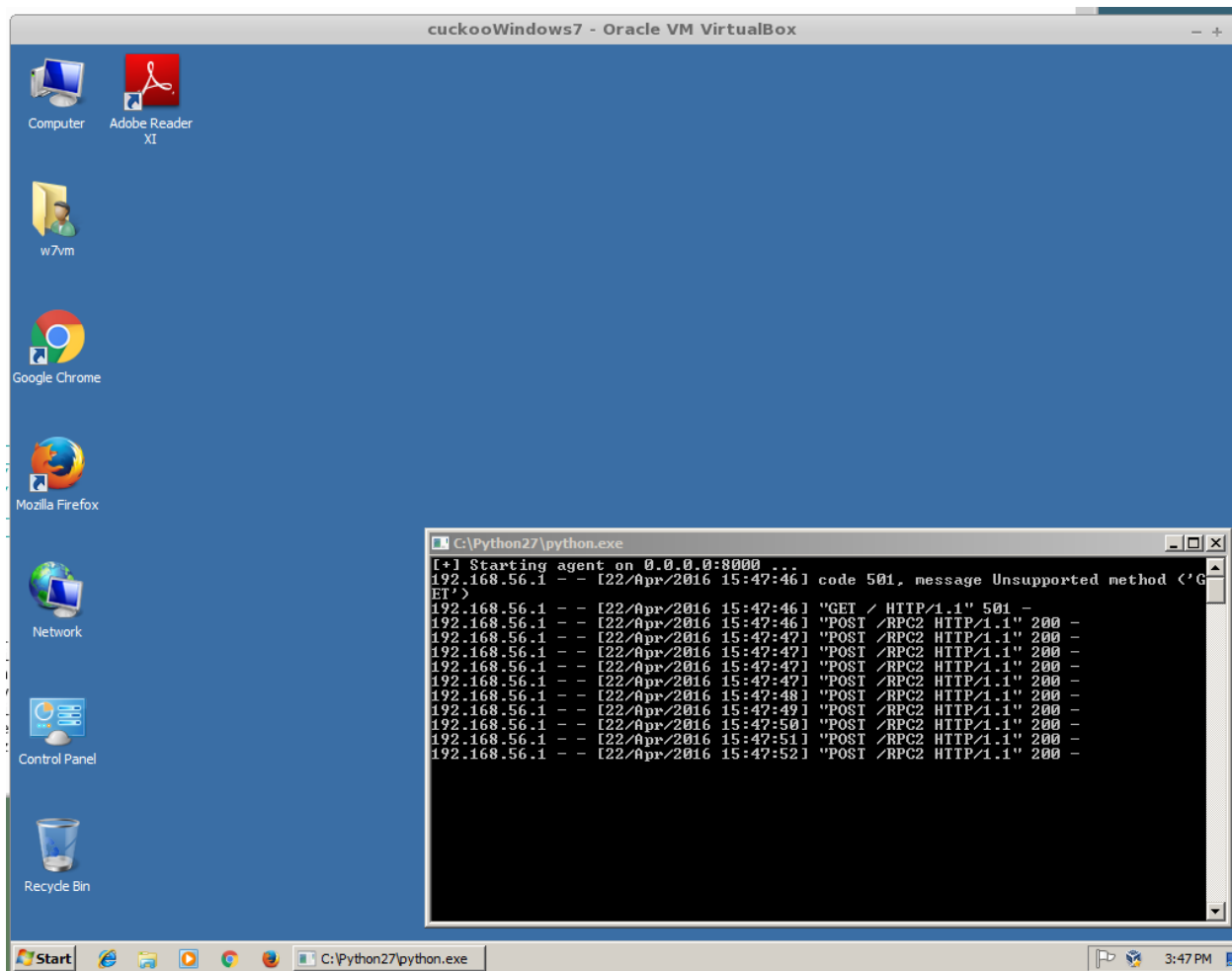
Changed in version 2.0-rc1.

Since 2.0-rc1 Cuckoo supports both the [legacy Cuckoo Agent](#) as well as a [new, REST API-based, Cuckoo Agent](#) for communication between the Guest and the Host machine. The new Cuckoo Agent is an improved Agent in the sense that it also allows usage outside of Cuckoo. As an example, it is used extensively by [VMCloak](#) in order to automatically create, configure, and cloak Virtual Machines.

Now in order to determine whether the Cuckoo Host is talking to the legacy or new Cuckoo Agent it does a HTTP GET request to the root path (/). The legacy Cuckoo Agent, which is based on `xmlrpc`, doesn't handle that specific route and therefore returns an error, 501 Unsupported method.

Having said that, the message is not actually an error, it is simply Cuckoo trying to determine to which version of the Cuckoo Agent it is talking.

Note: It should be noted that even though there is a new Cuckoo Agent available, backwards compatibility for the legacy Cuckoo Agent is still available and working properly.



Permission denied for tcpdump

Changed in version 2.0.0.

With the new Cuckoo structure in-place all storage is now, by default, located in `~/ .cuckoo`, including the PCAP file, which will be stored at `~/ .cuckoo/storage/analyses/task_id/dump.pcap`. On Ubuntu with AppArmor enabled (default configuration) `tcpdump` doesn't have write permission to dot-directories in `$HOME`, causing the permission denied message and preventing Cuckoo from capturing PCAP files.

One of the workaround is as follows - by installing AppArmor utilities and simply disabling the `tcpdump` AppArmor profile altogether (more appropriate solutions are welcome of course):

```

sudo apt-get install apparmor-utils
sudo aa-disable /usr/sbin/tcpdump

```

DistributionNotFound / No distribution matching the version..

Changed in version 2.0.0.

Installing Cuckoo through the Python package brings its own set of problems, namely that of outdated Python package management software. This FAQ entry targets the following issue..:

```
$ cuckoo
Traceback (most recent call last):
File "/usr/local/bin/cuckoo", line 5, in <module>
    from pkg_resources import load_entry_point
File "/usr/lib/python2.7/dist-packages/pkg_resources.py", line 2749, in <module>
    working_set = WorkingSet._build_master()
File "/usr/lib/python2.7/dist-packages/pkg_resources.py", line 446, in _build_master
    return cls._build_from_requirements(__requires__)
File "/usr/lib/python2.7/dist-packages/pkg_resources.py", line 459, in _build_from_
    requirements
    dists = ws.resolve(reqs, Environment())
File "/usr/lib/python2.7/dist-packages/pkg_resources.py", line 628, in resolve
    raise DistributionNotFound(req)
pkg_resources.DistributionNotFound: tlslite-ng==0.6.0a3
```

Those issues - and related ones - are caused by outdated Python package management software. Fortunately their fix is fairly trivial and therefore the following command should do the trick:

```
pip install -U pip setuptools
```

IOError: [Errno 24] Too many open files

It is most certainly possible running into this issue when analyzing samples that have a lot of dropped files, so many that the *Processing Utility* can't allocate any new file descriptors anymore.

The easiest workaround for this issue is to bump the soft and hard file descriptor limit for the current user. This may be done as documented in the [following blogpost](#).

Remember that you have to login to a new shell (i.e., usually check out first) session in order for the changes to take effect.

pkg_resources.ContextualVersionConflict

In case you're installing or upgrading the Cuckoo Package, it has happened before to people that they got an error much like the following:

```
pkg_resources.ContextualVersionConflict: (HTTPReplay 0.1.5
(/usr/local/lib/python2.7/dist-packages),
Requirement.parse('HTTPReplay==0.1.17'), set(['Cuckoo']))
```

Now this is quite odd, as generally speaking we've specifically requested pip to install all dependencies with their exact version (and in fact, if you look at `pip freeze` you'll see the correct version), but it does happen sometimes that older versions of various libraries are still around.

The easiest way to resolve this issue is by uninstalling all versions of said dependency and reinstalling Cuckoo. In the case presented above, with HTTPReplay, this may look as follows:

```
$ sudo pip uninstall httpreplay
Uninstalling HTTPReplay-0.1.17:
/usr/local/bin/httpreplay
/usr/local/bin/pcap2mitm
/usr/local/lib/python2.7/dist-packages/HTTPReplay-0.1.17-py2.7.egg-info
...
Proceed (y/n)? y
Successfully uninstalled HTTPReplay-0.1.17
```

```
$ sudo pip uninstall httpreplay
Uninstalling HTTPReplay-0.1.5:
/usr/local/lib/python2.7/dist-packages/HTTPReplay-0.1.5-py2.7.egg-info
Proceed (y/n)? y
Successfully uninstalled HTTPReplay-0.1.5

$ sudo pip uninstall httpreplay
Cannot uninstall requirement httpreplay, not installed
```

Then reinstalling Cuckoo again is simply invoking `pip install -U cuckoo` or similar.

Troubleshooting VM network configuration

In case the network configuration of your Virtual Machine isn't working as expected, you'll be prompted with the message to resolve this issue as Cuckoo isn't able to use it for analyses as-is. There are numerous possibilities as to why the network configuration and/or your setup are incorrect so please read our documentation once more. However, most often the issue lies within one of the following reasons:

- The IP address of the VM has been configured incorrectly. Please verify that the VM has a static IP address, that it matches the one in the Cuckoo configuration, and that the configured network interface exists and is up. Also, in case of VirtualBox, did you configure the network interface to be a `Host-Only` interface?
- Check that there are no firewalls in-place that hinder the communication between your Host and Guest and double check that the Host and Guest can ping each other as well as connect to each other.

If connections from the Cuckoo Host to the Guest work, but the other way around don't, then some additional problems may be at hand:

- Is the network configuration equivalent on the host and in the VM? If not, e.g., if the VM sees different IP ranges, then you'll have to configure the `resultserver_ip` and `resultserver_port`, for which we have separate documentation.
- If you've modified the Cuckoo Analyzer (located at `$CWD/analyzer`) this error message may indicate that a syntax error or other exception was introduced, preventing the Analyzer from being properly started, and thus not being able to perform the analysis as expected.

If you've triple-checked the above and are still experiencing issues, then please contact us through one of the various communication channels.

Otherwise you can ask the developers and/or other Cuckoo users, see [Join the discussion](#).

Introduction

This is an introductory chapter to Cuckoo Sandbox. It explains some basic malware analysis concepts, what's Cuckoo and how it can fit in malware analysis.

Sandboxing

As defined by [Wikipedia](#), “*in computer security, a sandbox is a security mechanism for separating running programs. It is often used to execute untested code, or untrusted programs from unverified third-parties, suppliers, untrusted users and untrusted websites.*”.

This concept applies to malware analysis' sandboxing too: our goal is to run an unknown and untrusted application or file inside an isolated environment and get information on what it does.

Malware sandboxing is a practical application of the dynamical analysis approach: instead of statically analyzing the binary file, it gets executed and monitored in real-time.

This approach obviously has pros and cons, but it's a valuable technique to obtain additional details on the malware, such as its network behavior. Therefore it's a good practice to perform both static and dynamic analysis while inspecting a malware, in order to gain a deeper understanding of it.

Simple as it is, Cuckoo is a tool that allows you to perform sandboxed malware analysis.

Using a Sandbox

Before starting to install, configure and use Cuckoo, you should take some time to think on what you want to achieve with it and how.

Some questions you should ask yourself:

- What kind of files do I want to analyze?
- What volume of analyses do I want to be able to handle?

- Which platform do I want to use to run my analysis on?
- What kind of information I want about the file?

The creation of the isolated environment (for example a virtual machine) is probably the most critical and important part of a sandbox deployment: it should be done carefully and with proper planning.

Before getting hands on the virtualization product of your choice, you should already have a design plan that defines:

- Which operating system, language and patching level to use.
- Which software to install and which versions (particularly important when analyzing exploits).

Consider that automated malware analysis is not deterministic and its success might depend on a trillion of factors: you are trying to make a malware run in a virtualized system as it would do on a native one, which could be tricky to achieve and may not always succeed. Your goal should be both to create a system able to handle all the requirements you need as well as try to make it as realistic as possible.

For example you could consider leaving some intentional traces of normal usage, such as browsing history, cookies, documents, images etc. If a malware is designed to operate, manipulate or steal such files you'll be able to notice it.

Virtualized operating systems usually carry a lot of traces with them that makes them very easily detectable. Even if you shouldn't overestimate this problem, you might want to take care of this and try to hide as many virtualization traces as possible. There is a lot of literature on Internet regarding virtualization detection techniques and countermeasures.

Once you finished designing and preparing the prototype of system you want, you can proceed creating it and deploying it. You will be always in time to change things or slightly fix them, but remember that good planning at the beginning always means less troubles in the long run.

What is Cuckoo?

Cuckoo is an open source automated malware analysis system.

It's used to automatically run and analyze files and collect comprehensive analysis results that outline what the malware does while running inside an isolated operating system.

It can retrieve the following type of results:

- Traces of calls performed by all processes spawned by the malware.
- Files being created, deleted and downloaded by the malware during its execution.
- Memory dumps of the malware processes.
- Network traffic trace in PCAP format.
- Screenshots taken during the execution of the malware.
- Full memory dumps of the machines.

Some History

Cuckoo Sandbox started as a [Google Summer of Code](#) project in 2010 within [The HoneyNet Project](#). It was originally designed and developed by *Claudio "nex" Guarnieri*, who is still the project leader and core developer.

After initial work during the summer 2010, the first beta release was published on Feb. 5th 2011, when Cuckoo was publicly announced and distributed for the first time.

In March 2011, Cuckoo has been selected again as a supported project during Google Summer of Code 2011 with The HoneyNet Project, during which *Dario Fernandes* joined the project and extended its functionality.

On November 2nd 2011 Cuckoo the release of its 0.2 version to the public as the first real stable release. On late November 2011 *Alessandro “jekil” Tanasi* joined the team expanding Cuckoo’s processing and reporting functionality.

On December 2011 Cuckoo v0.3 gets released and quickly hits release 0.3.2 in early February.

In late January 2012 we opened Malwr.com, a free and public running Cuckoo Sandbox instance provided with a full fledged interface through which people can submit files to be analysed and get results back.

In March 2012 Cuckoo Sandbox wins the first round of the [Magnificent7](#) program organized by [Rapid7](#).

During the Summer of 2012 *Jurriaan “skier” Bremer* joined the development team, refactoring the Windows analysis component sensibly improving the analysis’ quality.

On 24th July 2012, Cuckoo Sandbox 0.4 is released.

On 20th December 2012, Cuckoo Sandbox 0.5 “To The End Of The World” is released.

On 15th April 2013 we released Cuckoo Sandbox 0.6, shortly after having launched the second version of Malwr.com.

On 1st August 2013 *Claudio “nex” Guarnieri*, *Jurriaan “skier” Bremer* and *Mark “rep” Schloesser* presented [Mo’ Malware Mo’ Problems - Cuckoo Sandbox to the rescue](#) at Black Hat Las Vegas.

On 9th January 2014, Cuckoo Sandbox 1.0 is released.

In March 2014 [Cuckoo Foundation](#) born as non-profit organization dedicated to growth of Cuckoo Sandbox and the surrounding projects and initiatives.

On 7th April 2014, Cuckoo Sandbox 1.1 is released.

On the 7th of October 2014, Cuckoo Sandbox 1.1.1 is released after a [Critical Vulnerability](#) had been disclosed by Robert Michel.

On the 4th of March 2015, Cuckoo Sandbox 1.2 has been released featuring a wide array of improvements regarding the usability of Cuckoo.

During summer 2015 Cuckoo Sandbox started the development of Mac OS X malware analysis as a [Google Summer of Code](#) project within [The Honeynet Project](#). *Dmitry Rodionov* qualified for the project and developed a working analyzer for Mac OS X.

On the 21st of February 2016 [version 2.0 Release Candidate 1](#) is released. This version ships with almost two years of combined effort into making Cuckoo Sandbox a better project for daily usage.

Use Cases

Cuckoo is designed to be used both as a standalone application as well as to be integrated in larger frameworks, thanks to its extremely modular design.

It can be used to analyze:

- Generic Windows executables
- DLL files
- PDF documents
- Microsoft Office documents
- URLs and HTML files
- PHP scripts
- CPL files
- Visual Basic (VB) scripts
- ZIP files

- Java JAR
- Python files
- *Almost anything else*

Thanks to its modularity and powerful scripting capabilities, there's no limit to what you can achieve with Cuckoo.

For more information on customizing Cuckoo, see the [Customization](#) chapter.

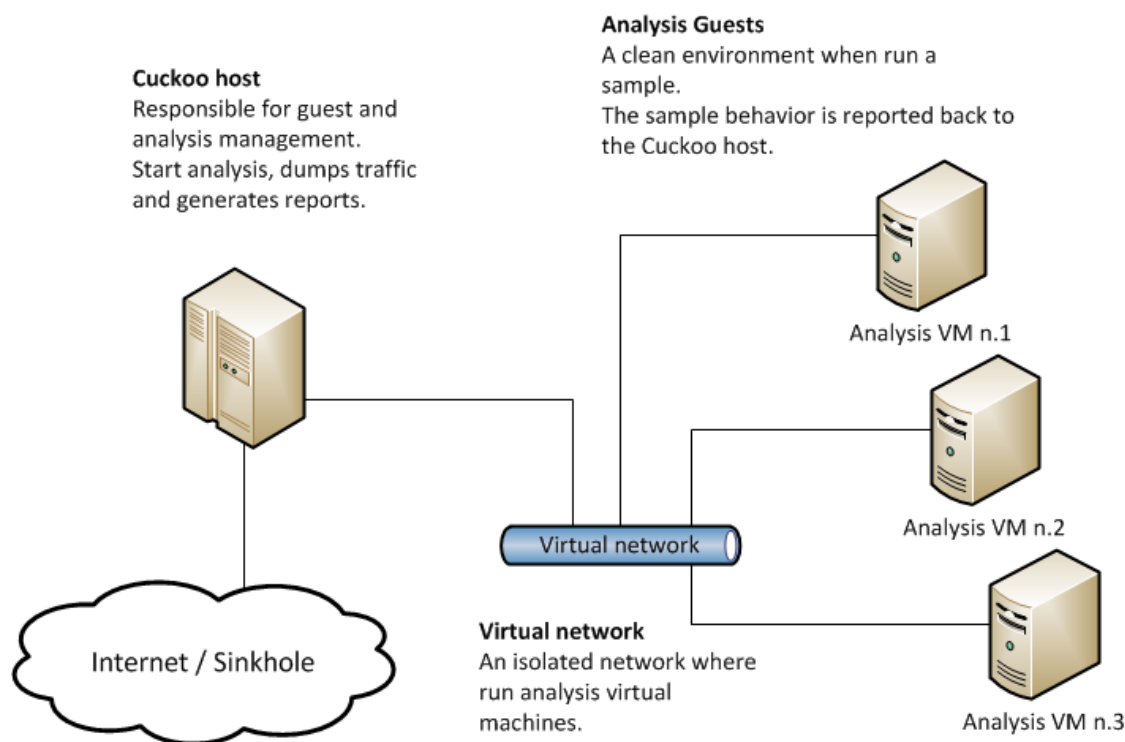
Architecture

Cuckoo Sandbox consists of a central management software which handles sample execution and analysis.

Each analysis is launched in a fresh and isolated virtual or physical machine. The main components of Cuckoo's infrastructure are an Host machine (the management software) and a number of Guest machines (virtual or physical machines for analysis).

The Host runs the core component of the sandbox that manages the whole analysis process, while the Guests are the isolated environments where the malware samples get actually safely executed and analyzed.

The following picture explains Cuckoo's main architecture:



Obtaining Cuckoo

Deprecated since version 2.0-rc2: Although Cuckoo can still be downloaded from the website we discourage from doing so, given that simply installing it through pip is the preferred way to get Cuckoo. Please refer to [Installing Cuckoo](#).

Cuckoo can be downloaded from the [official website](#), where the stable and packaged releases are distributed, or can be cloned from our [official git repository](#).

Warning: While being more updated, including new features and bugfixes, the version available in the git repository should be considered an *under development* stage. Therefore its stability is not guaranteed and it most likely lacks updated documentation.

License

Cuckoo Sandbox license is shipped with Cuckoo and contained in the “LICENSE” file inside the “docs” folder.

Disclaimer

Cuckoo is distributed as it is, in the hope that it will be useful, but without any warranty neither the implied merchantability or fitness for a particular purpose.

Whatever you do with this tool is uniquely your own responsibility.

Cuckoo Foundation

The [Cuckoo Foundation](#) is a non-profit organization incorporated as a Stichting in the Netherlands and it's mainly dedicated to support of the development and growth of Cuckoo Sandbox, an open source malware analysis system, and the surrounding projects and initiatives.

The Foundation operates to secure financial and infrastructure support to our software projects and coordinates the development and contributions from the community.

Installation

This chapter explains how to install Cuckoo.

Although the recommended setup is *GNU/Linux* (Debian or Ubuntu preferably), Cuckoo has proved to work smoothly on *Mac OS X* and *Microsoft Windows 7* as host as well. The recommended and tested setup for guests are *Windows XP* and *64-bit Windows 7* for Windows analysis, *Mac OS X Yosemite* for Mac OS X analysis, and Debian for Linux Analysis, although Cuckoo should work with other releases of guest Operating Systems as well.

Note: This documentation refers to *Host* as the underlying operating systems on which you are running Cuckoo (generally being a GNU/Linux distribution) and to *Guest* as the Windows virtual machine used to run the isolated analysis.

Preparing the Host

To run Cuckoo we suggest a *GNU/Linux* operating system. We'll be using the **latest Ubuntu LTS** (16.04 at the time of writing) throughout our documentation.

Requirements

Before proceeding to installing and configuring Cuckoo, you'll need to install some required software packages and libraries.

Installing Python libraries (on Ubuntu/Debian-based distributions)

The Cuckoo host components is completely written in Python, therefore it is required to have an appropriate version of Python installed. At this point we only fully support **Python 2.7**. Older version of Python and Python 3 versions are not supported by us (although Python 3 support is on our TODO list with a low priority).

The following software packages from the apt repositories are required to get Cuckoo to install and run properly:

```
$ sudo apt-get install python python-pip python-dev libffi-dev libssl-dev
$ sudo apt-get install python-virtualenv python-setuptools
$ sudo apt-get install libjpeg-dev zlib1g-dev swig
```

In order to use the Django-based Web Interface, MongoDB is required:

```
$ sudo apt-get install mongodb
```

In order to use PostgreSQL as database (our recommendation), PostgreSQL will have to be installed as well:

```
$ sudo apt-get install postgresql libpq-dev
```

Yara and **Pydeep** are *optional* plugins but will have to be installed manually, so please refer to their websites.

If you want to use KVM as machinery module you will have to install KVM:

```
$ sudo apt-get install qemu-kvm libvirt-bin ubuntu-vm-builder bridge-utils python-
↳ libvirt
```

If you want to use XenServer you'll have to install the *XenAPI* Python package:

```
$ sudo pip install XenAPI
```

If you want to use the *mitm* auxiliary module (to intercept SSL/TLS generated traffic), you need to install **mitmproxy**. Please refer to its website for installation instructions.

Installing Python libraries (on Mac OS X)

This is mostly the same as the installation on Ubuntu/Debian, except that we'll be using the **brew** package manager. Install all the required dependencies as follows (this list is WIP):

```
$ brew install libmagic
```

Installing Python libraries (on Windows 7)

To be documented.

Virtualization Software

Cuckoo Sandbox supports most Virtualization Software solutions. As you will see throughout the documentation, Cuckoo has been setup to remain as modular as possible and in case integration with a piece of software is missing this could be easily added.

For the sake of this guide we will assume that you have VirtualBox installed (which is the default), but this does **not** affect the execution and general configuration of the sandbox.

You are completely responsible for the choice, configuration, and execution of your virtualization software. Please read our extensive documentation and FAQ before reaching out to us with questions on how to set Cuckoo up.

Assuming you decide to go for VirtualBox, you can get the proper package for your distribution at the [official download page](#). Please find following the commands to install the latest version of VirtualBox on your Ubuntu LTS machine. Note that Cuckoo supports VirtualBox 4.3, 5.0, and 5.1:

```
$ echo deb http://download.virtualbox.org/virtualbox/debian xenial contrib | sudo tee -a /etc/apt/sources.list.d/virtualbox.list
$ wget -q https://www.virtualbox.org/download/oracle_vbox_2016.asc -O- | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install virtualbox-5.1
```

For more information on VirtualBox, please refer to the [official documentation](#).

Installing tcpdump

In order to dump the network activity performed by the malware during execution, you'll need a network sniffer properly configured to capture the traffic and dump it to a file.

By default Cuckoo adopts [tcpdump](#), the prominent open source solution.

Install it on Ubuntu:

```
$ sudo apt-get install tcpdump apparmor-utils
$ sudo aa-disable /usr/sbin/tcpdump
```

Note that the AppArmor profile disabling (the `aa-disable` command) is only required when using the default CWD directory as AppArmor would otherwise prevent the creation of the actual PCAP files (see also [Permission denied for tcpdump](#)).

For Linux platforms with AppArmor disabled (e.g., Debian) the following command will suffice to install [tcpdump](#):

```
$ sudo apt-get install tcpdump
```

Tcpdump requires root privileges, but since you don't want Cuckoo to run as root you'll have to set specific Linux capabilities to the binary:

```
$ sudo setcap cap_net_raw,cap_net_admin=eip /usr/sbin/tcpdump
```

You can verify the results of the last command with:

```
$ getcap /usr/sbin/tcpdump
/usr/sbin/tcpdump = cap_net_admin,cap_net_raw=eip
```

If you don't have `setcap` installed you can get it with:

```
$ sudo apt-get install libcap2-bin
```

Or otherwise (**not recommended**) do:

```
$ sudo chmod +s /usr/sbin/tcpdump
```

Please keep in mind that even the `setcap` method is not perfectly safe (due to potential security vulnerabilities) if the system has other users which are potentially untrusted. We recommend to run Cuckoo on a dedicated system or a trusted environment where the privileged tcpdump execution is contained otherwise.

Installing Volatility

Volatility is an optional tool to do forensic analysis on memory dumps. In combination with Cuckoo, it can automatically provide additional visibility into deep modifications in the operating system as well as detect the presence of rootkit technology that escaped the monitoring domain of Cuckoo's analyzer.

In order to function properly, Cuckoo requires at least version 2.3 of Volatility, but recommends the latest version, Volatility 2.5. You can download it from their [official repository](#).

See the volatility documentation for detailed instructions on how to install it.

Installing M2Crypto

Currently the M2Crypto library is only supported when SWIG has been installed. On Ubuntu/Debian-like systems this may be done as follows:

```
$ sudo apt-get install swig
```

If SWIG is present on the system, Cuckoo will automatically install the M2Crypto dependency upon installation.

Installing Cuckoo

Create a user

You can either run Cuckoo from your own user or create a new one dedicated just for your sandbox setup. Make sure that the user that runs Cuckoo is the same user that you will use to create and run the virtual machines (at least in the case of VirtualBox), otherwise Cuckoo won't be able to identify and launch these Virtual Machines.

Create a new user:

```
$ sudo adduser cuckoo
```

If you're using VirtualBox, make sure the new user belongs to the "vboxusers" group (or the group you used to run VirtualBox):

```
$ sudo usermod -a -G vboxusers cuckoo
```

If you're using KVM or any other libvirt based module, make sure the new user belongs to the "libvirtd" group (or the group your Linux distribution uses to run libvirt):

```
$ sudo usermod -a -G libvirtd cuckoo
```

Raising file limits

As outlined in the [FAQ](#) entry *IOError: [Errno 24] Too many open files* one may want to bump the file count limits before starting Cuckoo as otherwise some samples will fail to properly process the report (due to opening more files than allowed by the Operating System).

Install Cuckoo

Installing the latest version of Cuckoo is as simple as follows. Note that it is recommended to first upgrade the `pip` and `setuptools` libraries as they're often outdated, leading to issues when trying to install Cuckoo (see also *DistributionNotFound / No distribution matching the version..*).

Warning: It is not unlikely that you'll be missing one or more system packages required to build various Python dependencies. Please read and re-read *Requirements* to resolve these sorts of issues.

```
$ sudo pip install -U pip setuptools
$ sudo pip install -U cuckoo
```

Although the above, a *global* installation of Cuckoo in your OS works mostly fine, we **highly recommend** installing Cuckoo in a `virtualenv`, which looks roughly as follows:

```
$ virtualenv venv
$ . venv/bin/activate
(venv)$ pip install -U pip setuptools
(venv)$ pip install -U cuckoo
```

Some reasons for using a `virtualenv`:

- Cuckoo's dependencies may not be entirely up-to-date, but instead pin to a known-to-work-properly version.
- The dependencies of other software installed on your system may conflict with those required by Cuckoo, due to incompatible version requirements (and yes, this is also possible when Cuckoo supports the latest version, simply because the other software may have pinned to an older version).
- Using a `virtualenv` allows non-root users to install additional packages or upgrade Cuckoo at a later point in time.
- And simply put, `virtualenv` is considered a best practice.

Please refer to *Cuckoo Working Directory* and *Cuckoo Working Directory Usage* to learn more about the Cuckoo Working Directory and how to operate it.

Install Cuckoo from file

By downloading a hard copy of the Cuckoo Package and installing it *offline*, one may set up Cuckoo using a cached copy and/or have a backup copy of current Cuckoo versions in the future. We also feature the option to download such a tarball on our website.

Obtaining the tarball of Cuckoo and all of its dependencies manually may be done as follows:

```
$ pip download cuckoo
```

You will end up with a file `Cuckoo-2.0.0.tar.gz` (or a higher number, depending on the latest released stable version) as well as all of its dependencies (e.g., `alembic-0.8.8.tar.gz`).

Installing that exact version of Cuckoo may be done as you're familiar with from installing it using `pip` directly, except now using the filename of the tarball:

```
$ pip install Cuckoo-2.0.0.tar.gz
```

On systems where no internet connection is available, the `$ pip download cuckoo` command may be used to fetch all of the required dependencies and as such one should be able to - in theory - install Cuckoo completely offline using those files, i.e., by executing something like the following:

```
$ pip install *.tar.gz
```

Per-Analysis Network Routing

Since Cuckoo 2.0-rc1 it is possible to feature per-analysis network routing. In other words, if you have one VM and three samples to analyze, it is possible to deny internet access for the first analysis, route the second analysis through a VPN, and pull the third analysis through the Tor network.

However, aside from the more advanced per-analysis routing, it is naturally also possible to have one default route - a setup that used to be popular before, when the more luxurious routing was not yet available.

In our examples we'll be focusing on VirtualBox as it is our default machinery choice.

Simple Global Routing

Before delving into the more complex and feature-rich per-analysis network routing we'll first cover the older approach, which is based on global `iptables` rules that are, once set, not changed anymore.

In the following setup we're assuming that the interface assigned to our VirtualBox VM is `vboxnet0`, the IP address of our VM is `192.168.56.101` (in a `/24` subnet), and that the outgoing interface connected to the internet is `eth0`. With such a setup, the following `iptables` rules will allow the VMs access to the Cuckoo host machine (`192.168.56.1` in this setup) as well as the entire internet as you would expect from any application connecting to the internet.

```
$ sudo iptables -t nat -A POSTROUTING -o eth0 -s 192.168.56.0/24 -j MASQUERADE

# Default drop.
$ sudo iptables -P FORWARD DROP

# Existing connections.
$ sudo iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT

# Accept connections from vboxnet to the whole internet.
$ sudo iptables -A FORWARD -s 192.168.56.0/24 -j ACCEPT

# Internal traffic.
$ sudo iptables -A FORWARD -s 192.168.56.0/24 -d 192.168.56.0/24 -j ACCEPT

# Log stuff that reaches this point (could be noisy).
$ sudo iptables -A FORWARD -j LOG
```

And that's pretty much it, with these rules set we're almost good to go. However, these rules won't be doing any packet forwarding unless IP forwarding is explicitly enabled in the kernel. To do so, there is a temporary method that survives until a shutdown or reboot, and a permanent method that is taken into account when booting the machine. Simply put, generally speaking you'll want to run both commands:

```
$ echo 1 | sudo tee -a /proc/sys/net/ipv4/ip_forward
$ sudo sysctl -w net.ipv4.ip_forward=1
```

`Iptables` rules are not persistent between reboots, so if want to keep them you should use a script or just install `iptables-persistent`.

Per-Analysis Network Routing Options

Having discussed the old school method for routing analyses through a network interface we will now walk through the dynamic network routing components that allow for much more granular network routing.

As outlined in the introduction for this chapter of the documentation it has been possible since Cuckoo 2.0-rc1, when we introduced the *Cuckoo Rooter*, to do per-analysis network routing. Since then various bugs have been resolved and more network routing options have been added.

Following is the list of available routing options.

Routing Option	Description
<i>None Routing</i>	No routing whatsoever, the only option that does <i>not</i> require the Cuckoo Rooter to be run (and therefore also the default routing option).
<i>Drop Routing</i>	Completely drops all non-Cuckoo traffic, including traffic within the VMs' subnet.
<i>Internet Routing</i>	Full internet access as provided by the given network interface (similar to the <i>Simple Global Routing</i> setup).
<i>InetSim Routing</i>	Routes all traffic to an InetSim instance - which provides fake services - running on the host machine.
<i>Tor Routing</i>	Routes all traffic through Tor.
<i>VPN Routing</i>	Routes all traffic through one of perhaps multiple pre-defined VPN endpoints.

Using Per-Analysis Network Routing

Having knowledge about the available network routing options it is time to actually use it in practice. Assuming Cuckoo has been configured properly taking advantage of its features is really as simple as **starting the Cuckoo Rooter and choosing a network routing option for your analysis**.

Documentation on starting the Cuckoo Rooter may be found in the *Cuckoo Rooter Usage* document.

Configuring iproute2

For Linux kernel TCP/IP source routing reasons it is required to register each of the network interfaces that we use with `iproute2`. This is trivial, but necessary.

As an example we'll be configuring *Internet Routing* (aka the `dirty line`) for which we'll be using the `eth0` network interface - reverting back to Ubuntu 14.04 and older terminology here for a second (Ubuntu 16.04 uses network interface names based on the hardware manufacturer, as you will likely have seen happen on BSD-based systems since forever).

To configure `iproute2` with `eth0` we're going to open the `/etc/iproute2/rt_tables` file which will look roughly as follows:

```
#
# reserved values
#
255    local
254    main
253    default
0      unspec
#
```

```
# local
#
```

Now roll a random number that is not yet present in this file with your dice of choice and use it to craft a new line at the end of the file. As an example, registering `eth0` with `iproute2` could look as follows:

```
#
# reserved values
#
255      local
254      main
253      default
0        unspec
#
# local
#
400      eth0
```

And that's really all there is to it. You will have to do this for each network interface you intend to use for network routing.

None Routing

The default routing mechanism in the sense that Cuckoo allows the analysis to route as defined by a third party. As in, it literally doesn't do anything. One may use the `none` routing in conjunction with the [Simple Global Routing](#).

Drop Routing

The `drop` routing option is somewhat like a default [None Routing](#) setup (as in, in a machine where no global `iptables` rules have been created providing full internet access to VMs or so), except that it is much more aggressive in actively locking down the internet access provided to the VM.

With `drop` routing the only traffic possible is internal Cuckoo traffic and hence any DNS requests or outgoing TCP/IP connections are blocked.

Internet Routing

By using the `internet` routing one may provide full internet access to VMs through one of the connected network interfaces. We also refer to this option as the `dirty line` due to its nature of allowing all potentially malicious samples to connect to the internet through the same uplink.

Note: It is required to register the dirty line network interface with `iproute2` as described in the [Configuring iproute2](#) section.

InetSim Routing

For those that have not heard of [InetSim](#), it's a project that provides fake services for malware to talk to. In order to use `InetSim` routing one will have to setup `InetSim` on the host machine (or in a separate VM) and configure Cuckoo so that it knows where to find the `InetSim` server.

The configuration for InetSim is self-explanatory and can be found as part of the `$CWD/conf/routing.conf` configuration file:

```
[inetsim]
enabled = yes
server = 192.168.56.1
```

In order to quickly get started with InetSim it is possible to download the latest version of the [REMnux](#) distribution which features - among many other tools - the latest version of InetSim. Naturally this VM will require its own static IP address which should then be configured in the `routing.conf` configuration file.

Tor Routing

Note: Although we **highly discourage** the use of Tor for malware analysis - the maintainers of Tor `exit nodes` already have a hard enough time keeping up their servers - it is in fact a well-supported feature.

First of all Tor will have to be installed. Please find instructions on installing the [latest stable version of Tor here](#).

We'll then have to modify the Tor configuration file (not talking about Cuckoo's configuration for Tor yet!) In order to do so, we will have to provide Tor with the listening address and port for TCP/IP connections and UDP requests. For a default VirtualBox setup, where the host machine has IP address `192.168.56.1`, the following lines will have to be configured in the `/etc/tor/torrc` file:

```
TransPort 192.168.56.1:9040
DNSPort 192.168.56.1:5353
```

Don't forget to restart Tor (`/etc/init.d/tor restart`). That leaves us with the Tor configuration for Cuckoo, which may be found in the `$CWD/conf/routing.conf` file. The configuration is pretty self-explanatory so we'll leave filling it out as an exercise to the reader (in fact, toggling the `enabled` field goes a long way):

```
[tor]
enabled = yes
dnspoint = 5353
proxyport = 9040
```

Note that the port numbers in the `/etc/tor/torrc` and `$CWD/conf/routing.conf` files must match in order for the two to interact correctly.

VPN Routing

Last but not least, it is possible to route analyses through a number of VPNs. By defining a couple of VPNs, perhaps ending up in different countries, it may be possible to see if potentially malicious samples behave differently depending on the country of origin of its IP address.

The configuration for a VPN is much like the configuration of a VM. For each VPN you will need one section in the `$CWD/conf/routing.conf` configuration file detailing the relevant information for the VPN. In the configuration the VPN will also have to be *registered* in the list of available VPNs (exactly the same as you'd do for registering more VMs).

Configuration for a single VPN looks roughly as follows:

```
[vpn]
# Are VPNs enabled?
```

```
enabled = yes

# Comma-separated list of the available VPNs.
vpns = vpn0

[vpn0]
# Name of this VPN. The name is represented by the filepath to the
# configuration file, e.g., cuckoo would represent /etc/openvpn/cuckoo.conf
# Note that you can't assign the names "none" and "internet" as those would
# conflict with the routing section in cuckoo.conf.
name = vpn0

# The description of this VPN which will be displayed in the web interface.
# Can be used to for example describe the country where this VPN ends up.
description = Spain, Europe

# The tun device hardcoded for this VPN. Each VPN must be configured to use
# a hardcoded/persistent tun device by explicitly adding the line "dev tunX"
# to its configuration (e.g., /etc/openvpn/vpn1.conf) where X in tunX is a
# unique number between 0 and your lucky number of choice.
interface = tun0

# Routing table name/id for this VPN. If table name is used it must be
# added to /etc/iproute2/rt_tables as "<id> <name>" line (e.g., "201 tun0").
# ID and name must be unique across the system (refer /etc/iproute2/rt_tables
# for existing names and IDs).
rt_table = tun0
```

Note: It is required to register each VPN network interface with iproute2 as described in the *Configuring iproute2* section.

Cuckoo Working Directory

New in version 2.0.0.

A new concept is the Cuckoo Working Directory. From this point forward all configurable components, generated data, and results of Cuckoo will be stored in this directory. These files include but are not limited to the following:

- Configuration
- Cuckoo Signatures
- Cuckoo Analyzer
- Cuckoo Agent
- Yara rules
- Cuckoo Storage (where analysis results go)
- And much more..

The Cuckoo Working Directory comes with a couple of advantages over the legacy approach that Cuckoo used. Following we will study how the Cuckoo Working Directory (CWD from now on) overcomes various every-day hurdles.

requires an updated Configuration then Cuckoo will guide the user through it - instead of overwriting the Configuration files itself).

CWD path

Even though the CWD defaults to `~/ .cuckoo` this path is completely configurable. The following lists the order of precedence for Cuckoo to determine the CWD.

- Through the `--cwd` command-line option (e.g., `--cwd ~/ .cuckoo`).
- Through the `CUCKOO` environment variable (e.g., `export CUCKOO=~/ .cuckoo`).
- Through the `CUCKOO_CWD` environment variable.
- If the current directory is a CWD (e.g., `cd ~/ .cuckoo` assuming that a CWD has been created in that directory).
- The default, `~/ .cuckoo`.

By using alternative CWD paths it is **possible to run multiple Cuckoo instances with different configurations using the same Cuckoo setup**. If for some reason one requires two or three separate Cuckoo setups, e.g., in the case that you want to run Windows analysis and Android analysis in parallel, then not having to upgrade each instance one-by-one every time there is an update surely is a great step forward.

Following some examples to show how to configure the CWD.

```
# Places the CWD in /opt/cuckoo. Note that Cuckoo will normally create the
# CWD itself, but in order to create a directory in /opt root capabilities
# are usually required.
$ sudo mkdir /opt/cuckoo
$ sudo chown cuckoo:cuckoo /opt/cuckoo
$ cuckoo --cwd /opt/cuckoo

# You could place this line in your .bashrc, for example.
$ export CUCKOO=/opt/cuckoo
$ cuckoo
```

Experimenting with multiple Cuckoo setups is now as simple as creating multiple CWD's and configuring them accordingly.

Configuration

Cuckoo relies on a couple of main configuration files:

- *cuckoo.conf*: for configuring general behavior and analysis options.
- *auxiliary.conf*: for enabling and configuring auxiliary modules.
- *<machinery>.conf*: for defining the options for your virtualization software (the file has the same name of the machinery module you choose in *cuckoo.conf*).
- *memory.conf*: Volatility configuration.
- *processing.conf*: for enabling and configuring processing modules.
- *reporting.conf*: for enabling or disabling report formats.

To get Cuckoo working you should at the very least edit *cuckoo.conf* and *<machinery>.conf*.

cuckoo.conf

The first file to edit is `$CWD/conf/cuckoo.conf`. Note that we'll be referring to the *Cuckoo Working Directory* when we talk about `$CWD`. The `cuckoo.conf` file contains generic configuration options that you will want to verify or at least familiarize yourself with before launching Cuckoo.

The file is largely commented and self-explanatory, but some of the options may be of special interest to you:

- **machinery in [cuckoo]:** This option defines which Machinery module you want Cuckoo to use to interact with your analysis machines. The value must be the name of the module without extension (e.g., `virtualbox` or `vmware`).
- **ip and port in [resultserver]:** These define the local IP address and port that Cuckoo is going to try to bind the result server on. Make sure this matches the network configuration of your analysis machines or they won't be able to return any results.
- **connection in [database]:** The database connection string defines how Cuckoo will connect to the internal database. You can use any DBMS supported by [SQLAlchemy](#) using a valid [Database Urls](#) syntax.

Warning: Check your interface for resultserver IP! Some virtualization software (for example Virtualbox) don't bring up the virtual networking interfaces until a virtual machine is started. Cuckoo needs to have the interface where you bind the resultserver up before the start, so please check your network setup. If you are not sure about how to get the interface up, a good trick is to manually start and stop an analysis virtual machine, this will bring virtual networking up. If you are using NAT/PAT in your network, you can set up the resultserver IP to 0.0.0.0 to listen on all interfaces, then use the specific options `resultserver_ip` and `resultserver_port` in `<machinery>.conf` to specify the address and port as every machine sees them. Note that if you set resultserver IP to 0.0.0.0 in `cuckoo.conf` you have to set `resultserver_ip` for all your virtual machines.

auxiliary.conf

Auxiliary modules are scripts that run concurrently with malware analysis, this file defines their options.

Following is the default `$CWD/conf/auxiliary.conf` file.

<machinery>.conf

Machinery modules are scripts that define how Cuckoo should interact with your virtualization software of choice.

Every module has a dedicated configuration file which defines the details on the available machines. For example, Cuckoo comes with a `VMware` machinery module. In order to use it one has to specify `vmware` as `machinery` option in `$CWD/conf/cuckoo.conf` and populate the `$CWD/conf/vmware.conf` file with the available Virtual Machines.

Cuckoo provides some modules by default and for the sake of this guide, we'll assume you're going to use VirtualBox.

Following is the default `$CWD/conf/virtualbox.conf` file.

The configuration for the other machinery modules look mostly the same with some variations where required. E.g., `XenServer` operates through an API, so to access it a URL and credentials are required.

The comments for the options are self-explanatory.

Following is the default `$CWD/conf/kvm.conf` file.

memory.conf

The Volatility tool offers a large set of plugins for memory dump analysis. Some of them are quite slow. The `$CWD/conf/volatility.conf` file let's you enable or disable plugins of your choice. To use Volatility you have to follow two steps:

- Enable volatility in `$CWD/conf/processing.conf`
- Enable memory_dump in `$CWD/conf/cuckoo.conf`

In `$CWD/conf/memory.conf`'s basic section you can configure the Volatility profile and whether memory dumps should be deleted after having been processed (this saves a lot of disk space):

```
# Basic settings
[basic]
# Profile to avoid wasting time identifying it
guest_profile = WinXPSP2x86
# Delete memory dump after volatility processing.
delete_memdump = no
```

After that every plugin has its own section for configuration:

```
# Scans for hidden/injected code and dlls
# http://code.google.com/p/volatility/wiki/CommandReference#malfind
[malfind]
enabled = on
filter = on

# Lists hooked api in user mode and kernel space
# Expect it to be very slow when enabled
# http://code.google.com/p/volatility/wiki/CommandReference#apihooks
[apihooks]
enabled = off
filter = on
```

The filter configuration helps you to remove known clean data from the resulting report. It can be configured separately for every plugin.

The filter itself is configured in the `[mask]` section. You can enter a list of pids in `pid_generic` to filter out processes:

```
# Masks. Data that should not be logged
# Just get this information from your plain VM Snapshot (without running malware)
# This will filter out unwanted information in the logs
[mask]
# pid_generic: a list of process ids that already existed on the machine before the_
↪malware was started.
pid_generic = 4, 680, 752, 776, 828, 840, 1000, 1052, 1168, 1364, 1428, 1476, 1808, ↪
↪452, 580, 652, 248, 1992, 1696, 1260, 1656, 1156
```

processing.conf

This file allows you to enable, disable and configure all processing modules. These modules are located under the `cuckoo.processing` module and define how to digest the raw data collected during the analysis.

You will find a section for each processing module in `$CWD/conf/processing.conf`.

You might want to configure the [VirusTotal](#) key if you have an account of your own.

reporting.conf

The `$CWD/conf/reporting.conf` file contains information on the reports generation.

It contains the following sections.

By setting those option to `on` or `off` you enable or disable the generation of such reports.

Monitoring Cuckoo with Icinga2

The following instructions assume that you have both your Cuckoo instance(s) as well as the Icinga2 instance (which, preferably, runs on a separate server) running on a Debian/Ubuntu-based distribution.

Note that all commands mentioned in this document should be ran as **root** and that any highlighted lines feature some sort of user-specific configuration.

Installing the Icinga2 master

In this chapter we'll install the `master`, also known as the node in which the results from the various clients / satellites may be monitored.

First add the apt key for Icinga2 on Ubuntu:

```
$ wget -O - http://packages.icinga.org/icinga.key | apt-key add -
$ echo 'deb http://packages.icinga.org/ubuntu icinga-trusty main' > /etc/apt/sources.
  ↳list.d/icinga.list
$ apt-get update
```

Or on Debian:

```
$ wget -O - http://packages.icinga.org/icinga.key | apt-key add -
$ echo 'deb http://packages.icinga.org/debian icinga-jessie main' > /etc/apt/sources.
  ↳list.d/icinga.list
$ apt-get update
```

Then we install the actual packages:

```
$ apt-get install icinga2 php5-json php5-gd php5-imagick php5-mysql
$ apt-get install php5-pgsql php5-intl php5-cli php5-common php5-fpm
$ echo 'date.timezone = "Europe/Amsterdam"' >> /etc/php5/fpm/php.ini
$ icinga2 feature enable command
$ service icinga2 restart
```

Setup the PostgreSQL database:

```
$ sudo -u postgres psql
postgres=# CREATE USER icingaweb WITH PASSWORD 'YOURDATABASEPASSWORD';
postgres=# CREATE DATABASE icingaweb;
```

Create a file, `/etc/icinga2/features-enabled/ido-pgsql.conf`, with the following contents. Use the Icinga2 database password you specified earlier:

```
library "db_ido_pgsql"

object IdoPgsqlConnection "ido-pgsql" {
    user = "icinga2",
```

```
password = "YOURDATABASEPASSWORD",
host = "localhost",
database = "icinga2"
}
```

Install the Icinga2 PostgreSQL configuration:

```
$ apt-get install icinga2-ido-pgsql
# splash screen - "configure now"
# Yes, fill in password

$ icinga2 feature enable ido-pgsql
$ service icinga2 restart
```

Install icingaweb2:

```
$ apt-get install icingaweb2

# If it installs apache2, remove it, as we'll be using nginx.
$ apt-get remove apache2 --purge
```

Create the following nginx configuration at `/etc/nginx/sites-available/icinga2`, adjust where needed:

```
server {
    listen 0.0.0.0:80;

    server_name icinga2.yourdomain.tld;

    location ~ ^/index\.php(.*)$ {
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME /usr/share/icingaweb2/public/index.php;
        fastcgi_param ICINGAWEB_CONFIGDIR /etc/icingaweb2;
    }

    location ~ ^/(.*)? {
        allow all;

        alias /usr/share/icingaweb2/public;
        index index.php;

        rewrite ^/$ /dashboard;
        try_files $1 $uri $uri/ /index.php$is_args$args;
    }
}
```

Make a symbolic link to the sites-enabled directory:

```
$ ln -s /etc/nginx/sites-available/icinga2 /etc/nginx/sites-enabled/icinga2
$ service nginx reload
```

The icingaweb2 service is now accessible through <http://<ip>:<port>>. Setup the web interface through the website. To start the setup, a one-time token is required for authentication, create it as follows:

```
icingacli setup token create
```

As for the setup itself, take the following steps.

- Step ‘modules’, click Next
- Step ‘icinga web 2’, should be all green
- Step ‘Authentication’, click Next
- Step ‘Database Resource’, fill in PostgreSQL details
- step ‘Authentication Backend’, click Next
- step ‘Administration’, create an admin account
- Next on all steps

After this is finished, login to the icinga2 web interface and notice that icinga2 is already logging the current machine.

Configuring the Icinga2 master

As this is the master node, we will have to configure it as such. We’ll use the wizard. Start as follows:

```
$ icinga2 node wizard
Welcome to the Icinga 2 Setup Wizard!

We'll guide you through all required configuration details.

Please specify if this is a satellite setup ('n' installs a master setup) [Y/n]: n
Starting the Master setup routine...
Please specify the common name (CN) [cuckooringa2]:
Checking for existing certificates for common name 'cuckooringa2'...
Certificates not yet generated. Running 'api setup' now.
[...]
Please specify the API bind host/port (optional):
Bind Host []: <YOUR IP ADDRESS>
Bind Port []: <YOUR PORT>
information/cli: Created backup file '/etc/icinga2/features-available/api.conf.orig'.
information/cli: Updating constants.conf.
information/cli: Created backup file '/etc/icinga2/constants.conf.orig'.
information/cli: Updating constants file '/etc/icinga2/constants.conf'.
information/cli: Updating constants file '/etc/icinga2/constants.conf'.
information/cli: Updating constants file '/etc/icinga2/constants.conf'.
Done.

Now restart your Icinga 2 daemon to finish the installation!

$ service icinga2 restart
```

The setup wizard will do the following:

- Generate a local CA in /var/lib/icinga2/ca (or use existing one)
- Generate a new CSR, sign it with the local CA and copying it into /etc/icinga2/pki
- Generate a local zone and endpoint configuration for this master based on FQDN
- Enabling the API feature, and setting optional bind_host and bind_port
- Setting the NodeName and TicketSalt constants in constants.conf

Create or modify the /etc/icinga2/zones.conf file and populate it with the following configuration (please customize as needed):

```
object Endpoint "icinga2.yourdomain.tld" {
}

object Zone "icinga2.yourdomain.tld" {
    // This is the local master zone = "master"
    endpoints = [ "icinga2.yourdomain.tld" ]
}
```

Finally restart Icinga2 once again to make sure all settings are applied:

```
service icinga2 restart
```

Notifications Events

We're almost done on the master. We're going to configure Icinga2 to call our custom script, `/etc/icinga2/scripts/notify.py` whenever the services `ping4`, `ssh`, and `check_cuckoo` fail. It is up to the user of Cuckoo to implement the actual `notify.py` script though, as this is out of scope for this documentation.

First of all, on master, append the following lines to `/etc/icinga2/conf.d/users.conf`:

```
object User "sysadmin" {
    display_name = "System Administrator"
    enable_notifications = true
    states = [ Warning, Critical ]
    types = [ Problem, Recovery ]
    email = "YOUREMAILADDRESS@YOURDOMAIN.TLD"
}

template Notification "generic-notification" {
    states = [ Warning, Critical, Unknown ]
    types = [ Problem, Acknowledgement, Recovery, Custom, FlappingStart,
        FlappingEnd, DowntimeStart, DowntimeEnd, DowntimeRemoved ]
}

apply Notification "notify-sysadmin" to Service {
    import "generic-notification"

    command = "notify-cuckoo"
    users = [ "sysadmin" ]

    assign where service.name in ["check_cuckoo", "ssh", "ping4"]
}

object NotificationCommand "notify-cuckoo" {
    import "plugin-notification-command"
    command = [
        SysconfDir + "/icinga2/scripts/notify.py"
    ]

    env = {
        NOTIFICATIONTYPE = "$notification.type$"
        SERVICEDESC = "$service.name$"
        HOSTALIAS = "$host.display_name$"
        HOSTADDRESS = "$address$"
        SERVICESTATE = "$service.state$"
        LONGDATETIME = "$icinga.long_date_time$"
    }
}
```

```

SERVICEOUTPUT = "$service.output$"
NOTIFICATIONAUTHORNAME = "$notification.author$"
NOTIFICATIONCOMMENT = "$notification.comment$"
HOSTDISPLAYNAME = "$host.display_name$"
SERVICEDISPLAYNAME = "$service.display_name$"
USEREMAIL = "$user.email$"
}
}

```

Then create the file `/etc/icinga2/scripts/notify.py` and have some meaningful code in there. It'll be called every time a service fails or recovers (you may want to use the ENV vars). Don't forget to make it executable:

```
$ chmod +x /etc/icinga2/scripts/notify.py
```

Configuring a Icinga2 satellite (client)

A satellite Icinga2 node connects to the master Icinga2 node using SSL. To get started, install Icinga2 on the satellite node, i.e., a Cuckoo node.

First add the apt key for Icinga2 on Ubuntu:

```

$ wget -O - http://packages.icinga.org/icinga.key | apt-key add -
$ echo 'deb http://packages.icinga.org/ubuntu icinga-trusty main' > /etc/apt/sources.
↪list.d/icinga.list
$ apt-get update

```

Or on Debian:

```

$ wget -O - http://packages.icinga.org/icinga.key | apt-key add -
$ echo 'deb http://packages.icinga.org/debian icinga-jessie main' > /etc/apt/sources.
↪list.d/icinga.list
$ apt-get update

```

Then install Icinga2 itself:

```

$ apt-get install icinga2
$ icinga2 feature enable command
$ service icinga2 restart

```

To have this satellite connect to master, we once again use the wizard to properly configure it:

```

$ icinga2 node wizard
- Satellite setup? [Y/n]: y
- For the common name, use the master common name you supplied earlier doing the_
↪wizard for master.
- Establish a connection to master? [Y/n]: y
- Fill in connection details to master
- Leave CSR signing connection details blank
- Please specify the request ticket: run the *hint* cmd on master to acquire the_
↪ticket and use it
- Leave API blank
- Accept config from master? [y/N]: y
- Accept commands from master? [y/N]: y

```

As an example wizard session:

```
$ icinga2 node wizard
Welcome to the Icinga 2 Setup Wizard!

We'll guide you through all required configuration details.

Please specify if this is a satellite setup ('n' installs a master setup) [Y/n]: y
Please specify the common name (CN) [cuckool]:
Please specify the master endpoint(s) this node should connect to:
Master Common Name (CN from your master setup): icinga2.yourdomain.tld
Do you want to establish a connection to the master from this node? [Y/n]: y
Please fill out the master connection information:
Master endpoint host (Your master's IP address or FQDN): <YOUR IP ADDRESS>
Master endpoint port [5665]: <YOUR PORT NUMBER>
Add more master endpoints? [y/N]: n
Please specify the master connection for CSR auto-signing (defaults to master_
→endpoint host):
Host [...]:
Port [...]:
[...]
Is this information correct? [y/N]: y
[...]
Please specify the request ticket generated on your Icinga 2 master.
(Hint: # icinga2 pki ticket --cn 'cuckool'): [...]
[...]
Please specify the API bind host/port (optional):
Bind Host []:
Bind Port []:
Accept config from master? [y/N]: y
Accept commands from master? [y/N]: y
[...]
Now restart your Icinga 2 daemon to finish the installation!

$ service icinga2 restart
```

To have master notice the newly added satellite, run the following commands on the server where the Icinga2 master is running:

```
$ icinga2 node update-config
$ service icinga2 restart

# Optionally you may verify the current configuration.
$ icinga2 object list --type Host
```

The newly added satellite should show up in the list.

Setting up the Cuckoo check service

We'll make a custom service that checks if Cuckoo is currently working on the satellite. This code will run locally on each satellite node.

On the satellite create the following file `/usr/lib/nagios/plugins/check_cuckoo` with the following contents.

```
#!/usr/bin/python
import sys
import argparse
import requests
```



```

from math import log

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-H", "--host", help="API server host",
                        default="localhost", action="store", required=True)
    parser.add_argument("-p", "--port", help="API server port",
                        default=8090, action="store", required=True)
    args = parser.parse_args()

def pretty_size(n, pow=0, b=1024, u='B', pre=[''] + [p + 'i' for p in 'KMGTPEZY']):
    pow, n = min(int(log(max(n * b ** pow, 1), b)), len(pre) - 1), n * b ** pow
    return "%s.%sif %s%s" % abs(pow % (-pow - 1)) % (n / b ** float(pow), pre[pow],
    ↪u)

def json_to_nagios(blob, base=""):
    def _format(key, val):
        if "diskspace" in key and isinstance(val, (int, float)):
            size = pretty_size(val, b=1024, u='B', pre=['', 'K', 'M', 'G'])
            return "%s=%s" % (key, size.replace(" ", ""))
        elif isinstance(val, (int, float)):
            return "%s=%s" % (key, str(val))
        else:
            # returning strings in nagios labels not allowed
            return ""

    rtn = ""
    if isinstance(blob, dict):
        for _k, _v in blob.iteritems():
            key = base + "_" + _k
            if isinstance(_v, dict):
                rtn += json_to_nagios(_v, key)
            else:
                rtn += _format(key, _v)
    elif isinstance(blob, list):
        #rtn += "%s=(%s)" % (base, ",".join([str(z) for z in blob]))
        pass
    else:
        rtn += _format(base, str(blob))

    return rtn

url = "http://%s:%s/cuckoo/status" % (args.host, args.port)

try:
    resp = requests.get(url, timeout=5)
    if not resp.status_code == 200:
        raise Exception("status code not 200")

    resp = resp.json()
except Exception as ex:
    print "Error - %s" % str(ex)
    sys.exit(2)

output = "Cuckoo %s OK|" % resp["version"]

for k, v in resp.iteritems():
    output += json_to_nagios(v, base=k)

```

```
print output
sys.exit(0)
```

Don't forget to make it executable:

```
$ chmod +x /usr/lib/nagios/plugins/check_cuckoo
```

Now open `/etc/icinga2/conf.d/services.conf`, remove everything, and paste the following lines:

```
apply Service "ping4" {
    import "generic-service"
    check_command = "ping4"
    assign where host.address
}

apply Service "ssh" {
    import "generic-service"
    check_command = "ssh"
    assign where (host.address || host.address6) && host.vars.os == "Linux"
}

apply Service for (disk => config in host.vars.disks) {
    import "generic-service"
    check_command = "disk"
    vars += config
}

apply Service "icinga" {
    import "generic-service"
    check_command = "icinga"
    assign where host.name == NodeName
}

apply Service "load" {
    import "generic-service"
    check_command = "load"
    /* Used by the ScheduledDowntime apply rule in `downtimes.conf`. */
    vars.backup_downtime = "02:00-03:00"
    assign where host.name == NodeName
}

apply Service "swap" {
    import "generic-service"
    check_command = "swap"
    assign where host.name == NodeName
}

apply Service "check_cuckoo" {
    import "generic-service"
    check_command = "check_cuckoo"
    assign where host.name == NodeName
}
```

Then append the following lines to `/etc/icinga2/conf.d/commands.conf`:

```
object CheckCommand "check_cuckoo" {
    import "plugin-check-command"
```

```
command = [ PluginDir + "/check_cuckoo" ]

arguments = {
    "-H" = "127.0.0.1",
    "-p" = "8090"
}
```

To finish off the installation of this satellite, run the following two commands on both the satellite and the master:

```
$ icinga2 node update-config
$ service icinga2 restart
```

The service checks for this satellite should now be visible in the Icinga2 dashboard and you should now have realtime monitoring enabled for your Cuckoo node.

Configuration (Android Analysis)

Deprecated since version 2.0-rc2: Android Analysis may not work as expected due to the changes to becoming a Cuckoo Package. Proper Android integration will be picked up as a Cuckoo update in the future.

To get Cuckoo running Android analysis you should download the [Android SDK](#) and extract it in a folder Cuckoo can access. You should also configure *avd.conf* with the settings of your setup.

avd.conf

The main file for Android environment settings is `$CWD/conf/avd.conf`, it contains all the generic configuration used to launch the Android emulator and run the analysis.

The file is largely commented and self-explanatory, but some important options are as follows:

- **emulator_path:** The path to the Android emulator (it is located inside Android SDK).
- **adb_path:** The path to the Android Debug Bridge utility (it is located inside Android SDK).
- **avd_path:** The path where the AVD images are located.

Preparing the Guest

At this point you should have configured the Cuckoo host component and you should have designed and defined the number and the names of the virtual machines you are going to use for malware execution.

Now it's time to create such machines and to configure them properly.

Creation of the Virtual Machine

Once you have *properly installed* your virtualization software, you can proceed on creating all the virtual machines you need.

Using and configuring your virtualization software is out of the scope of this guide, so please refer to the official documentation.

Note: You can find some hints and considerations on how to design and create your virtualized environment in the [Sandboxing](#) chapter.

Note: We recommend either 64-bit Windows 7 or Windows XP virtual machines. For Windows 7 you will have to disable User Access Control.

Changed in version 2.0-rc2: We used to suggest Windows XP as a guest VM but nowadays a 64-bit Windows 7 machine yields much better results.

Note: KVM Users - Be sure to choose a hard drive image format that supports snapshots. See [Saving the Virtual Machine](#) for more information.

When creating the virtual machine, Cuckoo doesn't require any specific configuration. You can choose the options that best fit your needs.

Requirements

In order to make Cuckoo run properly in your virtualized Windows system, you will have to install some required software and libraries.

Install Python

Python is a strict requirement for the Cuckoo guest component (*analyzer*) in order to run properly.

You can download the proper Windows installer from the [official website](#). Also in this case Python 2.7 is preferred.

Some Python libraries are optional and provide some additional features to Cuckoo guest component. They include:

- **Python Pillow:** it's used for taking screenshots of the Windows desktop during the analysis.

They are not strictly required by Cuckoo to work properly, but you are encouraged to install them if you want to have access to all available features. Make sure to download and install the proper packages according to your Python version.

Additional Software

At this point you should have installed everything needed by Cuckoo to run properly.

Depending on what kind of files you want to analyze and what kind of sandboxed Windows environment you want to run the malware samples in, you might want to install additional software such as browsers, PDF readers, office suites etc. Remember to disable the "auto update" or "check for updates" feature of any additional software.

This is completely up to you and to what your needs are. You can get some hints by reading the [Sandboxing](#) chapter.

Network Configuration

Now it's time to setup the network for your virtual machine.

Windows Settings

Before configuring the underlying networking of the virtual machine, you might want to tweak some settings inside Windows itself.

One of the most important things to do is **disabling Windows Firewall** and the *Automatic Updates*. The reason behind this is that they can affect the behavior of the malware under normal circumstances and that they can pollute the network analysis performed by Cuckoo, by dropping connections or including irrelevant requests.

You can do so from Windows' Control Panel as shown in the picture:



Virtual Networking

Now you need to decide how to make your virtual machine able to access Internet or your local network.

While in previous releases Cuckoo used shared folders to exchange data between the Host and Guests, from release 0.4 it adopts a custom agent that works over the network using a simple XMLRPC protocol.

In order to make it work properly you'll have to configure your machine's network so that the Host and the Guest can communicate. Testing the network access by pinging a guest is a good practice, to make sure the virtual network was set up correctly. Use only static IP addresses for your guest, as Cuckoo doesn't support DHCP and using it will break your setup.

This stage is very much up to your own requirements and to the characteristics of your virtualization software.

Warning: Virtual networking errors! Virtual networking is a vital component for Cuckoo, you must be really sure to get connectivity between host and guest. Most of the issues reported by users are related to a wrong setup of their networking. If you aren't sure about that check your virtualization software documentation and test connectivity with ping and telnet.

The recommended setup is using a **Host-Only networking layout** with proper forwarding. More on such network routing can be found in *Per-Analysis Network Routing*, which is part of the host machine setup.

Installing the Agent

From release 0.4 Cuckoo adopts a custom agent that runs inside the Guest and that handles the communication and the exchange of data with the Host. This agent is designed to be cross-platform, therefore you should be able to use it on Windows, Android, Linux, and Mac OS X. In order to make Cuckoo work properly, you'll have to install and start this agent.

It's quite simple.

In the `$CWD/agent/` directory you will find the `agent.py` file. Copy this file to the Guest operating system (in whatever way you want, perhaps a temporary shared folder or by downloading it from a webserver on the host, we recommend the latter) and run it. The Agent will launch a small API server that the host will be able to talk to.

On Windows simply launching the script will also spawn a Python window, if you want to hide it you can rename the file from `agent.py` to **agent.pyw** which will prevent the console window from spawning.

If you want the script to be launched at Windows' boot, just place the file in the *Startup* folder.

Saving the Virtual Machine

Now you should be ready to save the virtual machine to a snapshot state.

Before doing this **make sure you rebooted it softly and that it's currently running, with Cuckoo's agent running and with Windows fully booted.**

Now you can proceed saving the machine. The way to do it obviously depends on the virtualization software you decided to use.

If you follow all the below steps properly, your virtual machine should be ready to be used by Cuckoo.

VirtualBox

If you are going for VirtualBox you can take the snapshot from the graphical user interface or from the command line:

```
$ VBoxManage snapshot "<Name of VM>" take "<Name of snapshot>" --pause
```

After the snapshot creation is completed, you can power off the machine and restore it:

```
$ VBoxManage controlvm "<Name of VM>" poweroff
$ VBoxManage snapshot "<Name of VM>" restorecurrent
```

KVM

If decided to adopt KVM, you must first of all be sure to use a disk format for your virtual machines which supports snapshots. By default libvirt tools create RAW virtual disks, and since we need snapshots you'll either have to use QCOW2 or LVM. For the scope of this guide we adopt QCOW2, which is easier to setup than LVM.

The easiest way to create such a virtual disk correctly is using the tools provided by the libvirt suite. You can either use `virsh` if you prefer command-line interfaces or `virt-manager` for a nice GUI. You should be able to directly create it in QCOW2 format, but in case you have a RAW disk you can convert it like this:

```
$ cd /your/disk/image/path
$ qemu-img convert -O qcow2 your_disk.raw your_disk.qcow2
```

Now you have to edit your VM definition as follows:

```
$ virsh edit "<Name of VM>"
```

Find the disk section, it looks like this:

```
<disk type='file' device='disk'>
  <driver name='qemu' type='raw' />
  <source file='/your/disk/image/path/your_disk.raw' />
  <target dev='hda' bus='ide' />
  <address type='drive' controller='0' bus='0' unit='0' />
</disk>
```

And change “type” to qcow2 and “source file” to your qcow2 disk image, like this:

```
<disk type='file' device='disk'>
  <driver name='qemu' type='qcow2' />
  <source file='/your/disk/image/path/your_disk.qcow2' />
  <target dev='hda' bus='ide' />
  <address type='drive' controller='0' bus='0' unit='0' />
</disk>
```

Now test your virtual machine, if everything works prepare it for snapshotting while running Cuckoo’s agent. This means the virtual machine needs to be running while you are taking the snapshot. Then you can shut it down. You can finally take a snapshot with the following command:

```
$ virsh snapshot-create "<Name of VM>"
```

Having multiple snapshots can cause errors:

```
ERROR: No snapshot found for virtual machine VM-Name
```

VM snapshots can be managed using the following commands:

```
$ virsh snapshot-list "VM-Name"
$ virsh snapshot-delete "VM-Name" 1234567890
```

VMware Workstation

If you decided to adopt VMware Workstation, you can take the snapshot from the graphical user interface or from the command line:

```
$ vmrun snapshot "/your/disk/image/path/wmware_image_name.vmx" your_snapshot_name
```

Where your_snapshot_name is the name you choose for the snapshot. After that power off the machine from the GUI or from the command line:

```
$ vmrun stop "/your/disk/image/path/wmware_image_name.vmx" hard
```

XenServer

If you decided to adopt XenServer, the XenServer machinery supports starting virtual machines from either disk or a memory snapshot. Creating and reverting memory snapshots require that the Xen guest tools be installed in the virtual machine. The recommended method of booting XenServer virtual machines is through memory snapshots because they can greatly reduce the boot time of virtual machines during analysis. If, however, the option of installing the guest tools is not available, the virtual machine can be configured to have its disks reset on boot. Resetting the disk ensures that malware samples cannot permanently modify the virtual machine.

Memory Snapshots

The Xen guest tools can be installed from the XenCenter application that ships with XenServer. Once installed, restart the virtual machine and ensure that the Cuckoo agent is running.

Snapshots can be taken through the XenCenter application and the command line interface on the control domain (Dom0). When creating the snapshot from XenCenter, ensure that the “Snapshot disk and memory” is checked. Once created, right-click on the snapshot and note the snapshot UUID.

To snapshot from the command line interface, run the following command:

```
$ xe vm-checkpoint vm="vm_uuid_or_name" new-name-label="Snapshot Name/Description"
```

The snapshot UUID is printed to the screen once the command completes.

Regardless of how the snapshot was created, save the UUID in the virtual machine’s configuration section. Once the snapshot has been created, you can shutdown the virtual machine.

Bootting from Disk

If you can’t install the Xen guest tools or if you don’t need to use memory snapshots, you will need to ensure that the virtual machine’s disks are reset on boot and that the Cuckoo agent is set to run at boot time.

Running the agent at boot time can be configured in Windows by adding a startup item for the agent.

The following commands must be run while the virtual machine is powered off.

To set the virtual machine’s disks to reset on boot, you’ll first need to list all the attached disks for the virtual machine. To list all attached disks, run the following command:

```
$ xe vm-disk-list vm="vm_name_or_uuid"
```

Ignoring all CD-ROM and read-only disks, run the following command for each remaining disk to change it’s behavior to reset on boot:

```
$ xe vdi-param-set uuid="vdi_uuid" on-boot=reset
```

After the disk is set to reset on boot, no permanent changes can be made to the virtual machine’s disk. Modifications that occur while a virtual machine is running will not persist past shutdown.

Cloning the Virtual Machine

In case you planned to use more than one virtual machine, there’s no need to repeat all the steps done so far: you can clone it. In this way you’ll have a copy of the original virtualized Windows with all requirements already installed.

The new virtual machine will also contain all the settings of the original one, which is not good. Now you need to proceed repeating the steps explained in *Network Configuration*, *Installing the Agent* and *Saving the Virtual Machine* for this new machine.

Preparing the Guest (Physical Machine)

Warning: This chapter only applies for a Physical Machine setup! For normal Cuckoo usage please ignore it.

At this point you should have configured the Cuckoo host component and you should have designed and defined the number and the names of the physical machines you are going to use for malware execution.

Now it's time to create such machines and to configure them properly.

Creation of the Physical Machine

Once you have *properly installed* your imaging software, you can proceed on creating all the physical machines you need.

Using and configuring your imaging software is out of the scope of this guide, so please refer to the official documentation.

Note: You can find some hints and considerations on how to design and create your virtualized environment in the *Sandboxing* chapter.

Note: For analysis purposes you are recommended to use Windows XP Service Pack 3, but Cuckoo Sandbox also proved to work with Windows 7 with User Access Control disabled.

When creating the physical machine, Cuckoo doesn't require any specific configuration. You can choose the options that best fit your needs.

Requirements

In order to make Cuckoo run properly in your physical Windows system, you will have to install some required software and libraries.

Install Python

Python is a strict requirement for the Cuckoo guest component (*analyzer*) in order to run properly.

You can download the proper Windows installer from the [official website](#). Also in this case Python 2.7 is preferred.

Some Python libraries are optional and provide some additional features to Cuckoo guest component. They include:

- **Python Pillow:** it's used for taking screenshots of the Windows desktop during the analysis.

They are not strictly required by Cuckoo to work properly, but you are encouraged to install them if you want to have access to all available features. Make sure to download and install the proper packages according to your Python version.

Additional Software

At this point you should have installed everything needed by Cuckoo to run properly.

Depending on what kind of files you want to analyze and what kind of sandboxed Windows environment you want to run the malware samples in, you might want to install additional software such as browsers, PDF readers, office suites etc. Remember to disable the “auto update” or “check for updates” feature of any additional software.

This is completely up to you and to what your needs are. You can get some hints by reading the *Sandboxing* chapter.

Additional Host Requirements

The physical machine manager uses RPC requests to reboot physical machines. The *net* command is required for this to be accomplished, and is available from the *samba-common-bin* package.

On Debian/Ubuntu you can install it with:

```
$ sudo apt-get install samba-common-bin
```

In order for the physical machine manager to work, you must have a way for physical machines to be returned to a clean state. In development/testing *Fog* was used as a platform to handle re-imaging the physical machines. However, any re-imaging platform can be used (Clonezilla, Deepfreeze, etc) to accomplish this.

Network Configuration

Now it's time to setup the network for your physical machine.

Windows Settings

Before configuring the underlying networking of the sandbox, you might want to tweak some settings inside Windows itself.

One of the most important things to do is **disabling** *Windows Firewall* and the *Automatic Updates*. The reason behind this is that they can affect the behavior of the malware under normal circumstances and that they can pollute the network analysis performed by Cuckoo, by dropping connections or including irrelevant requests.

You can do so from Windows' Control Panel as shown in the picture:



Using a physical machine manager requires a few more configuration options than the virtual machine managers in order to run properly. In addition to the steps laid out in the regular Preparing the Guest section, some settings need to be changed for physical machines to work properly.

- Enable auto-login (Allows for the agent to start upon reboot)
- Enable Remote RPC (Allows for Cuckoo to reboot the sandbox using RPC)
- Turn off paging (Optional)
- Disable Screen Saver (Optional)

In Windows 7 the following commands can be entered into an Administrative command prompt to enable auto-login and Remote RPC.

```
reg add "hkml\software\Microsoft\Windows NT\CurrentVersion\WinLogon" /v
↪DefaultUserName /d <USERNAME> /t REG_SZ /f
reg add "hkml\software\Microsoft\Windows NT\CurrentVersion\WinLogon" /v
↪DefaultPassword /d <PASSWORD> /t REG_SZ /f
reg add "hkml\software\Microsoft\Windows NT\CurrentVersion\WinLogon" /v
↪AutoAdminLogon /d 1 /t REG_SZ /f
reg add "hkml\system\CurrentControlSet\Control\TerminalServer" /v AllowRemoteRPC /d
↪0x01 /t REG_DWORD /f
reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System"
↪ /v LocalAccountTokenFilterPolicy /d 0x01 /t REG_DWORD /f
```

Networking

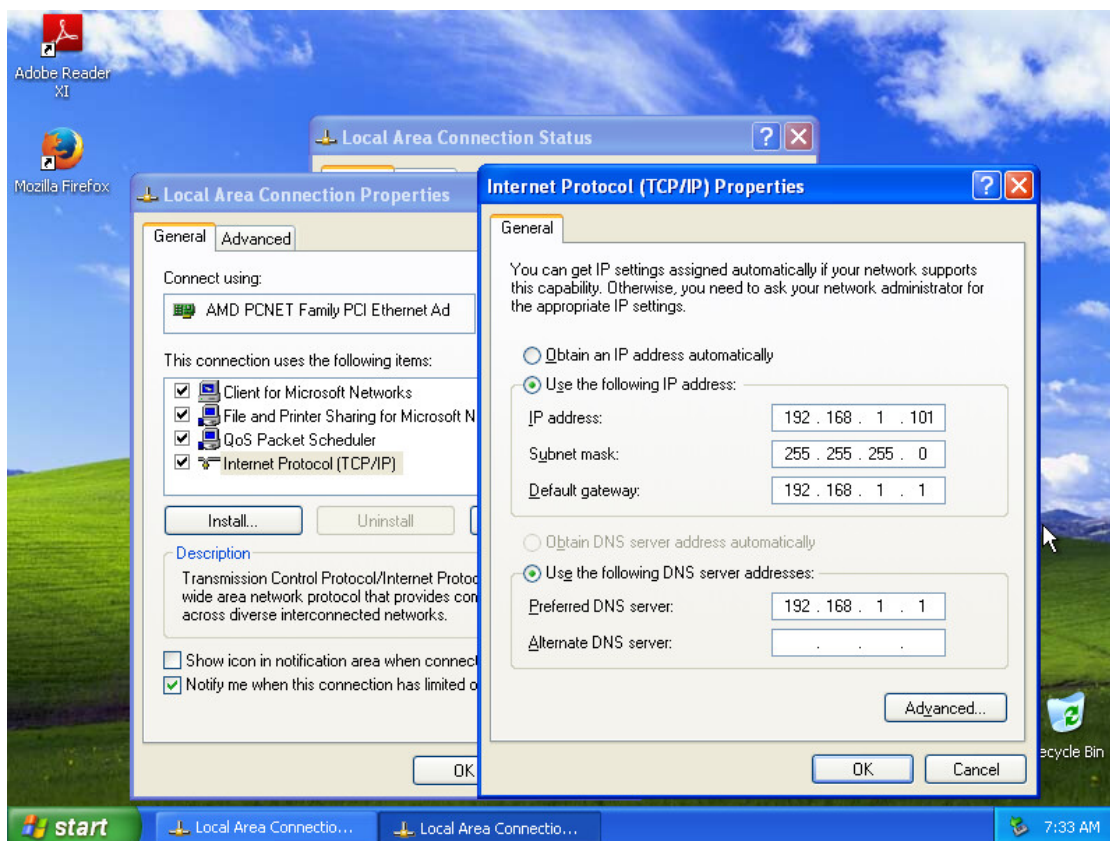
Now you need to decide how to make your physical machine able to access Internet or your local network.

While in previous releases Cuckoo used shared folders to exchange data between the Host and Guests, from release 0.4 it adopts a custom agent that works over the network using a simple XMLRPC protocol.

In order to make it work properly you'll have to configure your machine's network so that the Host and the Guest can communicate. Testing the network access by pinging a guest is a good practice, to make sure the virtual network was set up correctly. Use only static IP addresses for your guest, as today Cuckoo doesn't support DHCP and using it will break your setup.

This stage is very much up to your own requirements and to the characteristics of your virtualization software.

For physical machines, make sure when setting the IP address of the guest to also set the Gateway and DNS server to be the IP address of the Cuckoo server on the physical network. For example, if your Cuckoo server has the IP address of 192.168.1.1, then you would set the Gateway and DNS server in Windows Settings to be 192.168.1.1 as well.



Installing the Agent

Installing the Agent on a Physical machine is the same as installing it in a Virtual Machine, therefore please refer to *Installing the Agent*.

Saving the Guest

Now you should be ready to save the physical machine to a clean state. In order for the physical machine manager to work, you must have a way for physical machines to be returned to a clean state.

Before doing this **make sure you rebooted it softly and that it's currently running, with Cuckoo's agent running and with Windows fully booted.**

Now you can proceed saving the machine. The way to do it obviously depends on the imaging software you decided to use.

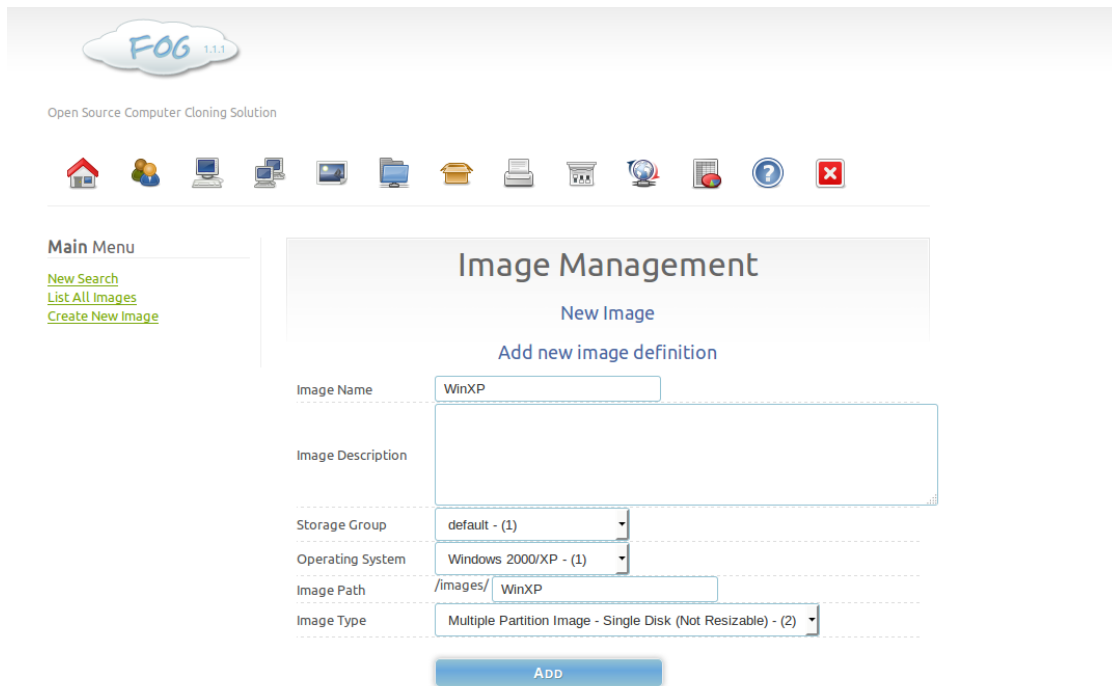
In development/testing Fog (<http://www.fogproject.org/>) was used as a platform to handle re-imaging the physical machines. However, any re-imaging platform can be used (Clonezilla, Deepfreeze, etc.) to accomplish this.

If you follow all the below steps properly, your virtual machine should be ready to be used by Cuckoo.

Fog

After installing Fog, you will need to create an image and add an image and a host to the Fog server.

To add an image to the fog server, open the Image Management window (http://<your_fog_server>/fog/management/index.php?node=images) and click “Create New Image.” Provide the proper inputs for your OS configuration and click “Add”



The screenshot shows the Fog 1.3.3 web interface. At the top, there's a logo and the text "Open Source Computer Cloning Solution". Below that is a navigation bar with icons for home, users, computers, networks, images, printers, storage, and help. The main content area is titled "Image Management" and contains a "New Image" section. This section has a form with the following fields:

- Image Name:** WinXP
- Image Description:** (empty text area)
- Storage Group:** default - (1)
- Operating System:** Windows 2000/XP - (1)
- Image Path:** /images/ WinXP
- Image Type:** Multiple Partition Image - Single Disk (Not Resizable) - (2)

An "Add" button is located at the bottom of the form.

Next you will need to add the host you plan to re-image to Fog. To add a host, open a web browser and navigate to the Host Management page of Fog (http://<your_fog_server>/fog/management/index.php?node=host). Click “Create New Host.” Provide the proper inputs for your host configuration. Be sure to select the image you created above from the “Host Image” option, when finished click the “Add” button.

FOG 1.1.1

Open Source Computer Cloning Solution

Main Menu

- [New Search](#)
- [List All Hosts](#)
- [Create New Host](#)
- [Export Hosts](#)
- [Import Hosts](#)

Host Management

New Host

Add new host definition

Host Name: WinXP *

Primary MAC: 00:01:02:03:04:05 *

Host Description:

Host Image: WinXP_v1.00 - (1)

Host Kernel:

Host Kernel Arguments:

Host Primary Disk:

Active Directory

Join Domain after image task: ☐

Domain Name:

Domain OU:

Domain Username:

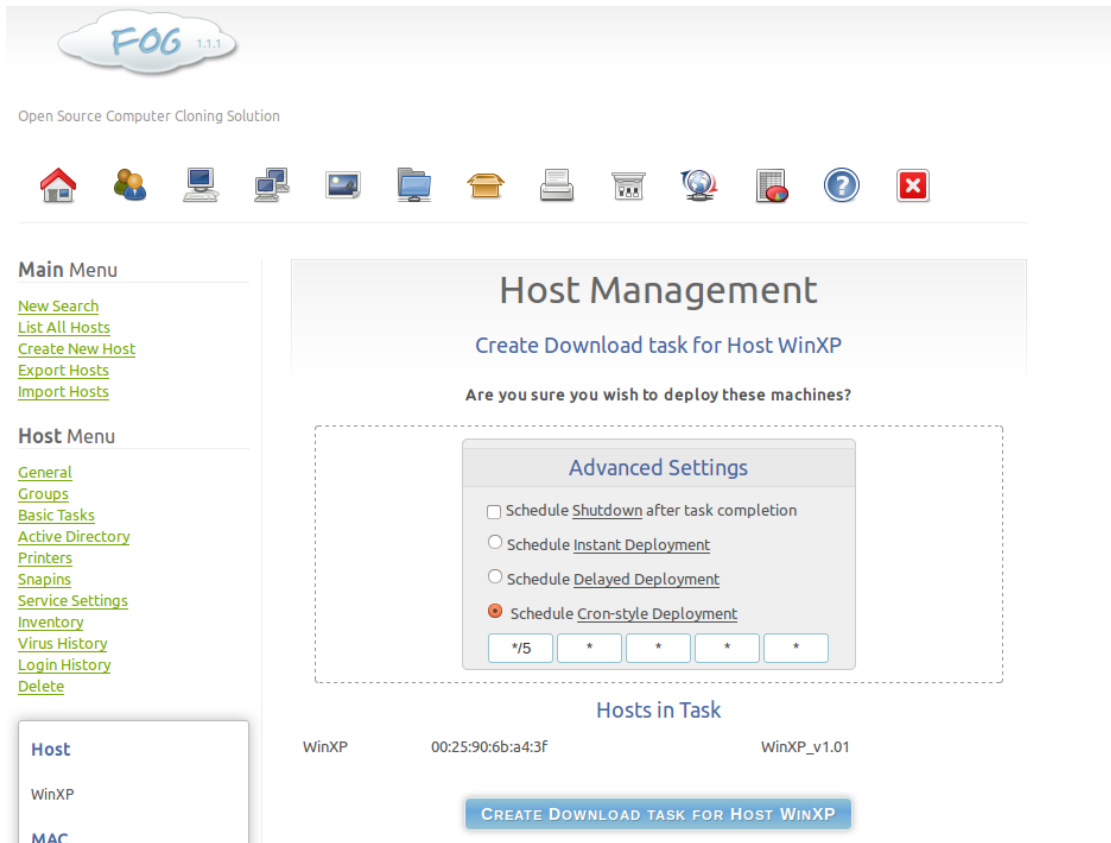
Domain Password:

Must be encrypted: ☐

Add

At this point you should be ready to take an image from the guest machine. In order to take an image you will need to navigate to the Task Management page and list all hosts (http://<your_fog_server>/fog/management/index.php?node=tasks&sub=listhosts). From here you should be able to click the Upload icon (Green up arrow), which should instantly add a task to the queue to take an image. Now you should reboot your Cuckoo guest image and it should PXE boot into Fog and capture the base image from the cuckoo guest.

After you have successfully taken an image of the guest machine, you can use that image as one to deploy to the Cuckoo physical sandbox as needed. It is recommended to use a scheduled task to accomplish this. In order to create a scheduled task to re-image sandboxes, navigate to the Host Management page on Fog (http://<your_fog_server>/fog/management/index.php?node=host&sub=list). Then click “Download” the machine you wish to schedule the re-image task for. From this menu, select “Schedule Cron-style Deployment” and put in the values you wish for the schedule to apply to (* / 5 * * * *) in the case shown in the screenshot below, but you may need to tweak these times for your environment.



Setup using VMWare (Bonus!)

Traditionally Cuckoo requires to be running some sort of virtualization software (e.g. VMware, Virtualbox, etc). The physical machine manager will also work with other virtual machines, so long as they are configured to revert to a snapshot on shutdown/reboot, and running the agent.py script. A use case for this functionality would be to run the cuckoo server and the guest sandboxes each in their own virtual machine on a single host, allowing for development/testing of Cuckoo without requiring a dedicated Linux host.

Upgrading from a previous release

New in version 2.0.0: Automatically upgrade from an older Cuckoo setup into a new one by importing the old setup.

This document describes the process of *importing* an older Cuckoo setup in order to upgrade your Cuckoo to the latest and greatest version. This importing process is possible for **Cuckoo 0.6 and upwards**. Naturally it doesn't re-apply any **custom code changes** that you applied to your old setup, but it does migrate your database, configuration, and analyses to the new version (in a best-effort manner).

Now, in order to upgrade your setup, you'll simply have to perform the following steps:

1. Come up with a *Cuckoo Working Directory* for the new setup (although the default one should work just fine, assuming it doesn't exist yet).
2. Optionally create a backup of your data (Cuckoo will also offer to do this for you before doing the actual setup import).
3. Run the `cuckoo import` command.

The cuckoo import command

The `cuckoo import` App performs a number of steps in order to import an older setup. Previously we had manual steps for performing a database migration, these have been integrated in the import process as well.

The usage of `cuckoo import` is as follows:

```
$ cuckoo import --help
Usage: cuckoo import [OPTIONS] PATH

Imports an older Cuckoo setup into a new CWD. The old setup should be
identified by PATH and the new CWD may be specified with the --cwd
parameter, e.g., "cuckoo --cwd /tmp/cwd import old-cuckoo".

Options:
  --copy      Copy all existing analyses to the new CWD (default)
  --move      Move all existing analyses to the new CWD
  --symlink   Symlink all existing analyses to the new CWD
  --help      Show this message and exit.
```

As per the limited usage documentation of this command, there is an input and an output directory and a couple of different *modes*. The rest is done by `cuckoo import` according to best-practice manners.

The three different modes are best described as follows. Keep in mind that these modes only inform the importing process on what to do with the existing analyses - these modes do not apply to any used databases or other data.

- **copy:** **Copies** all the analyses from the old setup to the new CWD. In this mode the old `storage/` folder will be copied to `$CWD/storage/`. The `copy` mode is useful if you want to maintain a backup of the old setup and its analyses, allowing one to restore it with the appropriate SQL backup. *Note that this mode will double the size of your existing analyses directory as it does a full copy.*
- **move:** **Moves** all the analyses from the old setup to the new CWD. In this mode the old `storage/` folder is moved to `$CWD/storage/`. After the import process you won't have a backup of your old data anymore, but you will be able to reference to it in the new CWD / setup.
- **symlink:** Creates a **symbolic link** from each analysis in the old setup, i.e., `storage/analyses/XYZ`, to the new CWD, i.e., `$CWD/storage/XYZ`. This method is the most desired (as you'll be able to access the existing analyses in both the old setup as well as the new CWD), but doesn't work on Windows.

The default mode is `copy` due to its feature of remaining available on both the old setup as well as the new CWD as well as being cross-platform (i.e., `symlink` mode isn't supported on Windows). After reading this documentation one may opt to go for `symlink` or `move` mode on non-Windows systems and `move` mode on Windows systems, though.

Following are the steps taken by Cuckoo when performing an import:

- The user has to accept a non-binding EULA-like agreement that (just kidding) attempts to inform him or her regarding the implications of importing an older setup.
- The version of the old Cuckoo setup is identified.
- It is ensured that the new CWD does **not** already exist.
- The old Cuckoo Configuration is read, **migrated**, and then validated to be fit for usage with the new Cuckoo version, i.e., you can configure a Cuckoo 0.6 setup and migrate it all the way to the latest version and it will simply work.
- The new CWD is created and it is configured with the migrated configuration.
- The user is prompted to *optionally* create a SQL database backup. On Linux-based systems this should work out of the box (and you'll get a hard error otherwise), but due to issues with `$PATH` this may require manually fixing up the command on Windows & Mac OS X systems.

- After the ability to create a SQL database backup, the **database schema** is **migrated** to the latest version **in-place**, i.e., you will not be able to use your old Cuckoo setup with this database anymore (hence the backup).
- Any and all existing analyses are imported to the new CWD using the `mode` as specified, or if it has not been specified, the default `copy` method.

You are now the happy owner of an up-to-date Cuckoo setup. Please inform us of any feedback that you may have through one of the various communication channels that we've put in-place.

Warning: One should **not** clean the old Cuckoo setup after the import. By attempting to do so you may loose the existing analyses (if `symlink` mode is used) and the SQL, MongoDB, and ElasticSearch databases.

Usage

This chapter explains how to use Cuckoo.

Starting Cuckoo

To start Cuckoo use the command:

```
$ cuckoo
```

You will get an output similar to this:

```

  eeee e   e eeee e   e   eeeee eeeee
  8  8 8   8 8  8 8   8  8  88 8  88
  8e   8e  8 8e   8eee8e 8   8 8   8
  88   88  8 88   88   8 8   8 8   8
  88e8 88ee8 88e8 88   8 8eee8 8eee8

Cuckoo Sandbox 2.0.0
www.cuckoosandbox.org
Copyright (c) 2010-2017

Checking for updates...
Good! You have the latest version available.

2017-03-31 17:08:53,527 [cuckoo.core.scheduler] INFO: Using "virtualbox" as machine_
↪manager
2017-03-31 17:08:53,935 [cuckoo.core.scheduler] INFO: Loaded 1 machine/s
2017-03-31 17:08:53,964 [cuckoo.core.scheduler] INFO: Waiting for analysis tasks.
```

Note that Cuckoo checks for updates on a remote API located at `api.cuckoosandbox.org`. You can avoid this by disabling the `version_check` option in the configuration file.

Now Cuckoo is ready to run and it's waiting for submissions.

cuckoo accepts some command line options as shown by the help:

```
$ cuckoo --help
Usage: cuckoo [OPTIONS] COMMAND [ARGS]...

Invokes the Cuckoo daemon or one of its subcommands.
```

To be able to use different Cuckoo configurations on the same machine with the same Cuckoo installation, we use the so-called Cuckoo Working Directory (aka "CWD"). A default CWD is available, but may be overridden through the following options - listed in order of precedence.

- * Command-line option (`--cwd`)
- * Environment option (`"CUCKOO"`)
- * Environment option (`"CUCKOO_CWD"`)
- * Current directory (if the `".cwd"` file exists)
- * Default value (`"~/cuckoo"`)

Options:

<code>-d, --debug</code>	Enable verbose logging
<code>-q, --quiet</code>	Only log warnings and critical messages
<code>-m, --maxcount INTEGER</code>	Maximum number of analyses to process
<code>--user TEXT</code>	Drop privileges to this user
<code>--cwd TEXT</code>	Cuckoo Working Directory
<code>--help</code>	Show this message and exit.

Commands:

<code>api</code>	Operate the Cuckoo REST API.
<code>clean</code>	Clean the CWD and associated databases.
<code>community</code>	Fetch supplies from the Cuckoo Community.
<code>distributed</code>	Distributed Cuckoo helper utilities.
<code>dnsserve</code>	Custom DNS server.
<code>import</code>	Imports an older Cuckoo setup into a new CWD.
<code>init</code>	Initializes Cuckoo and its configuration.
<code>machine</code>	Dynamically add/remove machines.
<code>migrate</code>	Perform database migrations.
<code>process</code>	Process raw task data into reports.
<code>rooter</code>	Instantiates the Cuckoo Rooter.
<code>submit</code>	Submit one or more files or URLs to Cuckoo.
<code>web</code>	Operate the Cuckoo Web Interface.

The `--debug` and `--quiet` flags increase and decrease the logging verbosity for the `cuckoo` command or any of its subcommands.

Cuckoo in the background

Running Cuckoo manually is useful the first few times you start using it, but if you're running multiple machines with Cuckoo on it, you will want the process of running Cuckoo to be automated.

Fortunately Cuckoo will automatically provide one with a `supervisord.conf` file in the Cuckoo Working Directory (this topic will be explained on the next page) which may be started either by running `supervisord` from the CWD directory, or by providing the configuration directly to `supervisord` as follows:

```
$ supervisord -c $CWD/supervisord.conf
```

It should be noted that, by default, `supervisord` will also start four *Processing Utility* instances, which means that, as per its documentation, the `process_results` configuration in `$CWD/conf/cuckoo.conf` should be disabled (i.e., change the value from `on` to `off`).

From there on, one may start and stop the various cuckoo processes (i.e., the main cuckoo process and the four processing instances) by running commands such as the following (assuming that they're run from the CWD):

```
# Stop the Cuckoo daemon and the processing utilities.
$ supervisorctl stop cuckoo:

# Start the Cuckoo daemon and the processing utilities.
$ supervisorctl start cuckoo:
```

Note that you'll need the trailing colon (i.e., `cuckoo:`) so to denote the Cuckoo supervisor group, containing the Cuckoo daemon process as well as the various processing utilities.

Cuckoo Working Directory Usage

Note: Before reading this page, please read on installing Cuckoo and the *Cuckoo Working Directory*.

Before we go into the subject of using the CWD we're first going to walk you through the many improvements on your Quality of Life during your daily usage of Cuckoo Sandbox with the introduction of the Cuckoo Package and CWD and some of the new features that come along with this.

So simply put, the CWD is a per-Cuckoo instance configuration directory. While people generally speaking only run one Cuckoo instance per server, this still yields a lot of maintenance-related improvements:

- As outlined by *Installing Cuckoo* installing Cuckoo and updating it will now be pretty much `pip install -U cuckoo`.
- Due to Cuckoo now being an official Python Package we have a much tighter control on how its installed on users' systems. No longer will users have incorrect versions of third party libraries installed breaking their setup.
- Because updating is much easier (again, `pip install -U cuckoo`) we will be able to **put out new versions more often**. E.g., when one or more users run into a bug, we'll be able to put out a fix quickly - this has happened a few times in the past in a way that we weren't able to properly mitigate such issues (leaving users high & dry for months).
- **The Cuckoo Configuration is no longer part of the Git repository.** Users who have updated Cuckoo in the past will have seen the effort involved in making a backup of their configuration, pulling a new version of Cuckoo, and either restoring their old configuration or applying the configuration against the new Cuckoo version by hand.
- With the new CWD all configurable files will be in one centralized place in logically structured subdirectories.
- Given that a CWD denotes *one* Cuckoo instance, it is possible to have multiple Cuckoo instances through multiple CWD's while having installed/deployed Cuckoo only once.
- With the addition of the `cuckoo` executable and its associated *Cuckoo Apps* (subcommands) **the various Cuckoo commands are now centralized into one command.**

Usage

After having installed the Cuckoo Package (*Installing Cuckoo*) and setup the initial Cuckoo Working Directory (*Cuckoo Working Directory*) it is time to actually get started with Cuckoo. Just to reiterate, installing the latest version of Cuckoo in a `virtualenv` environment may look roughly as follows (note the `pip install -U pip setuptools`, for more information see also *DistributionNotFound / No distribution matching the version.*).

```
$ virtualenv venv
$ . venv/bin/activate
(venv)$ pip install -U pip setuptools
```

```
(venv)$ pip install -U cuckoo
(venv)$ cuckoo --cwd ~/.cuckoo
```

First of all you'll probably want to update the default Cuckoo configuration in the `$CWD/conf/` directory. If just to switch from the default SQLite3 database to, e.g., PostgreSQL, or to register some virtual machines (more information on setting up Virtual Machines can be found in [Preparing the Guest](#)). Note that in order to view the results of analyses in the Web Interface later on it is necessary to enable the `mongodb` reporting module in `$CWD/conf/reporting.conf` (see also [Web interface](#)).

We then proceed by downloading the Cuckoo Community which includes over 300 Cuckoo Signatures which summarize a wide array of malicious behavior in a digestible way, simplifying the final results of an analysis. Downloading the Cuckoo Community into our CWD may be done as follows:

```
(venv)$ cuckoo community
```

Alternatively, if you have a local copy of the community `.tar.gz` file (e.g., after running `wget https://github.com/cuckoosandbox/community/archive/master.tar.gz`) this can be imported as follows:

```
(venv)$ cuckoo community --file master.tar.gz
```

Now we're good to go let's submit some samples and URLs using the command-line [Submission Utility](#). Note that multiple tasks may be submitted at once:

```
(venv)$ cuckoo submit /tmp/sample1.exe /tmp/sample2.exe /tmp/sample3.exe
Success: File "/tmp/sample1.exe" added as task with ID #1
Success: File "/tmp/sample2.exe" added as task with ID #2
Success: File "/tmp/sample3.exe" added as task with ID #3
(venv)$ cuckoo submit --url google.com bing.com
Success: URL "google.com" added as task with ID #4
Success: URL "bing.com" added as task with ID #5
```

For the actual analysis of these samples, one will have to run the Cuckoo daemon. Which is equally straightforward. Do keep in mind that, by default, the command will run indefinitely (unless a maximum analysis count was provided through the `-m` parameter, e.g., `-m 5`).

```
# This command is equal to what used to be "./cuckoo.py -d".
(venv)$ cuckoo -d
```

Now in order to inspect the analyses that have run we start the Web Interface. For small and/or home setups this may be done using the built-in Django web server as follows, although we recommend a proper [Web Deployment](#) for any bigger setup.

```
(venv)$ cuckoo web
Performing system checks...

System check identified no issues (0 silenced).
March 31, 2017 - 12:10:46
Django version 1.8.4, using settings 'cuckoo.web.web.settings'
Starting development server at http://localhost:8000/
Quit the server with CONTROL-C.
```

There are some additional Cuckoo Apps such as `cuckoo clean` ([Clean all Tasks and Samples](#)), the [Cuckoo Router](#), and various other utilities listed in [Cuckoo Apps](#), but other than that there's not much more to learn about installing and running Cuckoo Sandbox - so, happy analyzing.

Submit an Analysis

- *Submission Utility*
- *API*
- *Distributed Cuckoo*
- *Python Functions*

Submission Utility

The easiest way to submit an analysis is to use the `cuckoo submit` utility. It currently has the following options available:

```
$ cuckoo submit --help
Usage: cuckoo submit [OPTIONS] [TARGET]...

    Submit one or more files or URLs to Cuckoo.

Options:
  -u, --url                Submitting URLs instead of samples
  -o, --options TEXT       Options for these tasks
  --package TEXT           Analysis package to use
  --custom TEXT            Custom information to pass along this task
  --owner TEXT             Owner of this task
  --timeout INTEGER        Analysis time in seconds
  --priority INTEGER       Priority of this task
  --machine TEXT           Machine to analyze these tasks on
  --platform TEXT          Analysis platform
  --memory                 Enable memory dumping
  --enforce-timeout        Don't terminate the analysis early
  --clock TEXT             Set the system clock
  --tags TEXT              Analysis tags
  --baseline               Create baseline task
  --remote TEXT            Submit to a remote Cuckoo instance
  --shuffle                Shuffle the submitted tasks
  --pattern TEXT           Provide a glob-pattern when submitting a
                           directory
  --max INTEGER            Submit up to X tasks at once
  --unique                 Only submit samples that have not been
                           analyzed before
  -d, --debug              Enable verbose logging
  -q, --quiet              Only log warnings and critical messages
  --help                  Show this message and exit.
```

You may specify multiple files or directories at once. For directories `cuckoo submit` will enumerate all its files and submit them one by one.

The concept of analysis packages will be dealt later in this documentation (at [Analysis Packages](#)). Following are some usage examples:

Example: submit a local binary:

```
$ cuckoo submit /path/to/binary
```

Example: submit an URL:

```
$ cuckoo submit --url http://www.example.com
```

Example: submit a local binary and specify an higher priority:

```
$ cuckoo submit --priority 5 /path/to/binary
```

Example: submit a local binary and specify a custom analysis timeout of 60 seconds:

```
$ cuckoo submit --timeout 60 /path/to/binary
```

Example: submit a local binary and specify a custom analysis package:

```
$ cuckoo submit --package <name of package> /path/to/binary
```

Example: submit a local binary and specify a custom analysis package and some options (in this case a command line argument for the malware):

```
$ cuckoo submit --package exe --options arguments=--dosomething /path/to/binary.exe
```

Example: submit a local binary to be run on virtual machine *cuckool*:

```
$ cuckoo submit --machine cuckool /path/to/binary
```

Example: submit a local binary to be run on a Windows machine:

```
$ cuckoo submit --platform windows /path/to/binary
```

Example: submit a local binary and take a full memory dump of the analysis machine:

```
$ cuckoo submit --memory /path/to/binary
```

Example: submit a local binary and force the analysis to be executed for the full timeout (disregarding the internal mechanism that Cuckoo uses to decide when to terminate the analysis):

```
$ cuckoo submit --enforce-timeout /path/to/binary
```

Example: submit a local binary and set virtual machine clock. Format is %m-%d-%Y %H:%M:%S. If not specified, the current time is used. For example if we want run a sample the 24 january 2001 at 14:41:20:

```
$ cuckoo submit --clock "01-24-2001 14:41:20" /path/to/binary
```

Example: submit a sample for Volatility analysis (to reduce side effects of the cuckoo hooking, switch it off with *options free=True*):

```
$ cuckoo submit --memory --options free=yes /path/to/binary
```

API

Detailed usage of the REST API interface is described in [REST API](#).

Distributed Cuckoo

Detailed usage of the Distributed Cuckoo API interface is described in [Distributed Cuckoo](#).

Python Functions

In order to keep track of submissions, samples and overall execution, Cuckoo uses a popular Python ORM called [SQLAlchemy](#) that allows you to make the sandbox use SQLite, MySQL or MariaDB, PostgreSQL and several other SQL database systems.

Cuckoo is designed to be easily integrated in larger solutions and to be fully automated. In order to automate analysis submission we suggest to use the REST API interface described in [REST API](#), but in case you want to write your own Python submission script, you can also use the `add_path()` and `add_url()` functions.

add_path (*file_path*[, *timeout=0*[, *package=None*[, *options=None*[, *priority=1*[, *custom=None*[, *owner=""*[, *machine=None*[, *platform=None*[, *tags=None*[, *memory=False*[, *enforce_timeout=False*], *clock=None*]]]]]]]]))

Add a local file to the list of pending analysis tasks. Returns the ID of the newly generated task.

Parameters

- **file_path** (*string*) – path to the file to submit
- **timeout** (*integer*) – maximum amount of seconds to run the analysis for
- **package** (*string or None*) – analysis package you want to use for the specified file
- **options** (*string or None*) – list of options to be passed to the analysis package (in the format `key=value, key=value`)
- **priority** (*integer*) – numeric representation of the priority to assign to the specified file (1 being low, 2 medium, 3 high)
- **custom** (*string or None*) – custom value to be passed over and possibly reused at processing or reporting
- **owner** (*string or None*) – task owner
- **machine** (*string or None*) – Cuckoo identifier of the virtual machine you want to use, if none is specified one will be selected automatically
- **platform** (*string or None*) – operating system platform you want to run the analysis one (currently only Windows)
- **tags** (*string or None*) – tags for machine selection
- **memory** (*True or False*) – set to `True` to generate a full memory dump of the analysis machine
- **enforce_timeout** (*True or False*) – set to `True` to force the execution for the full timeout
- **clock** (*string or None*) – provide a custom clock time to set in the analysis machine

Return type integer

Example usage:

```
1 >>> from cuckoo.core.database import Database
2 >>> db = Database()
3 >>> db.add_path("/tmp/malware.exe")
4 1
5 >>>
```

add_url (*url*[, *timeout=0*[, *package=None*[, *options=None*[, *priority=1*[, *custom=None*[, *owner=""*[, *machine=None*[, *platform=None*[, *tags=None*[, *memory=False*[, *enforce_timeout=False*], *clock=None*]]]]]]]]))

Add a local file to the list of pending analysis tasks. Returns the ID of the newly generated task.

Parameters

- **url** (*string*) – URL to analyze
- **timeout** (*integer*) – maximum amount of seconds to run the analysis for
- **package** (*string or None*) – analysis package you want to use for the specified URL
- **options** (*string or None*) – list of options to be passed to the analysis package (in the format `key=value, key=value`)
- **priority** (*integer*) – numeric representation of the priority to assign to the specified URL (1 being low, 2 medium, 3 high)
- **custom** (*string or None*) – custom value to be passed over and possibly reused at processing or reporting
- **owner** (*string or None*) – task owner
- **machine** (*string or None*) – Cuckoo identifier of the virtual machine you want to use, if none is specified one will be selected automatically
- **platform** (*string or None*) – operating system platform you want to run the analysis one (currently only Windows)
- **tags** (*string or None*) – tags for machine selection
- **memory** (*True or False*) – set to `True` to generate a full memory dump of the analysis machine
- **enforce_timeout** (*True or False*) – set to `True` to force the execution for the full timeout
- **clock** (*string or None*) – provide a custom clock time to set in the analysis machine

Return type integer

Example Usage:

```
1 >>> from cuckoo.core.database import Database
2 >>> db = Database()
3 >>> db.connect()
4 >>> db.add_url("http://www.cuckoosandbox.org")
5 2
6 >>>
```

Web interface

Cuckoo provides a full-fledged web interface in the form of a Django application. This interface will allow you to submit files, browse through the reports, and search across all the analysis results.

Configuration

The web interface pulls data from a Mongo database, so having the Mongo reporting module enabled in `reporting.conf` is mandatory for the Web Interface to function. If that's not the case, the Web Interface won't be able to start and will instead raise an exception.

Some additional configuration options exist in the `$CWD/web/local_settings.py` configuration file.


```

# Copyright (C) 2010-2013 Claudio Guarnieri.
# Copyright (C) 2014-2017 Cuckoo Foundation.
# This file is part of Cuckoo Sandbox - http://www.cuckoosandbox.org
# See the file 'docs/LICENSE' for copying permission.

import web.errors

# Maximum upload size (10GB, so there's basically no limit).
MAX_UPLOAD_SIZE = 10*1024*1024*1024

# Override default secret key stored in $CWD/web/.secret_key
# Make this unique, and don't share it with anybody.
# SECRET_KEY = "YOUR_RANDOM_KEY"

# Language code for this installation. All choices can be found here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = "en-us"

ADMINS = (
    # ("Your Name", "your_email@example.com"),
)

MANAGERS = ADMINS

# Allow verbose debug error message in case of application fault.
# It's strongly suggested to set it to False if you are serving the
# web application from a web server front-end (i.e. Apache).
DEBUG = False
DEBUG404 = False

# A list of strings representing the host/domain names that this Django site
# can serve.
# Values in this list can be fully qualified names (e.g. 'www.example.com').
# When DEBUG is True or when running tests, host validation is disabled; any
# host will be accepted. Thus it's usually only necessary to set it in production.
ALLOWED_HOSTS = ["*"]

handler404 = web.errors.handler404
handler500 = web.errors.handler500

```

It is recommended to keep the `DEBUG` variable at `False` in production setups and to configure at least one `ADMIN` entry to enable error notification by email.

Changed in version 2.0.0: The default maximum upload size has been bumped from 25 MB to 10 GB so that virtually any file should be accepted.

Starting the Web Interface

In order to start the web interface, you can simply run the following command from the `web/` directory:

```
$ cuckoo web runserver
```

If you want to configure the web interface as listening for any IP on a specified port, you can start it with the following command (replace `PORT` with the desired port number):

```
$ cuckoo web runserver 0.0.0.0:PORT
```

Or directly without the `runserver` part as follows while also specifying the host to listen on:

```
$ cuckoo web -H 0
```

Web Deployment

While the default method of starting the Web Interface server works fine for many cases, some users may wish to deploy the server in a more robust manner. This can be done by exposing the Web Interface as a WSGI application to a web server. This section shows a simple example of deploying the Web Interface via `uWSGI` and `nginx`. These instructions are written with Ubuntu GNU/Linux in mind, but may be adapted to other platforms.

This solution requires `uWSGI`, the `uWSGI Python plugin`, and `nginx`. All are available as packages:

```
$ sudo apt-get install uwsgi uwsgi-plugin-python nginx
```

uWSGI setup

First, use `uWSGI` to run the Web Interface server as an application.

To begin, create a `uWSGI` configuration file at `/etc/uwsgi/apps-available/cuckoo-web.ini` that contains the actual configuration as reported by the `cuckoo web --uwsgi` command, e.g.:

```
$ cuckoo web --uwsgi
[uwsgi]
plugins = python
virtualenv = /home/cuckoo/cuckoo
module = cuckoo.web.web.wsgi
uid = cuckoo
gid = cuckoo
static-map = /static=/home/..somepath..
# If you're getting errors about the PYTHON_EGG_CACHE, then
# uncomment the following line and add some path that is
# writable from the defined user.
# env = PYTHON_EGG_CACHE=
env = CUCKOO_APP=web
env = CUCKOO_CWD=/home/..somepath..
```

This configuration inherits a number of settings from the distribution's default `uWSGI` configuration and imports `cuckoo.web.web.wsgi` from the Cuckoo package to do the actual work. In this example we installed Cuckoo in a `virtualenv` located at `/home/cuckoo/cuckoo`. If Cuckoo is installed globally no `virtualenv` option is required (and `cuckoo web --uwsgi` would not report one).

Enable the app configuration and start the server.

```
$ sudo ln -s /etc/uwsgi/apps-available/cuckoo-web.ini /etc/uwsgi/apps-enabled/
$ sudo service uwsgi start cuckoo-web      # or reload, if already running
```

Note: Logs for the application may be found in the standard directory for distribution app instances, i.e., `/var/log/uwsgi/app/cuckoo-web.log`. The UNIX socket is created in a conventional location as well, `/run/uwsgi/app/cuckoo-web/socket`.

nginx setup

With the Web Interface server running in uWSGI, nginx can now be set up to run as a web server/reverse proxy, backending HTTP requests to it.

To begin, create a nginx configuration file at `/etc/nginx/sites-available/cuckoo-web` that contains the actual configuration as reported by the `cuckoo web --nginx` command:

```
$ cuckoo web --nginx
upstream _uwsgi_cuckoo_web {
    server unix:/run/uwsgi/app/cuckoo-web/socket;
}

server {
    listen localhost:8000;

    # Cuckoo Web Interface
    location / {
        client_max_body_size 1G;
        uwsgi_pass _uwsgi_cuckoo_web;
        include uwsgi_params;
    }
}
```

Make sure that nginx can connect to the uWSGI socket by placing its user in the **cuckoo** group:

```
$ sudo adduser www-data cuckoo
```

Enable the server configuration and start the server.

```
$ sudo ln -s /etc/nginx/sites-available/cuckoo-web /etc/nginx/sites-enabled/
$ sudo service nginx start    # or reload, if already running
```

At this point, the Web Interface server should be available at port **8000** on the server. Various configurations may be applied to extend this configuration, such as to tune server performance, add authentication, or to secure communications using HTTPS. However, we leave this as an exercise for the user.

REST API

As mentioned in [Submit an Analysis](#), Cuckoo provides a simple and lightweight REST API server that is under the hood implemented using [Flask](#).

Starting the API server

In order to start the API server you can simply do:

```
$ cuckoo api
```

By default it will bind the service on **localhost:8090**. If you want to change those values, you can use the following syntax:

```
$ cuckoo api --host 0.0.0.0 --port 1337
$ cuckoo api -H 0.0.0.0 -p 1337
```

Web deployment

While the default method of starting the API server works fine for many cases, some users may wish to deploy the server in a robust manner. This can be done by exposing the API as a WSGI application through a web server. This section shows a simple example of deploying the API via **uWSGI** and **nginx**. These instructions are written with Ubuntu GNU/Linux in mind, but may be adapted for other platforms.

This solution requires uWSGI, the uWSGI Python plugin, and nginx. All are available as packages:

```
$ sudo apt-get install uwsgi uwsgi-plugin-python nginx
```

uWSGI setup

First, use uWSGI to run the API server as an application.

To begin, create a uWSGI configuration file at `/etc/uwsgi/apps-available/cuckoo-api.ini` that contains the actual configuration as reported by the `cuckoo api --uwsgi` command:

```
$ cuckoo api --uwsgi
[uwsgi]
plugins = python
virtualenv = /home/cuckoo/cuckoo
module = cuckoo.apps.api
callable = app
uid = cuckoo
gid = cuckoo
env = CUCKOO_APP=api
env = CUCKOO_CWD=/home/..somepath..
```

This configuration inherits a number of settings from the distribution's default uWSGI configuration and imports `cuckoo.apps.api` from the Cuckoo package to do the actual work. In this example we installed Cuckoo in a virtualenv located at `/home/cuckoo/cuckoo`. If Cuckoo is installed globally no virtualenv option is required.

Enable the app configuration and start the server.

```
$ sudo ln -s /etc/uwsgi/apps-available/cuckoo-api.ini /etc/uwsgi/apps-enabled/
$ sudo service uwsgi start cuckoo-api      # or reload, if already running
```

Note: Logs for the application may be found in the standard directory for distribution app instances, i.e., `/var/log/uwsgi/app/cuckoo-api.log`. The UNIX socket is created in a conventional location as well, `/run/uwsgi/app/cuckoo-api/socket`.

nginx setup

With the API server running in uWSGI, nginx can now be set up to run as a web server/reverse proxy, backending HTTP requests to it.

To begin, create a nginx configuration file at `/etc/nginx/sites-available/cuckoo-api` that contains the actual configuration as reported by the `cuckoo api --nginx` command:

```
$ cuckoo api --nginx
upstream _uwsgi_cuckoo_api {
    server unix:/run/uwsgi/app/cuckoo-api/socket;
```

```
}  
  
server {  
    listen localhost:8090;  
  
    # REST API app  
    location / {  
        client_max_body_size 1G;  
        uwsgi_pass _uwsgi_cuckoo_api;  
        include uwsgi_params;  
    }  
}
```

Make sure that nginx can connect to the uWSGI socket by placing its user in the **cuckoo** group:

```
$ sudo adduser www-data cuckoo
```

Enable the server configuration and start the server.

```
$ sudo ln -s /etc/nginx/sites-available/cuckoo-api /etc/nginx/sites-enabled/  
$ sudo service nginx start      # or reload, if already running
```

At this point, the API server should be available at port **8090** on the server. Various configurations may be applied to extend this configuration, such as to tune server performance, add authentication, or to secure communications using HTTPS.

Resources

Following is a list of currently available resources and a brief description of each one. For details click on the resource name.

Resource	Description
POST /tasks/create/file	Adds a file to the list of pending tasks to be processed and analyzed.
POST /tasks/create/url	Adds an URL to the list of pending tasks to be processed and analyzed.
POST /tasks/create/submit	Adds one or more files and/or files embedded in archives to the list of pending tasks.
GET /tasks/list	Returns the list of tasks stored in the internal Cuckoo database. You can optionally specify a limit of entries to return.
GET /tasks/view	Returns the details on the task assigned to the specified ID.
GET /tasks/reschedule	Reschedule a task assigned to the specified ID.
GET /tasks/delete	Removes the given task from the database and deletes the results.
GET /tasks/report	Returns the report generated out of the analysis of the task associated with the specified ID. You can optionally specify which report format to return, if none is specified the JSON report will be returned.
GET /tasks/screenshots	Retrieves one or all screenshots associated with a given analysis task ID.
GET /tasks/rereport	Re-run reporting for task associated with a given analysis task ID.
GET /tasks/reboot	Reboot a given analysis task ID.
GET /memory/list	Returns a list of memory dump files associated with a given analysis task ID.
GET /memory/get	Retrieves one memory dump file associated with a given analysis task ID.
GET /files/view	Search the analyzed binaries by MD5 hash, SHA256 hash or internal ID (referenced by the tasks details).
GET /files/get	Returns the content of the binary with the specified SHA256 hash.
GET /pcap/get	Returns the content of the PCAP associated with the given task.
GET /machines/list	Returns the list of analysis machines available to Cuckoo.
GET /machines/view	Returns details on the analysis machine associated with the specified name.
GET /cuckoo/status	Returns the basic cuckoo status, including version and tasks overview.
GET /vpn/status	Returns VPN status.
GET /exit	Shuts down the API server.

[/tasks/create/file](#)

POST [/tasks/create/file](#)

Adds a file to the list of pending tasks. Returns the ID of the newly created task.

Example request:

```
curl -F file=@/path/to/file http://localhost:8090/tasks/create/file
```

Example request using Python..

```
import requests

REST_URL = "http://localhost:8090/tasks/create/file"
SAMPLE_FILE = "/path/to/malwr.exe"

with open(SAMPLE_FILE, "rb") as sample:
    files = {"file": ("temp_file_name", sample)}
    r = requests.post(REST_URL, files=files)

# Add your code to error checking for r.status_code.

task_id = r.json()["task_id"]

# Add your code for error checking if task_id is None.
```

Example response.

```
{
  "task_id" : 1
}
```

Form parameters:

- `file` (*required*) - sample file (multipart encoded file content)
- `package` (*optional*) - analysis package to be used for the analysis
- `timeout` (*optional*) (*int*) - analysis timeout (in seconds)
- `priority` (*optional*) (*int*) - priority to assign to the task (1-3)
- `options` (*optional*) - options to pass to the analysis package
- `machine` (*optional*) - label of the analysis machine to use for the analysis
- `platform` (*optional*) - name of the platform to select the analysis machine from (e.g. "windows")
- `tags` (*optional*) - define machine to start by tags. Platform must be set to use that. Tags are comma separated
- `custom` (*optional*) - custom string to pass over the analysis and the processing/reporting modules
- `owner` (*optional*) - task owner in case multiple users can submit files to the same cuckoo instance
- `clock` (*optional*) - set virtual machine clock (format %m-%d-%Y %H:%M:%S)
- `memory` (*optional*) - enable the creation of a full memory dump of the analysis machine
- `unique` (*optional*) - only submit samples that have not been analyzed before
- `enforce_timeout` (*optional*) - enable to enforce the execution for the full timeout value

Status codes:

- 200 - no error
- 400 - duplicated file detected (when using unique option)

/tasks/create/url**POST /tasks/create/url**

Adds a file to the list of pending tasks. Returns the ID of the newly created task.

Example request.

```
curl -F url="http://www.malicious.site" http://localhost:8090/tasks/create/url
```

Example request using Python.

```
import requests

REST_URL = "http://localhost:8090/tasks/create/url"
SAMPLE_URL = "http://example.org/malwr.exe"

data = {"url": SAMPLE_URL}
r = requests.post(REST_URL, data=data)

# Add your code to error checking for r.status_code.

task_id = r.json()["task_id"]

# Add your code to error checking if task_id is None.
```

Example response.

```
{
  "task_id" : 1
}
```

Form parameters:

- `url` (*required*) - URL to analyze (multipart encoded content)
- `package` (*optional*) - analysis package to be used for the analysis
- `timeout` (*optional*) (*int*) - analysis timeout (in seconds)
- `priority` (*optional*) (*int*) - priority to assign to the task (1-3)
- `options` (*optional*) - options to pass to the analysis package
- `machine` (*optional*) - label of the analysis machine to use for the analysis
- `platform` (*optional*) - name of the platform to select the analysis machine from (e.g. "windows")
- `tags` (*optional*) - define machine to start by tags. Platform must be set to use that. Tags are comma separated
- `custom` (*optional*) - custom string to pass over the analysis and the processing/reporting modules
- `owner` (*optional*) - task owner in case multiple users can submit files to the same cuckoo instance
- `memory` (*optional*) - enable the creation of a full memory dump of the analysis machine
- `enforce_timeout` (*optional*) - enable to enforce the execution for the full timeout value
- `clock` (*optional*) - set virtual machine clock (format %m-%d-%Y %H:%M:%S)

Status codes:

- 200 - no error

`/tasks/create/submit`**POST `/tasks/create/submit`**

Adds one or more files and/or files embedded in archives to the list of pending tasks. Returns the submit ID as well as the task IDs of the newly created task(s).

Example request.

```
curl http://localhost:8090/tasks/create/submit -F files=@1.exe -F files=@2.exe
```

Example request using Python.

```
import requests

r = requests.post("http://localhost:8090/tasks/create/submit", files=[
    ("files", open("1.exe", "rb")),
    ("files", open("2.exe", "rb")),
])

# Add your code to error checking for r.status_code.

submit_id = r.json()["submit_id"]
task_ids = r.json()["task_ids"]
errors = r.json()["errors"]

# Add your code to error checking on "errors".
```

Example response.

```
{
  "submit_id": 1,
  "task_ids": [1, 2],
  "errors": []
}
```

Form parameters:

- *file (optional)* - backwards compatibility with naming scheme for */tasks/create/file*
- *files (required)* - sample(s) to inspect and add to our pending queue
- *timeout (optional) (int)* - analysis timeout (in seconds)
- *priority (optional) (int)* - priority to assign to the task (1-3)
- *options (optional)* - options to pass to the analysis package
- *tags (optional)* - define machine to start by tags. Platform must be set to use that. Tags are comma separated
- *custom (optional)* - custom string to pass over the analysis and the processing/reporting modules
- *owner (optional)* - task owner in case multiple users can submit files to the same cuckoo instance
- *memory (optional)* - enable the creation of a full memory dump of the analysis machine
- *enforce_timeout (optional)* - enable to enforce the execution for the full timeout value
- *clock (optional)* - set virtual machine clock (format %m-%d-%Y %H:%M:%S)

Status codes:

- 200 - no error

/tasks/list

GET /tasks/list/ (*int: limit*) / (*int: offset*)

Returns list of tasks.

Example request.

```
curl http://localhost:8090/tasks/list
```

Example response.

```
{
  "tasks": [
    {
      "category": "url",
      "machine": null,
      "errors": [],
      "target": "http://www.malicious.site",
      "package": null,
      "sample_id": null,
      "guest": {},
      "custom": null,
      "owner": "",
      "priority": 1,
      "platform": null,
      "options": null,
      "status": "pending",
      "enforce_timeout": false,
      "timeout": 0,
      "memory": false,
      "tags": []
      "id": 1,
      "added_on": "2012-12-19 14:18:25",
      "completed_on": null
    },
    {
      "category": "file",
      "machine": null,
      "errors": [],
      "target": "/tmp/malware.exe",
      "package": null,
      "sample_id": 1,
      "guest": {},
      "custom": null,
      "owner": "",
      "priority": 1,
      "platform": null,
      "options": null,
      "status": "pending",
      "enforce_timeout": false,
      "timeout": 0,
      "memory": false,
      "tags": [
        "32bit",
        "acrobat_6",
      ],
      "id": 2,
      "added_on": "2012-12-19 14:18:25",
```

```

        "completed_on": null
    }
]
}

```

Parameters:

- `limit` (*optional*) (*int*) - maximum number of returned tasks
- `offset` (*optional*) (*int*) - data offset

Status codes:

- 200 - no error

/tasks/view**GET /tasks/view/** (*int: id*)

Returns details on the task associated with the specified ID.

Example request.

```
curl http://localhost:8090/tasks/view/1
```

Example response.

```

{
  "task": {
    "category": "url",
    "machine": null,
    "errors": [],
    "target": "http://www.malicious.site",
    "package": null,
    "sample_id": null,
    "guest": {},
    "custom": null,
    "owner": "",
    "priority": 1,
    "platform": null,
    "options": null,
    "status": "pending",
    "enforce_timeout": false,
    "timeout": 0,
    "memory": false,
    "tags": [
      "32bit",
      "acrobat_6",
    ],
    "id": 1,
    "added_on": "2012-12-19 14:18:25",
    "completed_on": null
  }
}

```

Note: possible value for key `status`:

- `pending`

- running
- completed
- reported

Parameters:

- `id` (*required*) (*int*) - ID of the task to lookup

Status codes:

- 200 - no error
- 404 - task not found

/tasks/reschedule

GET /tasks/reschedule/ (*int: id*) / (*int: priority*)

Reschedule a task with the specified ID and priority (default priority is 1).

Example request.

```
curl http://localhost:8090/tasks/reschedule/1
```

Example response.

```
{
  "status": "OK"
}
```

Parameters:

- `id` (*required*) (*int*) - ID of the task to reschedule
- `priority` (*optional*) (*int*) - Task priority

Status codes:

- 200 - no error
- 404 - task not found

/tasks/delete

GET /tasks/delete/ (*int: id*)

Removes the given task from the database and deletes the results.

Example request.

```
curl http://localhost:8090/tasks/delete/1
```

Parameters:

- `id` (*required*) (*int*) - ID of the task to delete

Status codes:

- 200 - no error
- 404 - task not found

- 500 - unable to delete the task

/tasks/report

GET /tasks/report/ (*int: id*) / (*str: format*)

Returns the report associated with the specified task ID.

Example request.

```
curl http://localhost:8090/tasks/report/1
```

Parameters:

- *id* (*required*) (*int*) - ID of the task to get the report for
- *format* (*optional*) - format of the report to retrieve [json/html/all/dropped/package_files]. If none is specified the JSON report will be returned. *all* returns all the result files as tar.bz2, *dropped* the dropped files as tar.bz2, *package_files* files uploaded to host by analysis packages.

Status codes:

- 200 - no error
- 400 - invalid report format
- 404 - report not found

/tasks/screenshots

GET /tasks/screenshots/ (*int: id*) / (*str: number*)

Returns one or all screenshots associated with the specified task ID.

Example request.

```
wget http://localhost:8090/tasks/screenshots/1
```

Parameters:

- *id* (*required*) (*int*) - ID of the task to get the report for
- *screenshot* (*optional*) - numerical identifier of a single screenshot (e.g. 0001, 0002)

Status codes:

- 404 - file or folder not found

/tasks/rereport

GET /tasks/rereport/ (*int: id*)

Re-run reporting for task associated with the specified task ID.

Example request.

```
curl http://localhost:8090/tasks/rereport/1
```

Example response.

```
{
  "success": true
}
```

Parameters:

- `id` (*required*) (*int*) - ID of the task to re-run report

Status codes:

- 200 - no error
- 404 - task not found

/tasks/reboot

GET /tasks/reboot/ (*int: id*) **

Add a reboot task to database from an existing analysis ID.

Example request.

```
curl http://localhost:8090/tasks/reboot/1
```

Example response.

```
{
  "task_id": 1,
  "reboot_id": 3
}
```

Parameters:

- `id` (*required*) (*int*) - ID of the task

Status codes:

- 200 - success
- 404 - error creating reboot task

/memory/list

GET /memory/list/ (*int: id*)

Returns a list of memory dump files or one memory dump file associated with the specified task ID.

Example request.

```
wget http://localhost:8090/memory/list/1
```

Parameters:

- `id` (*required*) (*int*) - ID of the task to get the report for

Status codes:

- 404 - file or folder not found

/memory/get

GET /memory/get/ (*int: id*) / (*str: number*)

Returns one memory dump file associated with the specified task ID.

Example request.

```
wget http://localhost:8090/memory/get/1/1908
```

Parameters:

- *id* (*required*) (*int*) - ID of the task to get the report for
- *pid* (*required*) - numerical identifier (pid) of a single memory dump file (e.g. 205, 1908)

Status codes:

- 404 - file or folder not found

/files/view

GET /files/view/md5/ (*str: md5*)

GET /files/view/sha256/ (*str: sha256*)

GET /files/view/id/ (*int: id*)

Returns details on the file matching either the specified MD5 hash, SHA256 hash or ID.

Example request.

```
curl http://localhost:8090/files/view/id/1
```

Example response.

```
{
  "sample": {
    "sha1": "da39a3ee5e6b4b0d3255bfef95601890afd80709",
    "file_type": "empty",
    "file_size": 0,
    "crc32": "00000000",
    "ssdeep": "3::",
    "sha256": "e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855",
    "sha512":
    ↪ "cf83e1357ee5fb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d13c5d85f2b0ff8318d2877eec2f63
    ↪ ",
    "id": 1,
    "md5": "d41d8cd98f00b204e9800998ecf8427e"
  }
}
```

Parameters:

- *md5* (*optional*) - MD5 hash of the file to lookup
- *sha256* (*optional*) - SHA256 hash of the file to lookup
- *id* (*optional*) (*int*) - ID of the file to lookup

Status codes:

- 200 - no error
- 400 - invalid lookup term
- 404 - file not found

/files/get

GET /files/get/ (*str: sha256*)

Returns the binary content of the file matching the specified SHA256 hash.

Example request.

```
curl http://localhost:8090/files/get/  
↪e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855 > sample.exe
```

Status codes:

- 200 - no error
- 404 - file not found

/pcap/get

GET /pcap/get/ (*int: task*)

Returns the content of the PCAP associated with the given task.

Example request.

```
curl http://localhost:8090/pcap/get/1 > dump.pcap
```

Status codes:

- 200 - no error
- 404 - file not found

/machines/list

GET /machines/list

Returns a list with details on the analysis machines available to Cuckoo.

Example request.

```
curl http://localhost:8090/machines/list
```

Example response.

```
{  
  "machines": [  
    {  
      "status": null,  
      "locked": false,  
      "name": "cuckoo1",  
      "resultserver_ip": "192.168.56.1",  
    }  
  ]  
}
```



```

    "ip": "192.168.56.101",
    "tags": [
        "32bit",
        "acrobat_6",
    ],
    "label": "cuckoo1",
    "locked_changed_on": null,
    "platform": "windows",
    "snapshot": null,
    "interface": null,
    "status_changed_on": null,
    "id": 1,
    "resultserver_port": "2042"
  }
]
}

```

Status codes:

- 200 - no error

/machines/view**GET /machines/view/** (*str: name*)

Returns details on the analysis machine associated with the given name.

Example request.

```
curl http://localhost:8090/machines/view/cuckoo1
```

Example response.

```

{
  "machine": {
    "status": null,
    "locked": false,
    "name": "cuckoo1",
    "resultserver_ip": "192.168.56.1",
    "ip": "192.168.56.101",
    "tags": [
        "32bit",
        "acrobat_6",
    ],
    "label": "cuckoo1",
    "locked_changed_on": null,
    "platform": "windows",
    "snapshot": null,
    "interface": null,
    "status_changed_on": null,
    "id": 1,
    "resultserver_port": "2042"
  }
}

```

Status codes:

- 200 - no error

- 404 - machine not found

/cuckoo/status

GET /cuckoo/status/

Returns status of the cuckoo server. In version 1.3 the diskpace entry was added. The diskpace entry shows the used, free, and total diskpace at the disk where the respective directories can be found. The diskpace entry allows monitoring of a Cuckoo node through the Cuckoo API. Note that each directory is checked separately as one may create a symlink for \$CUCKOO/storage/analyses to a separate hddisk, but keep \$CUCKOO/storage/binaries as-is. (This feature is only available under Unix!)

In version 1.3 the cpuload entry was also added - the cpuload entry shows the CPU load for the past minute, the past 5 minutes, and the past 15 minutes, respectively. (This feature is only available under Unix!)

Diskspace directories:

- analyses - \$CUCKOO/storage/analyses/
- binaries - \$CUCKOO/storage/binaries/
- temporary - tmp path as specified in conf/cuckoo.conf

Example request.

```
curl http://localhost:8090/cuckoo/status
```

Example response.

```
{
  "tasks": {
    "reported": 165,
    "running": 2,
    "total": 167,
    "completed": 0,
    "pending": 0
  },
  "diskspace": {
    "analyses": {
      "total": 491271233536,
      "free": 71403470848,
      "used": 419867762688
    },
    "binaries": {
      "total": 491271233536,
      "free": 71403470848,
      "used": 419867762688
    },
    "temporary": {
      "total": 491271233536,
      "free": 71403470848,
      "used": 419867762688
    }
  },
  "version": "1.0",
  "protocol_version": 1,
  "hostname": "Patient0",
  "machines": {
    "available": 4,

```

```

    "total": 5
  }
}

```

Status codes:

- 200 - no error
- 404 - machine not found

/vpn/status**GET /vpn/status**

Returns VPN status.

Example request.

```
curl http://localhost:8090/vpn/status
```

Status codes:

- 200 - show status
- 500 - not available

/exit**GET /exit**

Shuts down the server if in debug mode and using the werkzeug server.

Example request.

```
curl http://localhost:8090/exit
```

Status codes:

- 200 - success
- 403 - this call can only be used in debug mode
- 500 - error

Distributed Cuckoo

As mentioned in *Submit an Analysis*, Cuckoo provides a REST API for Distributed Cuckoo usage. The distributed script allows one to setup a single REST API point to which samples and URLs can be submitted which will then, in turn, be submitted to one of the configured Cuckoo nodes.

A typical setup thus includes a machine on which Distributed Cuckoo is run and one or more machines running an instance of the Cuckoo daemon and the *Cuckoo REST API*.

A few notes;

- Using the distributed script only makes sense when running at least two cuckoo nodes.
- The distributed script can be run on a machine that also runs a Cuckoo daemon and REST API, however, make sure it has enough disk space if the intention is to submit a lot of samples.

Starting the Distributed REST API

The Distributed REST API has the following command line options:

```
$ cuckoo distributed server --help
Usage: cuckoo distributed server [OPTIONS]

Options:
  -H, --host TEXT      Host to bind the Distributed Cuckoo server on
  -p, --port INTEGER   Port to bind the Distributed Cuckoo server on
  --uwsgi              Dump uWSGI configuration
  --nginx              Dump nginx configuration
  --help              Show this message and exit.
```

As may be derived from the help output, starting Distributed Cuckoo may be as simple as running `cuckoo distributed server`.

The various configuration options are described in the configuration file, but following we have more in-depth descriptions as well. More advanced usage naturally includes deploying to `uWSGI` and `nginx`.

Distributed Cuckoo Configuration

Report Formats

The reporting formats denote which reports you'd like to retrieve later on. Note that all task-related data will be removed from the Cuckoo nodes once the related reports have been fetched so that the machines are not running out of disk space. This does, however, force you to specify all the report formats that you're interested in, because otherwise that information will be lost.

Reporting formats include, but are not limited to and may also include your own reporting formats, `report.json`, `report.html`, etc.

Samples Directory

The samples directory denotes the directory where the submitted samples will be stored temporarily, until the associated task has been deleted.

Reports Directory

Much like the `Samples Directory` the `Reports Directory` defines the directory where reports will be stored until they're fetched and deleted from the Distributed REST API.

RESTful resources

Following are all RESTful resources. Also make sure to check out the [Quick usage](#) section which documents the most commonly used commands.

Resource	Description
GET <i>GET /api/node</i>	Get a list of all enabled Cuckoo nodes.
POST <i>POST /api/node</i>	Register a new Cuckoo node.
GET <i>GET /api/node/<name></i>	Get basic information about a node.
PUT <i>PUT /api/node/<name></i>	Update basic information of a node.
DELETE <i>DELETE /api/node/<name></i>	Disable (not completely remove!) a node.
GET <i>GET /api/task</i>	Get a list of all (or a part) of the tasks in the database.
POST <i>POST /api/task</i>	Create a new analysis task.
GET <i>GET /api/task/<id></i>	Get basic information about a task.
DELETE <i>DELETE /api/task/<id></i>	Delete all associated information of a task.
GET <i>GET /api/report/<id>/<format></i>	Fetch an analysis report.

GET /api/node

Returns all enabled nodes. For each node the information includes the associated name, its API URL, and machines:

```
$ curl http://localhost:9003/api/node
{
  "success": true,
  "nodes": {
    "localhost": {
      "machines": [
        {
          "name": "cuckool",
          "platform": "windows",
          "tags": []
        }
      ],
      "name": "localhost",
      "url": "http://localhost:8090/"
    }
  }
}
```

POST /api/node

Register a new Cuckoo node by providing the name and the URL:

```
$ curl http://localhost:9003/api/node -F name=localhost \
-F url=http://localhost:8090/
{
  "success": true
}
```

GET /api/node/<name>

Get basic information about a particular Cuckoo node:

```
$ curl http://localhost:9003/api/node/localhost
{
  "success": true,
```

```
{
  "nodes": [
    {
      "name": "localhost",
      "url": "http://localhost:8090/"
      "machines": [
        {
          "name": "cuckool",
          "platform": "windows",
          "tags": []
        }
      ]
    }
  ]
}
```

PUT /api/node/<name>

Update basic information of a Cuckoo node:

```
$ curl -XPUT http://localhost:9003/api/node/localhost -F name=newhost \
-F url=http://1.2.3.4:8090/
{
  "success": true
}
```

DELETE /api/node/<name>

Disable a Cuckoo node, therefore not having it process any new tasks, but keeping its history in the Distributed Cuckoo database:

```
$ curl -XDELETE http://localhost:9003/api/node/localhost
{
  "success": true
}
```

GET /api/task

Get a list of all tasks in the database. In order to limit the amount of results, there's an `offset`, `limit`, `finished`, and `owner` field available:

```
$ curl http://localhost:9003/api/task?limit=1
{
  "success": true,
  "tasks": {
    "1": {
      "clock": null,
      "custom": null,
      "owner": "",
      "enforce_timeout": null,
      "machine": null,
      "memory": null,
      "options": null,

```

```

        "package": null,
        "path": "/tmp/dist-samples/tmphal8mS",
        "platform": "windows",
        "priority": 1,
        "tags": null,
        "task_id": 1,
        "timeout": null
    }
}

```

POST /api/task

Submit a new file or URL to be analyzed:

```

$ curl http://localhost:9003/api/task -F file=@sample.exe
{
    "success": true,
    "task_id": 2
}

```

GET /api/task/<id>

Get basic information about a particular task:

```

$ curl http://localhost:9003/api/task/2
{
    "success": true,
    "tasks": {
        "2": {
            "id": 2,
            "clock": null,
            "custom": null,
            "owner": "",
            "enforce_timeout": null,
            "machine": null,
            "memory": null,
            "options": null,
            "package": null,
            "path": "/tmp/tmpPwUeXm",
            "platform": "windows",
            "priority": 1,
            "tags": null,
            "timeout": null,
            "task_id": 1,
            "node_id": 2,
            "finished": false
        }
    }
}

```

DELETE /api/task/<id>

Delete all associated data of a task, namely the binary, the PCAP, and the reports:

```
$ curl -XDELETE http://localhost:9003/api/task/2
{
  "success": true
}
```

GET /api/report/<id>/<format>

Fetch a report for the given task in the specified format:

```
# Defaults to the JSON report.
$ curl http://localhost:9003/api/report/2
...
```

Quick usage

For practical usage the following few commands will be most interesting.

Register a Cuckoo node - a Cuckoo API running on the same machine in this case:

```
$ curl http://localhost:9003/api/node -F name=localhost -F ip=127.0.0.1
```

Disable a Cuckoo node:

```
$ curl -XDELETE http://localhost:9003/api/node/localhost
```

Submit a new analysis task without any special requirements (e.g., using Cuckoo tags, a particular machine, etc):

```
$ curl http://localhost:9003/api/task -F file=@/path/to/sample.exe
```

Get the report of a task has been finished (if it hasn't finished you'll get an error with code 420). Following example will default to the JSON report:

```
$ curl http://localhost:9003/api/report/1
```

Proposed setup

The following description depicts a Distributed Cuckoo setup with two Cuckoo machines, **cuckoo0** and **cuckoo1**. In this setup the first machine, cuckoo0, also hosts the Distributed Cuckoo REST API.

Configuration settings

Our setup will require a couple of updates with regards to the configuration files.

conf/cuckoo.conf

Update `process_results` to `off` as we will be running our own results processing script (for performance reasons).

Update `tmppath` to something that holds enough storage to store a few hundred binaries. On some servers or setups `/tmp` may have a limited amount of space and thus this wouldn't suffice.

Update `connection` to use something *not* `sqlite3`. Preferably PostgreSQL or MySQL. SQLite3 doesn't support multi-threaded applications and as such is not a good choice for systems such as Cuckoo (as-is).

You should create a database specifically for the distributed cuckoo setup. Do not be tempted to use any existing cuckoo database in order to avoid update problems with the DB scripts. In the configuration use the new database name. The remaining configuration such as usernames, servers, etc can be the same as for your cuckoo install. Don't forget to use one DB per node and one for the machine running Distributed Cuckoo (the "management machine" or "controller").

conf/processing.conf

You may want to disable some processing modules, such as `virustotal`.

conf/reporting.conf

Depending on which report(s) are required for integration with your system it might make sense to only make those report(s) that you're going to use. Thus disabling the other ones.

conf/virtualbox.conf

Assuming `VirtualBox` is the Virtual Machine manager of choice, the `mode` will have to be changed to `headless` or you will have some restless nights (this is the default nowadays).

Setup Cuckoo

On each machine you will have to run the Cuckoo Daemon, the Cuckoo API, and one or more Cuckoo Process instances. For more information on setting that up, please refer to [Starting Cuckoo](#).

Setup Distributed Cuckoo

On the Distributed Cuckoo machine you'll have to setup the Distributed Cuckoo REST API and the Distributed Cuckoo Worker.

As stated earlier, Distributed Cuckoo REST API may be started by running `cuckoo distributed server` or by deploying it properly with `uWSGI` and `nginx`.

The Distributed Cuckoo Worker may be started by running `supervisorctl start distributed` in the CWD. This will automatically start the Worker with the correct configuration and arguments, etc.

Register Cuckoo nodes

As outlined in *Quick usage* the Cuckoo nodes have to be registered with the Distributed Cuckoo REST API:

```
$ curl http://localhost:9003/api/node -F name=cuckoo0 -F url=http://localhost:8090/
$ curl http://localhost:9003/api/node -F name=cuckool -F url=http://1.2.3.4:8090/
```

Having registered the Cuckoo nodes all that's left to do now is to submit tasks and fetch reports once finished. Documentation on these commands can be found in the *Quick usage* section. In case your Cuckoo node is not on localhost, replace localhost with the IP address of the node where the Cuckoo REST API is running.

If you want to experiment a real load balancing between the nodes you may want to try using a lower value for the threshold parameter in the `$CWD/distributed/settings.py` file as the default value is 500 (meaning tasks are assigned to Cuckoo nodes in batches of 500).

Utilities

Cuckoo comes with a set of pre-built utilities to automate several common tasks. Before these utilities could be found in the `utils/` directory but since then we have moved to Cuckoo Apps.

Cuckoo Apps

A Cuckoo App is essentially just a Cuckoo sub-command. There exist a couple of Cuckoo Apps, each with their own functionality. It is important to note that each Cuckoo App can be invoked in the same way. Following are some examples:

```
$ cuckoo submit --help
$ cuckoo api --help
$ cuckoo clean --help
```

In these examples we provided the `--help` parameter which shows the functionality and all available parameters for the particular Cuckoo App.

Submission Utility

Submits samples to analysis. This tool is described in *Submit an Analysis*.

Web Utility

Cuckoo's web interface. This tool is described in *Web interface*.

Processing Utility

Changed in version 2.0.0: We used to have longstanding issues with `./utils/process.py` randomly freezing up and `./utils/process2.py` only being able to handle PostgreSQL-based databases. These two commands have now been merged into one Cuckoo App and no longer show signs of said issues or limitations.

For bigger Cuckoo setups it is recommended to separate the results processing from the Cuckoo analyses due to performance issues (with multiple threads & the *Python GIL*). Using `cuckoo process` it is also possible to regenerate Cuckoo reports, this is mostly used while developing and debugging Cuckoo Processing modules, Cuckoo Signatures, and Cuckoo Reporting modules.

In order to do results processing in one or more separate process(es) one has to disable the `process_results` configuration item in `$CWD/conf/cuckoo.conf` by setting the value to `off`. Then a Cuckoo Processing instance has to be started, this can be done as follows:

```
$ cuckoo process instance1
```

If one Cuckoo Processing instance is not enough to handle all the incoming analyses, simply create a second, third, and possibly more instances:

```
$ cuckoo process instance2
```

In order to re-generate a Cuckoo report of an analysis task, use the `-r` switch:

```
$ cuckoo process -r 1
```

It is also possible to re-generate multiple or a range of Cuckoo reports at once. The following will reprocess tasks 1, 2, 5, 6, 7, 8, 9, 10:

```
$ cuckoo process -r 1,2,5-10
```

For more information see also the help on this Cuckoo App:

```
$ cuckoo process --help
Usage: cuckoo process [OPTIONS] [INSTANCE]

    Process raw task data into reports.

Options:
  -r, --report TEXT          Re-generate one or more reports
  -m, --maxcount INTEGER    Maximum number of analyses to process
  --help                    Show this message and exit.
```

In automated mode an instance name is required (e.g., `instance1`) as seen in the examples earlier above!

Community Download Utility

This Cuckoo App downloads Cuckoo Signatures, the latest monitoring binaries, and other goodies from the [Cuckoo Community Repository](#) and installs them in your CWD.

To get all the latest and greatest from the Cuckoo Community simply execute as follows and wait until it finishes - it currently doesn't have any progress indication:

```
$ cuckoo community
```

For more usage see as follows:

```
$ cuckoo community --help
Usage: cuckoo community [OPTIONS]

    Utility to fetch supplies from the Cuckoo Community.

Options:
  -f, --force                Overwrite existing files
  -b, --branch TEXT          Specify a different community branch rather than
                             master
  --file, --filepath PATH    Specify a local copy of a community .tar.gz file
  --help                    Show this message and exit.
```

Database migration utility

Changed in version 2.0.0: This used to be a special process, but has since been integrated properly as a Cuckoo App.

This utility helps migrating your data between Cuckoo releases. It's developed on top of the [Alembic](#) framework and it should provide data migration for both SQL database and Mongo database. This tool is already described in [Upgrading from a previous release](#).

Stats utility

Deprecated since version 2.0-rc2: This utility will not be ported to a Cuckoo App as this information can also be retrieved through both the Cuckoo API as well as the Cuckoo Web Interface.

Machine utility

Changed in version 2.0.0: This used to be a standalone and hacky script directly modifying the Cuckoo configuration. It's now much better integrated and will be able to somewhat properly interact with Cuckoo.

The machine Cuckoo App is designed to help you automatize the configuration of virtual machines in Cuckoo. It takes a list of machine details as arguments and write them in the specified configuration file of the machinery module enabled in *cuckoo.conf*. Following are the available options:

```
$ cuckoo machine --help
Usage: cuckoo machine [OPTIONS] VMNAME [IP]

Options:
  --debug          Enable verbose logging
  --add            Add a Virtual Machine
  --delete         Delete a Virtual Machine
  --platform TEXT  Guest Operating System
  --options TEXT   Machine options
  --tags TEXT      Tags for this Virtual Machine
  --interface TEXT Sniffer interface for this Virtual Machine
  --snapshot TEXT  Specific Virtual Machine Snapshot to use
  --resultserver TEXT IP:Port of the Result Server
  --help          Show this message and exit.
```

As an example, a machine may be added to Cuckoo's configuration as follows:

```
$ cuckoo machine --add cuckoo1 192.168.56.101 --platform windows --snapshot vmcloak
```

Distributed scripts

This tool is described in [Distributed Cuckoo](#).

Mac OS X Bootstrap scripts

Deprecated since version 2.0.0: These files will be moved elsewhere in an upcoming update and so should any documentation that references these scripts.

A couple of bootstrap scripts used for Mac OS X analysis are located in *utils/darwin* folder, they are used to bootstrap the guest and host system for Mac OS X malware analysis. Some settings are defined as constants inside them, so it is suggested to have a look at them and configure them for your needs.

SMTP Sinkhole

Deprecated since version 2.0.0: This script has been removed since this functionality should be implemented properly using a Postfix setup.

Setup script

Deprecated since version 2.0.0: This script has been replaced by a similar but much more powerful SaltStack state.

Cuckoo Router

The `Cuckoo Router` is a new concept, providing `root` access for various commands to Cuckoo (which itself generally speaking runs as non-root). This command is currently only available for Ubuntu and Debian-like systems.

In particular, the `router` helps Cuckoo out with running network-related commands in order to provide **per-analysis routing** options. For more information on that, please refer to the [Per-Analysis Network Routing](#) document. Cuckoo and the `router` communicate through a UNIX socket for which the `router` makes sure that Cuckoo can reach it.

Its usage is as follows:

```
$ cuckoo router --help
Usage: cuckoo router [OPTIONS] [SOCKET]

Options:
  -g, --group TEXT  Unix socket group
  --ifconfig PATH   Path to ifconfig(8)
  --service PATH    Path to service(8) for invoking OpenVPN
  --iptables PATH   Path to iptables(8)
  --ip PATH         Path to ip(8)
  --sudo            Request superuser privileges
  --help            Show this message and exit.
```

By default the `router` will default to chown'ing the `cuckoo` user as user and group for the UNIX socket, as recommended when [Installing Cuckoo](#). If you're running Cuckoo under a user other than `cuckoo`, you will have to specify this to the `router` as follows:

```
$ sudo cuckoo router -g <user>
```

The other options are fairly straightforward - you can specify the paths to specific Linux commands. By default one shouldn't have to do this though, as the `router` takes the default paths for the various utilities as per a default setup.

Virtualenv

Due to the fact that the `router` must be run as `root` user, there are some slight complications when using a `virtualenv` to run Cuckoo. More specifically, when running `sudo cuckoo router`, the `$VIRTUAL_ENV` environment variable will not be passed along, due to which Python will not be executed from the same `virtualenv` as it would have been normally.

To resolve this one simply has to execute the `cuckoo` binary from the `virtualenv` session directly. E.g., if your `virtualenv` is located at `~/venv`, then running the `router` command could be done as follows:

```
$ sudo ~/venv/bin/cuckoo router
```

Alternatively one may use the `--sudo` flag which will call `sudo` on the correct `cuckoo` binary with all the provided flags. In turn the user will have to enter his or her password and, assuming all is fine, the Cuckoo Rooter will be started properly, e.g.:

```
(venv)$ cuckoo rooter --sudo
```

Cuckoo Rooter Usage

Using the `Cuckoo Rooter` is actually pretty easy. If you know how to start it, you're basically good to go. Even though Cuckoo talks with the Cuckoo Rooter for each analysis with a routing option other than *None Routing*, the Cuckoo Rooter does not keep any state or attach to any Cuckoo instance in particular.

It is therefore that once the Cuckoo Rooter has been started you may leave it be - the Cuckoo Rooter will take care of itself from that point onwards, no matter how often you restart your Cuckoo instance.

Cuckoo Feedback

New in version 2.0.0.

The Cuckoo Feedback form allows users to **provide instant feedback** to the Cuckoo Core Developer team. By doing so, our development team will be able to more **quickly react** upon errors, partially incorrect analysis results, errors occurred during an analysis or in the web interface, and anything else that our users think requires some extra attention. All in all, this optional feature gives those users that are interested in a *second opinion* the ability to do so in a convenient way for both the user as well as the team behind Cuckoo Sandbox.

Note: As a user you are able to ping back to us through the Cuckoo Feedback from embedded in most pages of the web interface (e.g., an analysis page or a 404 page not found / 500 internal error page).

Following a screenshot of a part of the *new* (as of Cuckoo 2.0.0) analysis results page with the side bar *locked in* (i.e., permanently open).

The screenshot displays the Cuckoo Sandbox web interface. The top navigation bar includes the Cuckoo logo, a 'Dashboard' button, and links for 'Recent', 'Pending', and 'Search'. A 'Submit' button is located on the right. The left sidebar contains a list of analysis options: Summary, Static Analysis, Behavioral Analysis (with a count of 1), Network Analysis, Dropped Files (with a count of 6), Dropped Buffers, Process Memory (with a count of 1), Compare Analysis, Export Analysis, Reboot Analysis, Options, Feedback, and an 'Unlock sidebar' button. The main content area is titled 'Summary' and shows details for the file 'File CVE-2011-2462.pdf_'. A 'Download' button is present. The file details are as follows:

Summary	
Size	269.2KB
Type	PDF document, version 1.7
MD5	721fda5df552f4130218ad9bd2a4ab78
SHA1	5d89644214f2783b99491ebad61b4c5a8844f7e3
SHA256	036e049c625a2c3fc5f434d0784a2a215fbde7a90c561db730c
SHA512	Show SHA512
CRC32	286348D3
ssdeep	None
Yara	None matched

Below the file details, there is a 'Score' section with a flame icon.

At the bottom of the side bar you'll see the **Feedback** button which will pop up the following feedback form. Naturally filling out all of the fields in this form will allow you to send us feedback (in a secure manner).

It should be noted that, may you decide to provide feedback on a regular, you can also fill out your name, company, and email address (where you'll receive any answers) in the `$CWD/conf/cuckoo.conf` configuration file so those will be auto-filled for you upon opening the feedback form.

The screenshot shows the Cuckoo Sandbox web interface with a sidebar on the left containing navigation links: Summary, Static Analysis, Behavioral Analysis, Network Analysis, Dropped Files, Dropped Buffers, Process Memory, Compare Analysis, Export Analysis, Reboot Analysis, Options, Feedback, and Unlock sidebar. The main content area displays a 'Feedback Form' modal window. The modal has a blue header with the title 'Feedback Form' and a close button. The body of the modal contains the following text: 'Expecting different results? Share this analysis report with us and we will investigate it. Please include a brief message of what you had expected to see and what you got instead.' Below this is a section titled 'The Cuckoo Sandbox team' with a text input field. There are two input fields for 'Your name' and 'Company'. Below these is an 'E-mail' input field. A large text area is labeled 'Message...'. At the bottom of the modal, there are two checkboxes: 'Include analysis' (checked) and 'Include memory dump'. Below the checkboxes, it says 'Estimated size: 513.7 KB'. At the very bottom of the modal are two buttons: 'Send' (green) and 'Close' (grey).

Analysis Packages

The **analysis packages** are a core component of Cuckoo Sandbox. They consist in structured Python classes which, when executed in the guest machines, describe how Cuckoo's analyzer component should conduct the analysis.

Cuckoo provides some default analysis packages that you can use, but you are able to create your own or modify the existing ones. You can find them at `analyzer/windows/modules/packages/`.

As described in [Submit an Analysis](#), you can specify some options to the analysis packages in the form of `key1=value1, key2=value2`. The existing analysis packages already include some default options that can be enabled.

Following is a list of the options that work for all analysis packages unless explicitly stated otherwise:

- `free [yes/no]`: if enabled, no behavioral logs will be produced and the malware will be executed freely.
- `procmemdump [yes/no]`: if enabled, take memory dumps of all actively monitored processes.
- `human 0`: if disabled, human-like interaction (i.e., mouse movements) will not be enabled

Following is the list of existing packages in alphabetical order:

- `applet`: used to analyze **Java applets**.

Options:

- `class`: specify the name of the class to be executed. This option is mandatory for a correct execution.
- `bin`: used to analyze generic binary data, such as **shellcodes**.
- `cpl`: used to analyze **Control Panel Applets**.
- `dll`: used to run and analyze **Dynamically Linked Libraries**.

Options:

- `function`: specify the function to be executed. If none is specified, Cuckoo will try to run `DllMain`.
- `arguments`: specify arguments to pass to the DLL through commandline.
- `loader`: specify a process name to use to fake the DLL launcher name instead of `rundll32.exe` (this is used to fool possible anti-sandboxing tricks of certain malware)
- `doc`: used to run and analyze **Microsoft Word documents**.
- `exe`: default analysis package used to analyze generic **Windows executables**.

Options:

- `arguments`: specify any command line argument to pass to the initial process of the submitted malware.
- `generic`: used to run and analyze **generic samples** via `cmd.exe`.
- `ie`: used to analyze **Internet Explorer**'s behavior when opening the given URL or HTML file.
- `js`: used to run and analyze **Javascript** files (e.g., those found in attachments of emails).
- `jar`: used to analyze **Java JAR** containers.

Options:

- `class`: specify the path of the class to be executed. If none is specified, Cuckoo will try to execute the main function specified in the Jar's MANIFEST file.
- `msi`: used to run and analyze **MSI windows installer**.
- `pdf`: used to run and analyze **PDF documents**.
- `ppt`: used to run and analyze **Microsoft PowerPoint documents**.
- `ps1`: used to run and analyze **PowerShell scripts**.
- `python`: used to run and analyze **Python scripts**.
- `vbs`: used to run and analyze **VBScript files**.
- `wsf`: used to run and analyze **Windows Script Host files**.
- `xls`: used to run and analyze **Microsoft Excel documents**.
- `zip`: used to run and analyze **Zip archives**.

Options:

- `file`: specify the name of the file contained in the archive to execute. If none is specified, Cuckoo will try to execute *sample.exe*.
- `arguments`: specify any command line argument to pass to the initial process of the submitted malware.
- `password`: specify the password of the archive. If none is specified, Cuckoo will try to extract the archive without password or use the password "*infected*".

You can find more details on how to start creating new analysis packages in the [Analysis Packages](#) customization chapter.

As you already know, you can select which analysis package to use by specifying its name at submission time (see [Submit an Analysis](#)) as follows:

```
$ cuckoo submit --package <package name> /path/to/malware
```

If none is specified, Cuckoo will try to detect the file type and select the correct analysis package accordingly. If the file type is not supported by default the analysis will be aborted, therefore we encourage to specify the package name whenever possible.

For example, to launch a malware and specify some options you can do:

```
$ cuckoo submit --package dll --options function=FunctionName,loader=explorer.exe /  
↪path/to/malware.dll
```

Analysis Results

Once an analysis is completed, several files are stored in a dedicated directory. All the analyses are stored under the `$CWD/storage/analyses/` inside a subdirectory named after the incremental numerical ID that represents the analysis task in the database.

Following is an example of an analysis directory structure:

```
.  
|-- analysis.log  
|-- binary  
|-- dump.pcap  
|-- memory.dmp  
|-- files  
|   |-- 1234567890_dropped.exe  
|-- logs  
|   |-- 1232.bson  
|   |-- 1540.bson  
|   |-- 1118.bson  
|-- reports  
|   |-- report.html  
|   |-- report.json  
|-- shots  
|   |-- 0001.jpg  
|   |-- 0002.jpg  
|   |-- 0003.jpg  
|   |-- 0004.jpg
```

analysis.log

This is a log file generated by the analyzer that contains a trace of the analysis execution inside the guest environment. It will report the creation of processes, files and eventual errors occurred during the execution.

dump.pcap

This is the network dump generated by tcpdump or any other corresponding network sniffer.

dump_sorted.pcap

This is a sorted version of `dump.pcap` in the sense that it allows the Web Interface to quickly lookup TCP stream.

memory.dmp

In case you enabled it, this file contains the full memory dump of the analysis machine.

files/

This directory contains all the files the malware operated on and that Cuckoo was able to dump.

files.json

This file contains a JSON-encoded entry for each dropped file available (i.e., all files in `files/`, `shots/`, etc). It contains meta information, where available, about all processes that touched the file, its original file path in the Guest, etc.

logs/

This directory contains all the raw logs generated by Cuckoo's process monitoring.

reports/

This directory contains all the reports generated by Cuckoo as explained in the [Configuration](#) chapter.

shots/

This directory contains all the screenshots of the guest's desktop taken during the malware execution.

tlsmaster.txt

This file contains the TLS Master Secrets that were captured during the analysis. TLS Master Secrets can be used to decrypt SSL/TLS traffic and are thus used to decrypt HTTPS streams.

Clean all Tasks and Samples

Changed in version 2.0.0: Turned into a proper Cuckoo App rather than a standalone script.

Since Cuckoo 1.2 a built-in **clean** feature has been featured, it drops all associated information of the tasks and samples in the database, on the hddisk, from MongoDB, and from ElasticSearch. If you submit a task after running **clean** you'll start over with `Task #1` again.

To clean your setup, run:

```
$ cuckoo clean
```

To sum up, this command does the following:

- Delete analysis results.

- Delete submitted binaries.
- Delete all associated information of the tasks and samples in the configured database.
- Delete all data in the configured MongoDB database (if configured and enabled in `$CWD/conf/reporting.conf`).
- Delete all data in the configured ElasticSearch database (if configured and enabled in `$CWD/conf/reporting.conf`).

Warning: If you use this command you will permanently delete all data stored by Cuckoo in all available storages: the file system, the SQL database, the MongoDB database, and the ElasticSearch database. Use it only if you are sure you would clean up all the data.

Customization

This chapter explains how to customize Cuckoo. Cuckoo is written in a modular architecture built to be as customizable as it can, to fit the needs of all users.

Auxiliary Modules

Auxiliary modules define some procedures that need to be executed in parallel to every single analysis process. All auxiliary modules should be placed under the `cuckoo/cuckoo/auxiliary/` directory, that way the module will fall under the `cuckoo.auxiliary` module.

The skeleton of a module would look something like this:

```
1 from cuckoo.common.abstracts import Auxiliary
2
3 class MyAuxiliary(Auxiliary):
4
5     def start(self):
6         # Do something.
7
8     def stop(self):
9         # Stop the execution.
```

The function `start()` will be executed before starting the analysis machine and effectively executing the submitted malicious file, while the `stop()` function will be launched at the very end of the analysis process, before launching the processing and reporting procedures.

For example, an auxiliary module provided by default in Cuckoo is called `sniffer.py` and takes care of executing `tcpdump` in order to dump the generated network traffic.

Machinery Modules

Machinery modules define how Cuckoo should interact with your virtualization software (or potentially even with physical disk imaging solutions). Since we decided to not enforce any particular vendor, from release 0.4 you are able to use your preferred solution and, in case it's not supported by default, write a custom Python module that defines how to make Cuckoo use it.

Every machinery module should be located inside the `cuckoo/cuckoo/machinery/` directory so that it will fall under the `cuckoo.machinery` module.

A basic machinery module would look like this:

```

1 from cuckoo.common.abstracts import Machinery
2 from cuckoo.common.exceptions import CuckooMachineError
3
4 class MyMachinery(Machinery):
5     def start(self, label):
6         try:
7             revert(label)
8             start(label)
9         except SomethingBadHappens:
10            raise CuckooMachineError("oops!")
11
12    def stop(self, label):
13        try:
14            stop(label)
15        except SomethingBadHappens:
16            raise CuckooMachineError("oops!")

```

The only requirements for Cuckoo are that:

- The class inherits from Machinery.
- You have a `start()` and `stop()` functions.
- You raise `CuckooMachineError` when something fails.

As you understand, the machinery module is a core part of a Cuckoo setup, therefore make sure to spend enough time debugging your code and make it solid and resistant to any unexpected error.

Configuration

Every machinery module should come with a dedicated configuration file located in `$CWD/conf/<machinery module name>.conf` (which translates to `cuckoo/data/conf/<machinery>conf` in the Git repository). For example for `cuckoo/cuckoo/machinery/kvm.py` we have a `$CWD/conf/kvm.conf`.

The configuration file should follow the default structure:

```

[kvm]
# Specify a comma-separated list of available machines to be used. For each
# specified ID you have to define a dedicated section containing the details
# on the respective machine. (E.g. cuckoo1,cuckoo2,cuckoo3)
machines = cuckoo1

[cuckoo1]
# Specify the label name of the current machine as specified in your
# libvirt configuration.
label = cuckoo1

# Specify the operating system platform used by current machine
# [windows/darwin/linux].
platform = windows

# Specify the IP address of the current machine. Make sure that the IP address
# is valid and that the host machine is able to reach it. If not, the analysis
# will fail.
ip = 192.168.122.105

```

A main section called [`<name of the module>`] with a `machines` field containing a comma-separated list of machines IDs.

For each machine you should specify a `label`, a `platform` and its `ip`.

These fields are required by Cuckoo in order to use the already embedded `initialize()` function that generates the list of available machines.

If you plan to change the configuration structure you should override the `initialize()` function (inside your own module, no need to modify Cuckoo's core code). You can find its original code in the `Machinery` abstract inside `cuckoo/common/abstracts.py`.

LibVirt

Starting with Cuckoo 0.5 developing new machinery modules based on LibVirt is easy. Inside `cuckoo/common/abstracts.py` you can find `LibVirtMachinery` that already provides all the functionality for a LibVirt module. Just inherit this base class and specify your connection string, as in the example below:

```
1 from cuckoo.common.abstracts import LibVirtMachinery
2
3 class MyMachinery(LibVirtMachinery):
4     # Set connection string.
5     dsn = "my:///connection"
```

This works for all the virtualization technologies supported by LibVirt. Just remember to check if your LibVirt package (if you are using one, for example from your Linux distribution) is compiled with the support for the technology you need.

You can check it with the following command:

```
$ virsh -V
Virsh command line tool of libvirt 0.9.13
See web site at http://libvirt.org/

Compiled with support for:
Hypervisors: QEmu/KVM LXC UML Xen OpenVZ VMWare Test
Networking: Remote Daemon Network Bridging Interface Nwfilter VirtualPort
Storage: Dir Disk Filesystem SCSI Multipath iSCSI LVM
Miscellaneous: Nodedev AppArmor Secrets Debug Readline Modular
```

If you don't find your virtualization technology in the list of `Hypervisors`, you will need to recompile LibVirt with the specific support for the missing one.

Analysis Packages

As explained in *Analysis Packages*, analysis packages are structured Python classes that describe how Cuckoo's analyzer component should conduct the analysis procedure for a given file inside the guest environment.

As you already know, you can create your own packages and add them along with the default ones. Designing new packages is very easy and requires just a minimal understanding of programming and of the Python language.

Getting started

As an example we'll take a look at the default package for analyzing generic Windows executables, located at `$CWD/analyzer/windows/packages/exe.py` (which translates to `cuckoo/data/analyzer/windows/packages/exe.py` in the Git repository):

```

1 from lib.common.abstracts import Package
2
3 class Exe(Package):
4     """EXE analysis package."""
5
6     def start(self, path):
7         args = self.options.get("arguments")
8         return self.execute(path, args)

```

It seems really easy, thanks to all method inherited by Package object. Let's have a look at some of the main methods an analysis package inherits from Package object:

```

1 from lib.api.process import Process
2 from lib.common.exceptions import CuckooPackageError
3
4 class Package(object):
5     def start(self):
6         raise NotImplementedError
7
8     def check(self):
9         return True
10
11    def execute(self, path, args):
12        dll = self.options.get("dll")
13        free = self.options.get("free")
14        suspended = True
15        if free:
16            suspended = False
17
18        p = Process()
19        if not p.execute(path=path, args=args, suspended=suspended):
20            raise CuckooPackageError(
21                "Unable to execute the initial process, analysis aborted."
22            )
23
24        if not free and suspended:
25            p.inject(dll)
26            p.resume()
27            p.close()
28            return p.pid
29
30    def finish(self):
31        if self.options.get("procmemdump"):
32            for pid in self.pids:
33                p = Process(pid=pid)
34                p.dump_memory()
35        return True

```

Let's walk through the code:

- Line 1: import the `Process` API class, which is used to create and manipulate Windows processes.
- Line 2: import the `CuckooPackageError` exception, which is used to notify issues with the execution of the package to the analyzer.
- Line 4: define the main class, inheriting `object`.
- Line 5: define the `start()` function, which takes as argument the path to the file to execute. It should be implemented by each analysis package.

- Line 8: define the `check()` function.
- Line 13: acquire the `free` option, which is used to define whether the process should be monitored or not.
- Line 18: initialize a `Process` instance.
- Line 19: try to execute the malware, if it fails it aborts the execution and notify the analyzer.
- Line 24: check if the process should be monitored.
- Line 25: inject the process with our DLL.
- Line 26: resume the process from the suspended state.
- Line 28: return the PID of the newly created process to the analyzer.
- Line 30: define the `finish()` function.
- Line 31: check if the `procmemdump` option was enabled.
- Line 32: loop through the currently monitored processes.
- Line 33: open a `Process` instance.
- Line 34: take a dump of the process memory.

`start()`

In this function you have to place all the initialization operations you want to run. This may include running the malware process, launching additional applications, taking memory snapshots and more.

`check()`

This function is executed by Cuckoo every second while the malware is running. You can use this function to perform any kind of recurrent operation.

For example if in your analysis you are looking for just one specific indicator to be created (e.g., a file) you could place your condition in this function and if it returns `False`, the analysis will terminate right away.

Think of it as “should the analysis continue or not?”.

For example:

```
def check(self):
    if os.path.exists("C:\\config.bin"):
        return False
    else:
        return True
```

This `check()` function will cause Cuckoo to immediately terminate the analysis whenever `C:\\config.bin` is created.

`execute()`

Wraps the malware execution and deal with DLL injection.

finish()

This function is simply called by Cuckoo before terminating the analysis and powering off the machine. By default, this function contains an optional feature to dump the process memory of all the monitored processes.

Options

Every package have automatically access to a dictionary containing all user-specified options (see [Submit an Analysis](#)).

Such options are made available in the attribute `self.options`. For example let's assume that the user specified the following string at submission:

```
foo=1,bar=2
```

The analysis package selected will have access to these values:

```
from lib.common.abstracts import Package

class Example(Package):

    def start(self, path):
        foo = self.options["foo"]
        bar = self.options["bar"]

    def check():
        return True

    def finish():
        return True
```

These options can be used for anything you might need to configure inside your package.

Process API

The `Process` class provides access to different process-related features and functions. You can import it in your analysis packages with:

```
from lib.api.process import Process
```

You then initialize an instance with:

```
p = Process()
```

In case you want to open an existing process instead of creating a new one, you can specify multiple arguments:

- `pid`: PID of the process you want to operate on.
- `h_process`: handle of a process you want to operate on.
- `thread_id`: thread ID of a process you want to operate on.
- `h_thread`: handle of the thread of a process you want to operate on.

This class implements several methods that you can use in your own scripts.

Methods

`Process.open()`

Opens an handle to a running process. Returns `True` or `False` in case of success or failure of the operation.

Return type `boolean`

Example Usage:

```
1 p = Process(pid=1234)
2 p.open()
3 handle = p.h_process
```

`Process.exit_code()`

Returns the exit code of the opened process. If it wasn't already done before, `exit_code()` will perform a call to `open()` to acquire an handle to the process.

Return type `ulong`

Example Usage:

```
1 p = Process(pid=1234)
2 code = p.exit_code()
```

`Process.is_alive()`

Calls `exit_code()` and verify if the returned code is `STILL_ACTIVE`, meaning that the given process is still running. Returns `True` or `False`.

Return type `boolean`

Example Usage:

```
1 p = Process(pid=1234)
2 if p.is_alive():
3     print("Still running!")
```

`Process.get_parent_pid()`

Returns the PID of the parent process of the opened process. If it wasn't already done before, `get_parent_pid()` will perform a call to `open()` to acquire an handle to the process.

Return type `int`

Example Usage:

```
1 p = Process(pid=1234)
2 ppid = p.get_parent_pid()
```

`Process.execute(path[, args=None[, suspended=False]])`

Executes the file at the specified path. Returns `True` or `False` in case of success or failure of the operation.

Parameters

- **path** (*string*) – path to the file to execute
- **args** (*string*) – arguments to pass to the process command line
- **suspended** (*boolean*) – enable or disable suspended mode flag at process creation

Return type `boolean`

Example Usage:

```

1 p = Process()
2 p.execute(path="C:\\WINDOWS\\system32\\calc.exe", args="Something",
↳suspended=True)

```

Process.resume()

Resumes the opened process from a suspended state. Returns True or False in case of success or failure of the operation.

Return type boolean

Example Usage:

```

1 p = Process()
2 p.execute(path="C:\\WINDOWS\\system32\\calc.exe", args="Something",
↳suspended=True)
3 p.resume()

```

Process.terminate()

Terminates the opened process. Returns True or False in case of success or failure of the operation.

Return type boolean

Example Usage:

```

1 p = Process(pid=1234)
2 if p.terminate():
3     print("Process terminated!")
4 else:
5     print("Could not terminate the process!")

```

Process.inject([dll[, apc=False]])

Injects our DLL into the opened process. Returns True or False in case of success or failure of the operation.

Parameters

- **dll** (*string*) – path to the DLL to inject into the process
- **apc** (*boolean*) – enable to use `QueueUserAPC()` injection instead of `CreateRemoteThread()`, beware that if the process is in suspended mode, Cuckoo will always use `QueueUserAPC()`

Return type boolean

Example Usage:

```

1 p = Process()
2 p.execute(path="C:\\WINDOWS\\system32\\calc.exe", args="Something",
↳suspended=True)
3 p.inject()
4 p.resume()

```

Process.dump_memory()

Takes a snapshot of the given process' memory space. Returns True or False in case of success or failure of the operation.

Return type boolean

Example Usage:

```

1 p = Process(pid=1234)
2 p.dump_memory()

```

Processing Modules

Cuckoo's processing modules are Python scripts that let you define custom ways to analyze the raw results generated by the sandbox and append some information to a global container that will be later used by the signatures and the reporting modules.

You can create as many modules as you want, as long as they follow a predefined structure that we will present in this chapter.

Global Container

After an analysis is completed, Cuckoo will invoke all the processing modules available in the `cuckoo/processing/` directory, all of which fall under the `cuckoo.processing` module. Any additional module you decide to create must be placed inside that directory.

Every module should also have a dedicated section in the `$CWD/conf/processing.conf` file: for example if you create a module `cuckoo/processing/foobar.py` you will have to append the following section to `$CWD/conf/processing.conf`:

```
[foobar]
enabled = yes
```

Every module will then be initialized and executed and the data returned will be appended in a data structure that we'll call **global container**.

This container is simply just a big Python dictionary that includes the abstracted results produced by all the modules classified by their identification key.

Cuckoo already provides a default set of modules which will generate a *standard* global container. It's important for the existing reporting modules (HTML report etc.) that these default modules are not modified, otherwise the resulting global container structure would change and the reporting modules wouldn't be able to recognize it and extract the information used to build the final reports.

The currently available default processing modules are:

- **AnalysisInfo** (`cuckoo/processing/analysisinfo.py`) - generates some basic information on the current analysis, such as timestamps, version of Cuckoo and so on.
- **ApkInfo** (`cuckoo/processing/apkinfo.py`) - generates some basic information on the current APK analysis (Android analysis).
- **Baseline** (`cuckoo/processing/baseline.py`) - baseline results from gathered information.
- **BehaviorAnalysis** (`cuckoo/processing/behavior.py`) - parses the raw behavioral logs and perform some initial transformations and interpretations, including the complete processes tracing, a behavioral summary and a process tree.
- **Buffer** (`cuckoo/processing/buffer.py`) - dropped buffer analysis.
- **Debug** (`cuckoo/processing/debug.py`) - includes errors and the *analysis.log* generated by the analyzer.
- **Droidmon** (`cuckoo/processing/droidmon.py`) - extract Dynamic API calls Info From Droidmon logs.
- **Dropped** (`cuckoo/processing/dropped.py`) - includes information on the files dropped by the malware and dumped by Cuckoo.
- **DumpTls** (`cuckoo/processing/dumptls.py`) - cross-references TLS master secrets extracted from the monitor and key information extracted from the PCAP to dump a master secrets file.

- **GooglePlay** (cuckoo/processing/googleplay.py) - Google Play information about the analysis session.
- **Irma** (cuckoo/processing/irma.py) - IRMA connector.
- **Memory** (cuckoo/processing/memory.py) - executes Volatility on a full memory dump.
- **Misp** (cuckoo/processing/misp.py) - MISP connector.
- **NetworkAnalysis** (cuckoo/processing/network.py) - parses the PCAP file and extracts some network information, such as DNS traffic, domains, IPs, HTTP requests, IRC and SMTP traffic.
- **ProcMemory** (cuckoo/processing/procmemory.py) - performs analysis of process memory dump. **Note:** the module is able to process user defined Yara rules from data/yara/memory/index_memory.yar. Just edit this file to add your Yara rules.
- **ProcMon** (cuckoo/processing/procmon.py) - extracts events from procmon.exe output.
- **Screenshots** (cuckoo/processing/screenshots.py) - screenshot and OCR analysis.
- **Snort** (cuckoo/processing/snort.py) - Snort processing module.
- **StaticAnalysis** (cuckoo/processing/static.py) - performs some static analysis of PE32 files.
- **Strings** (cuckoo/processing/strings.py) - extracts strings from the analyzed binary.
- **Suricata** (cuckoo/processing/suricata.py) - Suricata processing module.
- **TargetInfo** (cuckoo/processing/targetinfo.py) - includes information on the analyzed file, such as hashes.
- **VirusTotal** (cuckoo/processing/virustotal.py) - searches on VirusTotal.com for antivirus signatures of the analyzed file. **Note:** the file is not uploaded on VirusTotal.com, if the file was not previously uploaded on the website no results will be retrieved.

Getting started

In order to make them available to Cuckoo, all processing modules must be placed inside the cuckoo/processing/ directory.

A basic processing module could look like:

```

1 from cuckoo.common.abstracts import Processing
2
3 class MyModule(Processing):
4
5     def run(self):
6         self.key = "key"
7         data = do_something()
8         return data

```

Every processing module should contain:

- A class inheriting Processing.
- A run() function.
- A self.key attribute defining the name to be used as a sub container for the returned data.
- A set of data (list, dictionary, string, etc.) that will be appended to the global container.

You can also specify an order value, which allows you to run the available processing modules in an ordered sequence. By default all modules are set with an order value of 1 and are executed in alphabetical order.

If you want to change this value your module would look like:

```
1 from cuckoo.common.abstracts import Processing
2
3 class MyModule(Processing):
4     order = 2
5
6     def run(self):
7         self.key = "key"
8         data = do_something()
9         return data
```

You can also manually disable a processing module by setting the `enabled` attribute to `False`:

```
1 from cuckoo.common.abstracts import Processing
2
3 class MyModule(Processing):
4     enabled = False
5
6     def run(self):
7         self.key = "key"
8         data = do_something()
9         return data
```

The processing modules are provided with some attributes that can be used to access the raw results for the given analysis:

- `self.analysis_path`: path to the folder containing the results (e.g., `$CWD/storage/analysis/1`)
- `self.log_path`: path to the *analysis.log* file.
- `self.file_path`: path to the analyzed file.
- `self.dropped_path`: path to the folder containing the dropped files.
- `self.logs_path`: path to the folder containing the raw behavioral logs.
- `self.shots_path`: path to the folder containing the screenshots.
- `self.pcap_path`: path to the network pcap dump.
- `self.memory_path`: path to the full memory dump, if created.
- `self.pmemory_path`: path to the process memory dumps, if created.

With these attributes you should be able to easily access all the raw results stored by Cuckoo and perform your analytic operations on them.

As a last note, a good practice is to use the `CuckooProcessingError` exception whenever the module encounters an issue you want to report to Cuckoo. This can be done by importing the class like this:

```
1 from cuckoo.common.exceptions import CuckooProcessingError
2 from cuckoo.common.abstracts import Processing
3
4 class MyModule(Processing):
5
6     def run(self):
7         self.key = "key"
8
9         try:
10             data = do_something()
11         except SomethingFailed:
```

```

12         raise CuckooProcessingError("Failed")
13
14     return data

```

Signatures

With Cuckoo you're able to create some customized signatures that you can run against the analysis results in order to identify some predefined pattern that might represent a particular malicious behavior or an indicator you're interested in.

These signatures are very useful to give a context to the analyses: both because they simplify the interpretation of the results as well as for automatically identifying malware samples of interest.

Some examples of what you can use Cuckoo's signatures for:

- Identify a particular malware family you're interested in by isolating some unique behaviors (like file names or mutexes).
- Spot interesting modifications the malware performs on the system, such as installation of device drivers.
- Identify particular malware categories, such as Banking Trojans or Ransomware by isolating typical actions commonly performed by those.
- Classify samples into the categories malware/unknown (it is not possible to identify clean samples)

You can find signatures created by us and by other Cuckoo users on our [Community](#) repository.

Getting started

Creation of signatures is a fairly simple process and requires just a decent understanding of Python programming.

First things first, all signatures must be located inside the `cuckoo/cuckoo/signatures/` directory in Cuckoo or the `modules/signatures/` directory of the [Community](#) repository (the Community repository is still using legacy directory structuring).

The following is a basic example signature:

```

1  from cuckoo.common.abstracts import Signature
2
3  class CreatesExe(Signature):
4      name = "creates_exe"
5      description = "Creates a Windows executable on the filesystem"
6      severity = 2
7      categories = ["generic"]
8      authors = ["Cuckoo Developers"]
9      minimum = "2.0"
10
11     def on_complete(self):
12         return self.check_file(pattern=".*\\.exe$", regex=True)

```

As you can see the structure is really simple and consistent with the other modules. We're going to get into details later, but since version 1.2 Cuckoo provides some helper functions that make the process of creating signatures much easier.

In this example we just walk through all the accessed files in the summary and check if there is anything ending with `“.exe”`: in that case it will return `True`, meaning that the signature matched, otherwise return `False`.

The function `on_complete` is called at the end of the cuckoo signature process. Other function will be called before on specific events and help you to write more sophisticated and faster signatures.

In case the signature gets matched, a new entry in the “signatures” section will be added to the global container roughly as follows:

```
"signatures": [
  {
    "severity": 2,
    "description": "Creates a Windows executable on the filesystem",
    "alert": false,
    "references": [],
    "data": [
      {
        "file_name": "C:\\d.exe"
      }
    ],
    "name": "creates_exe"
  }
]
```

Creating your new signature

In order to make you better understand the process of creating a signature, we are going to create a very simple one together and walk through the steps and the available options. For this purpose, we’re simply going to create a signature that checks whether the malware analyzed opened a mutex named “`i_am_a_malware`”.

The first thing to do is import the dependencies, create a skeleton and define some initial attributes. These are the ones you can currently set:

- `name`: an identifier for the signature.
- `description`: a brief description of what the signature represents.
- `severity`: a number identifying the severity of the events matched (generally between 1 and 3).
- `categories`: a list of categories that describe the type of event being matched (for example “*banker*”, “*injection*” or “*anti-vm*”).
- `families`: a list of malware family names, in case the signature specifically matches a known one.
- `authors`: a list of people who authored the signature.
- `references`: a list of references (URLs) to give context to the signature.
- `enable`: if set to `False` the signature will be skipped.
- `alert`: if set to `True` can be used to specify that the signature should be reported (perhaps by a dedicated reporting module).
- `minimum`: the minimum required version of Cuckoo to successfully run this signature.
- `maximum`: the maximum required version of Cuckoo to successfully run this signature.

In our example, we would create the following skeleton:

```
1 from cuckoo.common.abstracts import Signature
2
3 class BadBadMalware(Signature): # We initialize the class inheriting Signature.
4     name = "badbadmalware" # We define the name of the signature
5     description = "Creates a mutex known to be associated with Win32.BadBadMalware" #
    ↪ We provide a description
```



```

6     severity = 3 # We set the severity to maximum
7     categories = ["trojan"] # We add a category
8     families = ["badbadmalware"] # We add the name of our fictional malware family
9     authors = ["Me"] # We specify the author
10    minimum = "2.0" # We specify that in order to run the signature, the user will
    ↪ simply need Cuckoo 2.0
11
12    def on_complete(self):
13        return

```

This is a perfectly valid signature. It doesn't really do anything yet, so now we need to define the conditions for the signature to be matched.

As we said, we want to match a particular mutex name, so we proceed as follows:

```

1  from cuckoo.common.abstracts import Signature
2
3  class BadBadMalware(Signature):
4      name = "badbadmalware"
5      description = "Creates a mutex known to be associated with Win32.BadBadMalware"
6      severity = 3
7      categories = ["trojan"]
8      families = ["badbadmalware"]
9      authors = ["Me"]
10     minimum = "2.0"
11
12     def on_complete(self):
13         return self.check_mutex("i_am_a_malware")

```

Simple as that, now our signature will return True whether the analyzed malware was observed opening the specified mutex.

If you want to be more explicit and directly access the global container, you could translate the previous signature in the following way:

```

1  from cuckoo.common.abstracts import Signature
2
3  class BadBadMalware(Signature):
4      name = "badbadmalware"
5      description = "Creates a mutex known to be associated with Win32.BadBadMalware"
6      severity = 3
7      categories = ["trojan"]
8      families = ["badbadmalware"]
9      authors = ["Me"]
10     minimum = "2.0"
11
12     def on_complete(self):
13         for process in self.get_processes_by_pid():
14             if "summary" in process and "mutexes" in process["summary"]:
15                 for mutex in process["summary"]["mutexes"]:
16                     if mutex == "i_am_a_malware":
17                         return True
18
19         return False

```

Evented Signatures

Since version 1.0, Cuckoo provides a way to write more high performance signatures. In the past every signature was required to loop through the whole collection of API calls collected during the analysis. This was unnecessarily causing performance issues when such collection would be of a large size.

Since 1.2 Cuckoo only supports the so called “evented signatures”. The old signatures based on the `run` function can be ported to using `on_complete`. The main difference is that with this new format, all the signatures will be executed in parallel and a callback function called `on_call()` will be invoked for each signature within one single loop through the collection of API calls.

An example signature using this technique is the following:

```
1 from cuckoo.common.abstracts import Signature
2
3 class SystemMetrics(Signature):
4     name = "generic_metrics"
5     description = "Uses GetSystemMetrics"
6     severity = 2
7     categories = ["generic"]
8     authors = ["Cuckoo Developers"]
9     minimum = "2.0"
10
11     # Evented signatures can specify filters that reduce the amount of
12     # API calls that are streamed in. One can filter Process name, API
13     # name/identifier and category. These should be sets for faster lookup.
14     filter_processnames = set()
15     filter_apinames = set(["GetSystemMetrics"])
16     filter_categories = set()
17
18     # This is a signature template. It should be used as a skeleton for
19     # creating custom signatures, therefore is disabled by default.
20     # The on_call function is used in "evented" signatures.
21     # These use a more efficient way of processing logged API calls.
22     enabled = False
23
24     def on_complete(self):
25         # In the on_complete method one can implement any cleanup code and
26         # decide one last time if this signature matches or not.
27         # Return True in case it matches.
28         return False
29
30     # This method will be called for every logged API call by the loop
31     # in the RunSignatures plugin. The return value determines the "state"
32     # of this signature. True means the signature matched and False it did not this_
33     ↪time.
34     # Use self.deactivate() to stop streaming in API calls.
35     def on_call(self, call, pid, tid):
36         # This check would in reality not be needed as we already make use
37         # of filter_apinames above.
38         if call["api"] == "GetSystemMetrics":
39             # Signature matched, return True.
40             return True
41
42         # continue
43         return None
```

The inline comments are already self-explanatory.

Another event is triggered when a signature matches.

```

1 def on_signature(self, matched_sig):
2     required = ["creates_exe", "badmalware"]
3     for sig in required:
4         if not sig in self.list_signatures():
5             return
6     return True

```

This kind of signature can be used to combine several signatures identifying anomalies into one signature classifying the sample (malware alert).

Marks & Helpers

Starting from version 1.2, signatures are able to log exactly what triggered the signature. This allows users to better understand why this signature is present in the log, and to be able to better focus malware analysis.

For examples on marks and helpers please refer to the Cuckoo [Community](#) for now - until we write some thorough up-to-date documentation on that.

Reporting Modules

After the raw analysis results have been processed and abstracted by the processing modules and the global container is generated (ref. *Processing Modules*), it is passed over by Cuckoo to all the reporting modules available, which will make use of it and will make it accessible and consumable in different formats.

Getting Started

All reporting modules must be placed inside the `cuckoo/cuckoo/reporting/` directory (which translates to the `cuckoo.reporting` module).

Every module must also have a dedicated section in the `$CWD/conf/reporting.conf` file: for example if you create a module `cuckoo/cuckoo/reporting/foobar.py` you will have to append the following section to `$CWD/conf/reporting.conf` (and thus `cuckoo/data/conf/reporting.conf` in the Git repository):

```

[foobar]
enabled = on

```

Every additional option you add to your section will be available to your reporting module in the `self.options` dictionary.

Following is an example of a working JSON reporting module:

```

1 import os
2 import json
3 import codecs
4
5 from cuckoo.common.abstracts import Report
6 from cuckoo.common.exceptions import CuckooReportError
7
8 class JsonDump(Report):
9     """Saves analysis results in JSON format."""
10
11     def run(self, results):
12         """Writes report.

```

```
13     @param results: Cuckoo results dict.
14     @raise CuckooReportError: if fails to write report.
15     """
16     try:
17         report = codecs.open(os.path.join(self.reports_path, "report.json"), "w",
↪ "utf-8")
18         json.dump(results, report, sort_keys=False, indent=4)
19         report.close()
20     except (UnicodeError, TypeError, IOError) as e:
21         raise CuckooReportError("Failed to generate JSON report: %s" % e)
```

This code is very simple, it receives the global container produced by the processing modules, converts it into JSON and writes it to a file.

There are few requirements for writing a valid reporting module:

- Declare your class inheriting from `Report`.
- Have a `run()` function performing the main operations.
- Try to catch most exceptions and raise `CuckooReportError` to notify the issue.

All reporting modules have access to some attributes:

- `self.analysis_path`: path to the folder containing the raw analysis results (e.g. *storage/analyses/1/*)
- `self.reports_path`: path to the folder where the reports should be written (e.g. *storage/analyses/1/reports/*)
- `self.options`: a dictionary containing all the options specified in the report's configuration section in *conf/reporting.conf*.

Development

This chapter explains how to write Cuckoo's code and how to contribute.

Development Notes

Git branches

Cuckoo Sandbox source code is available in our [official Git repository](#).

Up until version 1.0 we used to coordinate all ongoing development in a dedicated “development” branch and we’ve been exclusively merging pull requests in such branch. Since version 1.1 we moved development to the traditional “master” branch and we make use of GitHub’s tags and release system to reference development milestones in time.

Release Versioning

Cuckoo releases are named using three numbers separated by dots, such as 1.2.3, where the first number is the release, the second number is the major version, the third number is the bugfix version. The testing stage from git ends with “-beta” and development stage with “-dev”.

Warning: If you are using a “beta” or “dev” stage, please consider that it’s not meant to be an official release, therefore we don’t guarantee its functioning and we don’t generally provide support. If you think you encountered a bug there, make sure that the nature of the problem is not related to your own misconfiguration and collect all the details to be notified to our developers. Make sure to specify which exact version you are using, eventually with your current git commit id.

Ticketing system

To submit bug reports or feature requests, please use GitHub’s [Issue](#) tracking system.

Contribute

To submit your patch just create a Pull Request from your GitHub fork. If you don’t now how to create a Pull Request take a look to [GitHub help](#).

Coding Style

In order to contribute code to the project, you must diligently follow the style rules describe in this chapter. Having a clean and structured code is very important for our development lifecycle. We do help out with code refactoring where required, but please try to do as much as possible on your own.

Essentially Cuckoo’s code style is based on [PEP 8 - Style Guide for Python Code](#) and [PEP 257 – Docstring Conventions](#).

Formatting

Copyright header

All existing source code files start with the following copyright header:

```
# Copyright (C) 2010-2013 Claudio Guarnieri.
# Copyright (C) 2014-2016 Cuckoo Foundation.
# This file is part of Cuckoo Sandbox - http://www.cuckoosandbox.org
# See the file 'docs/LICENSE' for copying permission.
```

Newly created files should start with the following copyright header:

```
# Copyright (C) 2016 Cuckoo Foundation.
# This file is part of Cuckoo Sandbox - http://www.cuckoosandbox.org
# See the file 'docs/LICENSE' for copying permission.
```

Indentation

The code must have a 4-spaces-tabs indentation. Since Python enforce the indentation, make sure to configure your editor properly or your code might cause malfunctioning.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

Blank Lines

Separate the class definition and the top level function with one blank line. Methods definitions inside a class are separated by a single blank line:

```
class MyClass:
    """Doing something."""

    def __init__(self):
        """Initialize"""
        pass

    def do_it(self, what):
        """Do it.
        @param what: do what.
        """
        pass
```

Use blank lines in functions, sparingly, to isolate logic sections. Import blocks are separated by a single blank line, import blocks are separated from classes by one blank line.

Imports

Imports must be on separate lines. If you're importing multiple objects from a package, use a single line:

```
from lib import a, b, c
```

NOT:

```
from lib import a
from lib import b
from lib import c
```

Always specify explicitly the objects to import:

```
from lib import a, b, c
```

NOT:

```
from lib import *
```

Strings

Strings must be delimited by double quotes ("").

Printing and Logging

We discourage the use of `print()`: if you need to log an event please use Python's logging which is already initialized by Cuckoo.

In your module add:

```
import logging
log = logging.getLogger(__name__)
```

And use the `log` handle. More details can be found in the Python documentation, but as follows is an example:

```
log.info("Log message")
```

Exceptions

Custom exceptions must be defined in the `cuckoo/common/exceptions.py` file.

The following is the current Cuckoo exceptions chain:

```
.-- CuckooCriticalError
|   |-- CuckooStartupError
|   |-- CuckooDatabaseError
|   |-- CuckooMachineError
|   `-- CuckooDependencyError
|-- CuckooOperationalError
|   |-- CuckooAnalysisError
|   |-- CuckooProcessingError
|   `-- CuckooReportError
`-- CuckooGuestError
```

Beware that the use of `CuckooCriticalError` and its child exceptions will cause Cuckoo to terminate.

Naming

Custom exception names must start with “Cuckoo” and end with “Error” if it represents an unexpected malfunction.

Exception handling

When catching an exception and accessing its handle, use as `e`:

```
try:
    foo()
except Exception as e:
    bar()
```

NOT:

```
try:
    foo()
except Exception, something:
    bar()
```

It’s a good practice use “`e`” instead of “`e.message`”.

Documentation

All code must be documented in docstring format, see [PEP 257 – Docstring Conventions](#). Additional comments may be added in logical blocks to make the code easier to understand.

Automated testing

We believe in automated testing to provide high quality code and avoid dumb bugs. When possible, all code must be committed with proper unit tests. Particular attention must be placed when fixing bugs: it's good practice to write unit tests to reproduce the bug. All unit tests and fixtures are placed in the tests folder in the Cuckoo root. We have adopted [Pytest](#) as unit testing framework.

Development with the Python Package

With the new Python package developing and testing code now works slightly different than it used to be. As one will first have to [Install Cuckoo](#) before being able to use it in the first place, a simple modify-and-test development sequence doesn't work out-of-the-box as it used to do.

Following we outline how to develop and test new features while using the Cuckoo Package.

- Initialize a new virtualenv.

```
$ virtualenv /tmp/cuckoo-development
$ . /tmp/cuckoo-development/bin/activate
```

- In order to create a Cuckoo distribution package it is required to obtain the matching monitoring binaries from our [Community repository](#) for this version of Cuckoo. Fortunately we provide a simple-to-use script to fetch them semi-automatically for you. From the repository root directory one may run as follows to automatically grab the binaries.

```
(cuckoo-development)$ python stuff/monitor.py
```

- Install Cuckoo in development mode, in which files from the current directory (a git clone'd Cuckoo repository on the package branch) will be used during execution.

```
(cuckoo-development)$ python setup.py develop
```

You will now be ready to modify and test files. Note that the code files are located in the [cuckoo/ directory](#) of the Git repository and the fact that, even though you will be testing a development version of the repository, all the *rules* from the [Cuckoo Working Directory](#) and [Cuckoo Working Directory Usage](#) are still in-place.

Happy development! Please reach out to us if you require additional help to get up-and-running with the latest development tricks.

Final Remarks

Links

- www.cuckoosandbox.org
- community.cuckoosandbox.org
- github.com/cuckoosandbox
- malwr.com

Join the discussion

You can get in contact with the Cuckoo developers and users through the [Community](#) portal or on IRC at the official #cuckoosandbox channel.

If you are encountering an issue you can't solve and are looking for some help, go to our [Community](#) website.

Please read the following rules before posting:

- Before posting, read the [Community](#) archives, the Cuckoo blog, the documentation and Google about your issue. **DO NOT** post questions that have already been answered over and over everywhere.
- Posting messages saying just something like “Doesn't work, help me” are completely useless. If something is not working report the error, paste the logs, the config file, the information on the virtual machine, the results of the troubleshooting, etc. Give context. We are not wizards and we don't have a crystal ball.
- Use a proper title. Stuff like “Doesn't work”, “Help me”, “Error” are not proper titles.

Support Us

Cuckoo Sandbox is a completely open source software, released freely to the public and developed mostly during free time by volunteers. If you enjoy it and want to see it kept developed and updated, please consider supporting us.

We are always looking for financial support, hardware support and contributions of any sort. If you're interested in cooperating, feel free to contact us.

People

Cuckoo Sandbox is an open source project result of the efforts and contributions of a lot of people who enjoyed volunteering some of their time for a greater good :).

Active Developers

Name	Role	Contact
Claudio nex Guarnieri	Project Leader	nex at nex dot sx
Alessandro jekil Tanasi	Core Developer	alessandro at tanasi dot it
Jurriaan skier Bremer	Core Developer	jbr at cuckoo dot sh
Mark rep Schloesser	Core Developer	ms at mwcollect dot org

Contributors

It's hard at this point to keep track of all individual contributions. In the [Cuckoo Contributors](#) page there is the list of people who contributed code to our GitHub repository.

There is a number of friends who provided feedback, ideas and support during the years of development of this project, including but not limited to:

- Felix Leder
- Tillmann Werner
- Georg Wicherski
- David Watson
- Christian Seifert

Supporters

- The HoneyNet Project
- The Shadowserver Foundation

A

`add_path()` (built-in function), [59](#)

`add_url()` (built-in function), [59](#)

P

`Process.dump_memory()` (built-in function), [103](#)

`Process.execute()` (built-in function), [102](#)

`Process.exit_code()` (built-in function), [102](#)

`Process.get_parent_pid()` (built-in function), [102](#)

`Process.inject()` (built-in function), [103](#)

`Process.is_alive()` (built-in function), [102](#)

`Process.open()` (built-in function), [102](#)

`Process.resume()` (built-in function), [103](#)

`Process.terminate()` (built-in function), [103](#)