Cuckoo Sandbox Book

Release 0.5

Cuckoo Sandbox

September 02, 2015

Contents

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 you how to setup Cuckoo, use it and customize it.

CHAPTER 1

Having troubles?

If you're having troubles you might want to check out the FAQ it might already have the answers to your questions.

1.1 FAQ

Frequently Asked Questions:

1.1.1 General Questions

Can I use Volatility with Cuckoo?

Cuckoo 0.5 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 operaitons, the results of a forensic analysis would be polluted by the sandbox's components.

1.1.2 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 Upgrade from a previous release.

Cuckoo stumbles in some error I don't understand

Cuckoo is a young and still evolving project, it might definitely happen that you will occur in some problems while running it, but before you rush into sending emails to everyone make sure to 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 efforts 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 actual bugs.

- We have an 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 curent 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 firt 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 curent 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
```

Otherwise you can ask to the developers and to other Cuckoo users, see Join the discussion.

CHAPTER 2

Contents

2.1 Introduction

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

2.1.1 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 and what it does.

Malware sandboxing is a practical application of the dynamical analysis approach: instead of statically analyze 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 installing, configuring and using 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?
- Which volumes 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 (the 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 softwares 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 could 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.

2.1.2 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 Windows operating system.

It can retrieve the following type of results:

- Traces of win32 API 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 of Windows desktop 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 main developer and coordinates all efforts from joined developers and contributors.

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 functionalities.

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 functionalities.

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.

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
- PHP scripts
- *Almost anything else*

Thanks to its modularity and powerful scripting capabilities, there's not limit to what you can achieve with Cuckoo.

For more information on customizing Cuckoo, see the Customization chapter.

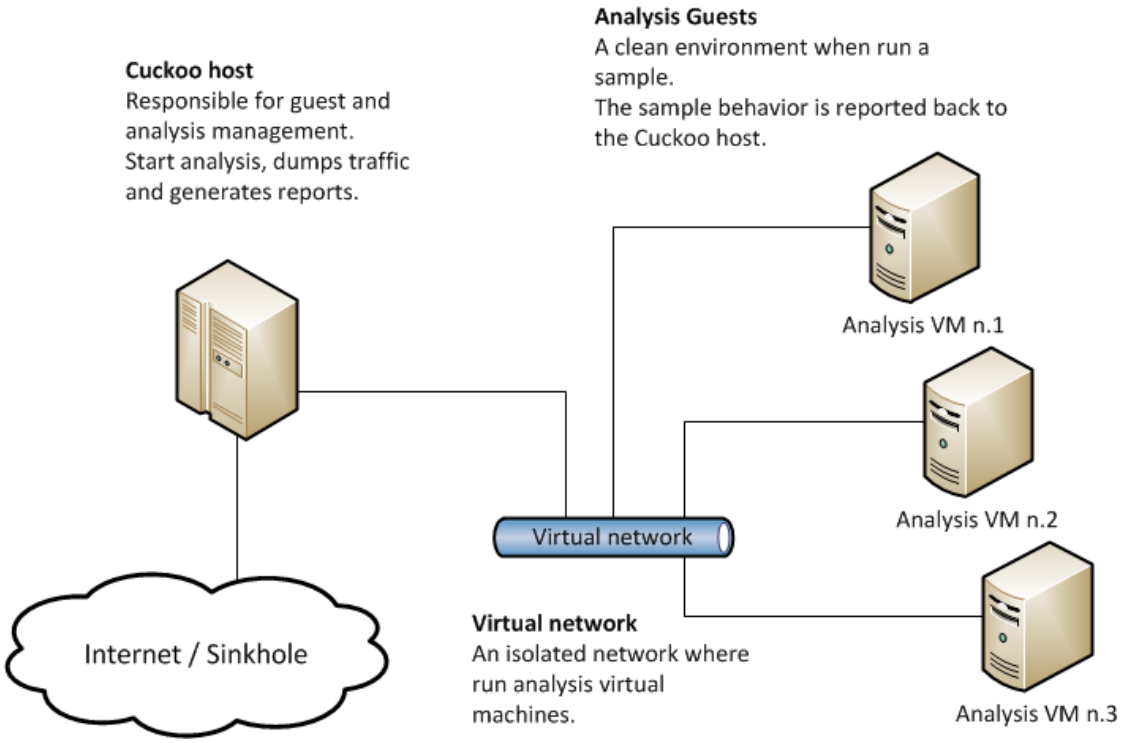
Architecture

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

Each analysis is launched in a fresh and isolated virtual machine. Cuckoo's infrastructure is composed by an Host machine (the management software) and a number of Guest machines (virtual 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 malwares get actually safely executed and analyzed.

The following picture explains Cuckoo's main architecture:



Although recommended setup is *GNU/Linux* (Ubuntu preferrably) as host and *Windows XP Service Pack 3* as guest, Cuckoo proved to work smoothly also on *Mac OS X* as host and *Windows Vista* and *Windows 7* as guests.

Obtaining 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.

2.1.3 License

Cuckoo Sandbox license is shipped with Cuckoo and contained in "LICENSE" file inside "docs" folder.

2.1.4 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.

2.2 Installation

This chapter explains how to install Cuckoo.

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.

2.2.1 Preparing the Host

Even though it's reported to run on other operating systems too, Cuckoo is originally supposed to run on a *GNU/Linux* native system. For the purpose of this documentation, we chose **latest Ubuntu LTS** as reference system for the commands examples.

Requirements

Before proceeding on configuring Cuckoo, you'll need to install some required softwares and libraries.

Installing Python libraries

Cuckoo host components are completely written in Python, therefore make sure to have an appropriate version installed. For current release **Python 2.7** is preferred.

Install Python on Ubuntu:

```
$ sudo apt-get install python
```

In order to properly function, Cuckoo requires SQLAlchemy to be installed.

Install with `apt-get`:

```
$ sudo apt-get install python-sqlalchemy
```

Install with `pip`:

```
$ sudo pip install sqlalchemy
```

There are other optional dependencies that are mostly used by modules and utilities. The following libraries are not strictly required, but their installation is recommended:

- Dpkt (Highly Recommended): for extracting relevant information from PCAP files.
- Jinja2 (Highly Recommended): for rendering the HTML reports and the web interface.
- Magic (Optional): for identifying files' formats (otherwise use "file" command line utility)
- Pydeep (Optional): for calculating ssdeep fuzzy hash of files.
- Pymongo (Optional): for storing the results in a MongoDB database.
- Yara and Yara Python (Optional): for matching Yara signatures (use the svn version).
- Libvirt (Optional): for using the KVM machine manager.
- Bottlepy (Optional): for using the `web.py` and `api.py` utilities.
- Pefile (Optional): used for static analysis of PE32 binaries.

Some of them are already packaged in Debian/Ubuntu and can be installed with the following command:

```
$ sudo apt-get install python-dpkt python-jinja2 python-magic python-pymongo python-libvirt python-bo
```

Except for *python-magic* and *python-libvirt*, the others can be installed through `pip` too:

```
$ sudo pip install dpkt jinja2 pymongo bottle pefile
```

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

If want to use KVM it's packaged too and you can install it with the following command:

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

Virtualization Software

Despite heavily relying on VirtualBox in the past, Cuckoo has moved on being architecturally independent from the virtualization software. As you will see throughout this documentation, you'll be able to define and write modules to support any software of your choice.

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

You are completely responsible for the choice, configuration and execution of your virtualization software, therefore please hold from asking help on it in our channels and lists: refer to the software's official documentation and support.

Assuming you decide to go for VirtualBox, you can get the proper package for your distribution at the official download page. The installation of VirtualBox is not in the purpose of this documentation, if you are not familiar with it 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
```

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 last command with:

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

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

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

Installing Cuckoo

Proceed with download and installation.

Create a user

You either can run Cuckoo from your own user or create a new one dedicated just to 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, otherwise Cuckoo won't be able to identify and launch them.

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 useradd -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 useradd -G libvirtd cuckoo
```

Install Cuckoo

Extract or checkout your copy of Cuckoo to a path of your choice and you're ready to go ;-).

Configuration

Cuckoo relies on three main configuration files:

- *cuckoo.conf*: for configuring general behavior and analysis options.
- *<machinemanager>.conf*: for defining the options for your virtualization software.
- *reporting.conf*: for enabling or disabling report formats.

cuckoo.conf

The first file to edit is *conf/cuckoo.conf*, whose content is:

```
[cuckoo]
# Enable or disable startup version check. When enabled, Cuckoo will connect
# to a remote location to verify whether the running version is the latest
# one available.
version_check = on

# If turned on, Cuckoo will delete the original file and will just store a
# copy in the local binaries repository.
delete_original = off

# Specify the name of the machine manager module to use, this module will
# define the interaction between Cuckoo and your virtualization software
# of choice.
machine_manager = virtualbox

# Enable creation of memory dump of the analysis machine before shutting
# down. Even if turned off, this functionality can also be enabled at
# submission. Currently available for: VirtualBox and libvirt modules (KVM).
```

```
memory_dump = off

[processing]
# Set the maximum size of analysis's generated files to process.
# This is used to avoid the processing of big files which can bring memory leak.
# The value is expressed in bytes, by default 100Mb.
analysis_size_limit = 104857600

# Enable or disable DNS lookups.
resolve_dns = on

[database]
# Specify the database connection string.
# Examples, see documentation for more:
# sqlite:///foo.db
# postgresql://foo:bar@localhost:5432/mydatabase
# mysql://foo:bar@localhost/mydatabase
# If empty, default is a SQLite in  db/cuckoo.db.
connection =

# Database connection timeout in seconds.
# If empty, default is set to 60 seconds.
timeout =

[timeouts]
# Set the default analysis timeout expressed in seconds. This value will be
# used to define after how many seconds the analysis will terminate unless
# otherwise specified at submission.
default = 120

# Set the critical timeout expressed in seconds. After this timeout is hit
# Cuckoo will consider the analysis failed and it will shutdown the machine
# no matter what. When this happens the analysis results will most likely
# be lost. Make sure to have a critical timeout greater than the
# default timeout.
critical = 600

# Maximum time to wait for virtual machine status change. For example when
# shutting down a vm. Default is 300 seconds.
vm_state = 300

[sniffer]
# Enable or disable the use of an external sniffer (tcpdump) [yes/no].
enabled = yes

# Specify the path to your local installation of tcpdump. Make sure this
# path is correct.
tcpdump = /usr/sbin/tcpdump

# Specify the network interface name on which tcpdump should monitor the
# traffic. Make sure the interface is active.
interface = vboxnet0

[graylog]
# Enable or disable remote logging to a Graylog2 server.
enabled = no

# Graylog2 server host.
```

```
host = localhost

# Graylog2 server port.
port = 12201

# Default logging level for Graylog2. [debug/info/error/critical].
level = error
```

The configuration file is self-explainatory.

<machinemanager>.conf

Machine managers are the modules that define how Cuckoo should interact with your virtualization software of choice.

Every module should have a dedicated configuration file which defines the details on the available machines. For example, if you created a *vmware.py* machine manager module, you should specify *vmware* in *conf/cuckoo.conf* and have a *conf/vmware.conf* file.

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 *conf/virtualbox.conf* file:

```
[virtualbox]
# Specify which VirtualBox mode you want to run your machines on.
# Can be "gui", "sdl" or "headless". Refer to VirtualBox's official
# documentation to understand the differences.
mode = gui

# Path to the local installation of the VBoxManage utility.
path = /usr/bin/VBoxManage

# 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
# VirtualBox 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 = 192.168.56.101
```

You can use this same configuration structure for any other machine manager module.

The comments for the options are self-explainatory.

reporting.conf

The *conf/reporting.conf* file contains information on the automated reports generation.

It contains the following sections:

```
# Enable or disable the available reporting modules [on/off].
# If you add a custom reporting module to your Cuckoo setup, you have to add
# a dedicated entry in this file, or it won't be executed.
# You can also add additional options under the section of your module and
# they will be available in your Python class.

[jsondump]
enabled = on

[reporthtml]
enabled = on

[pickled]
enabled = off

[metadata]
enabled = off

[maec11]
enabled = off

[mongodb]
enabled = off

[hpfclient]
enabled = off
host =
port = 10000
ident =
secret =
channel =
```

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

2.2.2 Preparing the Guest

At this point you should have configured 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: 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 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 softwares 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 Image Library: it's used for taking screenshots of 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 features available. 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 malwares in, you might want to install additional software such as browsers, PDF readers, office suites etc. Remeber to disable the "auto update" or "check for updates" feature of any additional software.

This is completely up to you and to how you, you might get some hints by reading the Sandboxing chapter.

Network Configuration

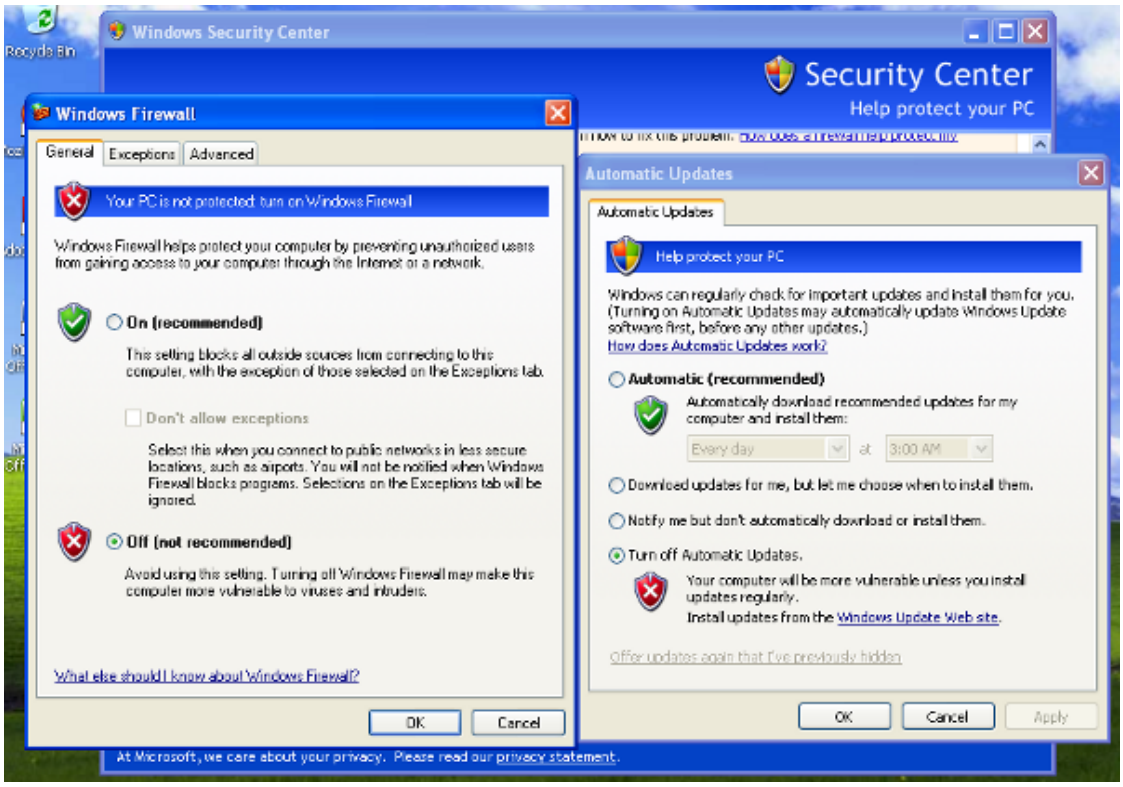
Now it's the time to setup the network configuration for your virtual machine.

Windows Settings

Before configuring the underlying networking of the virtual machine, you might want to trick 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 unrelevant 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. Test network trying to ping a guest is a good practice, to be sure about virtual network setup. Use only static address 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.

> **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. You 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 and filtering configuration done with `iptables` on the Host.

For example, using VirtualBox, you can enable Internet access to the virtual machines using the following `iptables` rules:

```
iptables -A FORWARD -o eth0 -i vboxnet0 -s 192.168.56.0/24 -m conntrack --ctstate NEW -j ACCEPT
iptables -A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A POSTROUTING -t nat -j MASQUERADE
```

And adding IP forward:

```
sysctl -w net.ipv4.ip_forward=1
```

Install 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 as well as on Linux and OS X. In order to make Cuckoo work properly, you'll have to install and start this agent.

It's very simple.

In the *agent/* directory you will find and *agent.py* file, just copy it to the Guest operating system (in whatever way you want, perhaps a temporary shared folder or by downloading it from a Host webserver) and run it. This will launch the XMLRPC server which will be listening for connections.

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 window from spawning.

Saving the Virtual Machine

Now you should be ready to go and 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 fist 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 in the correct way 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 following:

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

Now you have to edit your VM definition like following:

```
$ 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 all works prepare it for snapshotting while running Cuckoo's agent. You can finally take a snapshot with the following command:

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

VMware Workstation

If 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 graphical user interface or from the command line:

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

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 eventually bring along also the settings of the original one, which is not good. Now you need to proceed repeating the steps explained in Network Configuration, Install the Agent and Saving the Virtual Machine for this new machine.

2.2.3 Upgrade from a previous release

Cuckoo Sandbox grows really fast and in every release new features are added and some others are fixed or removed. If not otherwise specified in the release documentation, the suggested way to upgrade your Cuckoo instance is to perform a fresh setup as described in Installation.

The following steps are suggested:

1. Backup your installation.

2. Read the documentation shipped with the new release.
3. Make sure to have installed all required dependencies, otherwise install them.
4. Do a Cuckoo fresh installation of the Host components.
5. Reconfigure Cuckoo as explained in this book (copying old configuration files is not safe because options can change between releases).
6. Test it!

If something goes wrong you probably failed some steps during the fresh installation or reconfiguration. Check again the procedure explained in this book.

It's not recommended to rewrite an old Cuckoo installation with the latest release files, as it might raise some problems because:

- You are overwriting Python source files (.py) but Pyhton bytecode files (.pyc) are still in place.
- There are configuration files changes across the two versions, check our CHANGELOG file for added or removed configuration options.
- The part of Cuckoo which runs inside guests may change.

2.3 Usage

This chapter explains how to use Cuckoo.

2.3.1 Starting Cuckoo

To start Cuckoo use the command:

```
$ python cuckoo.py
```

Make sure to run it inside Cuckoo's root directory.

You will get an output similar to this:

```
                            .:
                            ::
   .-.     ,  :   .-.      ;;.-.  .-.   .-.
  ;       ;   ;  ;        ;; .' ;   ;';   ;'
  `;;;;'.'`..:;._`;;;;'_.'`   `.`;;'  `;;'

 Cuckoo Sandbox 0.5
 www.cuckoosandbox.org
 Copyright (c) 2010-2012

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

2012-12-18 14:56:31,036 [lib.cuckoo.core.scheduler] INFO: Using "virtualbox" machine manager
2012-12-18 14:56:31,861 [lib.cuckoo.core.scheduler] INFO: Loaded 1 machine/s
2012-12-18 14:56:31,862 [lib.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.py` accepts some command line options as shown by the help:

```
usage: cuckoo.py [-h] [-q] [-d] [-v] [-a]

optional arguments:
  -h, --help     show this help message and exit
  -q, --quiet    Display only error messages
  -d, --debug    Display debug messages
  -v, --version  show program's version number and exit
  -a, --artwork  Show artwork
```

Most importantly `--debug` and `--quiet` respectively increase and decrease the logging verbosity.

2.3.2 Submit an Analysis

- *Submission Utility*
- *API*
- *Web Utility*
- *Python Functions*

Submission Utility

The easiest way to submit an analysis is to use the provided *submit.py* command-line utility. It currently has the following options available:

```
usage: submit.py [-h] [--url] [--package PACKAGE] [--custom CUSTOM]
                 [--timeout TIMEOUT] [--options OPTIONS] [--priority PRIORITY]
                 [--machine MACHINE] [--platform PLATFORM] [--memory]
                 [--enforce-timeout]
                 target

positional arguments:
  target                URL, path to the file or folder to analyze

optional arguments:
  -h, --help            show this help message and exit
  --url                 Specify whether the target is an URL
  --package PACKAGE     Specify an analysis package
  --custom CUSTOM       Specify any custom value
  --timeout TIMEOUT     Specify an analysis timeout
  --options OPTIONS     Specify options for the analysis package (e.g.
                        "name=value,name2=value2")
  --priority PRIORITY   Specify a priority for the analysis represented by an
                        integer
  --machine MACHINE     Specify the identifier of a machine you want to use
  --platform PLATFORM   Specify the operating system platform you want to use
                        (windows/darwin/linux)
  --memory              Enable to take a memory dump of the analysis machine
  --enforce-timeout     Enable to force the analysis to run for the full
                        timeout period
```

If you specify a directory as path, all the files contained in it will be submitted for analysis.

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:

```
$ ./utils/submit.py /path/to/binary
```

Example: submit an URL:

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

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

```
$ ./utils/submit.py --priority 5 /path/to/binary
```

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

```
$ ./utils/submit.py --timeout 60 /path/to/binary
```

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

```
$ ./utils/submit.py --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):

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

Example: submit a local binary to be run on virtual machine *cuckoo1*:

```
$ ./utils/submit.py --machine cuckoo1 /path/to/binary
```

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

```
$ ./utils/submit.py --platform windows /path/to/binary
```

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

```
$ ./utils/submit.py --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):

```
$ ./utils/submit.py --enforce-timeout /path/to/binary
```

API

Detailed usage of the REST API interface is described in REST API.

Web Utility

Cuckoo provides a very basic web utility that you can use to submit files to be analyzed.

You can find the script at path *utils/web.py* and you can start it with:

```
$ python utils/web.py
```

By default it will create a webserver on localhost and port 8080. Open your browser at *http://localhost:8080* and it will prompt you a simple form that allows you to upload a file, specify some options (with the same format as the *submit.py* utility) and submit it.

In the *Browse* section you can track the status of pending, failed and succeeded analyses and, when available, you'll be prompted a link to view the HTML report.

Note: This is by no means supposed to be a full fledged web interface: it's a very simple utility that we put together to allow users to simply upload files and consumes the generated HTML report. Despite being incorporated and rendered dynamically, the results displayed are nothing else than the *report.html* file, therefore it is supposed to be independent from the utility.

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, 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 the case you want to write your own Python submission script, you can use the `add_path()` and `add_url()` functions.

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

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
- **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)
- **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 executuion for the full timeout

Return type integer

Example usage:

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

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

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
- **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)
- **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 executuion for the full timeout

Return type integer

Example Usage:

```
>>> from lib.cuckoo.core.database import Database
>>> db = Database()
>>> db.add_url("http://www.cuckoosandbox.org")
2
>>>
```

2.3.3 REST API

As mentioned in Submit an Analysis, Cuckoo provides a simple and lightweight REST API server implemented in Bottle.py, therefore in order to make the service work you'll need it installed.

On Debian/Ubuntu:

```
$ sudo apt-get install python-bottle
```

With Pip:

```
$ pip install bottle
```

Starting the API server

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

```
$ ./utils/api.py
```

By default it will bind the service on **localhost:8090**. If you want to change those values, you can for example with:

```
$ ./utils/api.py --host 0.0.0.0 --port 1337
```

Resources

Following is a list of currently available resources and a brief description. 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.
`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/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` */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` */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.

/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 response:

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

Form parameters:

- `file` *(required)* - path to the file to submit
- `package` *(optional)* - analysis package to be used for the analysis
- `timeout` *(optional) (int)* - priority to assign to the task (1-3)
- `options` *(optional)* - options to pass to the analysis package
- `machine` *(optional)* - ID of the analysis machine to use for the analysis
- `platform` *(optional)* - name of the platform to select the analysis machine from (e.g. “windows”)

- `custom` *(optional)* - custom string to pass over the analysis and the processing/reporting modules
- `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

Status codes:

- `200` - no error

/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 response:

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

Form parameters:

- `url` *(required)* - URL to analyze
- `package` *(optional)* - analysis package to be used for the analysis
- `timeout` *(optional) (int)* - priority to assign to the task (1-3)
- `options` *(optional)* - options to pass to the analysis package
- `machine` *(optional)* - ID of the analysis machine to use for the analysis
- `platform` *(optional)* - name of the platform to select the analysis machine from (e.g. "windows")
- `custom` *(optional)* - custom string to pass over the analysis and the processing/reporting modules
- `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

Status codes:

- `200` - no error

/tasks/list

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

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,
            "priority": 1,
            "platform": null,
            "options": null,
            "status": "pending",
            "enforce_timeout": false,
            "timeout": 0,
            "memory": false,
            "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,
            "priority": 1,
            "platform": null,
            "options": null,
            "status": "pending",
            "enforce_timeout": false,
            "timeout": 0,
            "memory": false,
            "id": 2,
            "added_on": "2012-12-19 14:18:25",
            "completed_on": null
        }
    ]
}
```


Parameters:

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

Status codes:

- `200` - no error

/tasks/view

GET /tasks/list/ *(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,
            "priority": 1,
            "platform": null,
            "options": null,
            "status": "pending",
            "enforce_timeout": false,
            "timeout": 0,
            "memory": false,
            "id": 1,
            "added_on": "2012-12-19 14:18:25",
            "completed_on": null
        }
    ]
}
```


Parameters:

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

Status codes:

- `200` - no error
- `404` - task not found

/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/maec/metadata/picke]. If none is specified the JSON report will be returned

Status codes:

- `200` - no error
- `400` - invalid report format
- `404` - report 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": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
        "sha512": "cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d13c5
        "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/e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991
```

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",
            "ip": "192.168.56.101",
            "label": "cuckoo1",
            "locked_changed_on": null,
            "platform": "windows",
            "status_changed_on": null,
            "id": 1
        }
    ]
}
```

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",
                    "ip": "192.168.56.101",
                    "label": "cuckoo1",
                    "locked_changed_on": null,
                    "platform": "windows",
                    "status_changed_on": null,
                    "id": 1
                }
            ]
        }
```

Status codes:

- `200` - no error
- `404` - machine not found

2.3.4 Analysis Packages

The **analysis packages** are a core component of Cuckoo Sandbox. They consist in structured Python classes which, 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 eventually modify the existing ones. You can find them located 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 the list of existing packages in alphabetical order:

- `applet`: used to analyze **Java applets**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `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**.
- `dll`: used to run and analyze **Dinamically Linked Libraries**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `function`: specify the function to be executed. If none is specified, Cuckoo will try to run `DllMain`.

- `doc`: used to run and analyze **Microsoft Word documents**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.

- `exe`: default analysis package used to analyze generic **Windows executables**.

Options:

- `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
- `arguments`: specify any command line argument to pass to the initial process of the submitted malware.

- `ie`: used to analyze **Internet Explorer**'s behavior when opening the given URL.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.

- `jar`: used to analyze **Java JAR** containers.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `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.

- `pdf`: used to run and analyze **PDF documents**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.

- `xls`: used to run and analyze **Microsoft Excel documents**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.

- `zip`: used to run and analyze **Zip archives**. The archive must be either password-less or the password must be "*infected*".

  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*.
  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `arguments`: specify any command line argument to pass to the initial process of the submitted malware.

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) like following:

```
$ ./utils/submit.py --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 you are always invited to specify the package name whenever it's possible.

For example, to launch a malware and specify some options you can do:

```
$ ./utils/submit.py --package dll --options function=FunctionName /path/to/malware.dll
```

2.3.5 Analysis Results

Once an analysis is completed, several files are stored in a dedicated directory. All the analysis are stored under the directory *storage/analyses/* inside a subdirectory named with the incremental numerical ID which represents the analysis task inside the database.

Following is an example of an analysis directory structure:

```
.
|-- analysis.conf
|-- analysis.log
|-- binary
|-- dump.pcap
|-- memory.dmp
|-- files
|   |-- 1234567890
|       `-- dropped.exe.bin
|-- logs
|   |-- 1232.csv
|   |-- 1540.csv
|   `-- 1118.csv
|-- reports
|   |-- report.html
|   |-- report.json
|   |-- report.maec11.xml
|   |-- report.metadata.xml
|   `-- report.pickle
`-- shots
    |-- 0001.jpg
    |-- 0002.jpg
    |-- 0003.jpg
    `-- 0004.jpg
```

analysis.conf

This is a configuration file automatically generated by Cuckoo to instruct its analyzer some details about the current analysis. It's generally of no interest for the end-user, as it's exclusively used internally by the sandbox.

analysis.log

This is a log file generated by the analyzer and that contains a trace of the analysis execution inside the guest environment. It will report the creation of processes, files and eventual error occurred during the execution.

dump.pcap

This is the network dump generated by tcpdump or any other corresponding network sniffer.

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.

logs/

This directory contains all the raw logs generated by Cuckoo's process monitoring They are named *<process id>.csv* and contain the monitored API calls in chronological order represented in a csv-like format.

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.

2.3.6 Utilities

Cuckoo comes with a set of pre-built utilities to automatize several common tasks. You can find them in "utils" folder.

Cleanup utility

If you want to delete all history, analysis, data and begin again from the first task you need clean.sh utility.

Note: Running clean.sh will delete: analysis results, binaries, SQLite database (if used) and logs.

To clean your setup, run:

```
$ ./utils/clean.sh
```

If you are using a custom database (MySQL, PostgreSQL or SQLite in custom location) clean.sh doesn't clean it, you have to take care of that.

Submission Utility

Submits sample to analysis. This tool is already described in Submit an Analysis.

Web Utility

Cuckoo's web interface. This tool is already described in Submit an Analysis.

Processing Utility

Run the results processing engine and optionally the reporting engine (run all reports) on an already available analysis folder, in order to not re-run the analysis if you want to re-generate the reports for it. This is used mainly in debugging and developing Cuckoo. For example if you want run again the report engine for analysis number 1:

```
$ ./utils/process.py storage/analyses/1/
```

If you want to re-generate the reports:

```
$ ./utils/process.py --report storage/analyses/1/
```

Community Download Utility

This utility downloads signatures from Cuckoo Community Repository and installs specific additional modules in your local setup and for example update id with all the latest available signatures. Following are the usage options:

```
$ ./utils/community.py

usage: community.py [-h] [-a] [-s] [-p] [-m] [-r] [-f] [-w]

optional arguments:
  -h, --help            show this help message and exit
  -a, --all             Download everything
  -s, --signatures      Download Cuckoo signatures
  -p, --processing      Download processing modules
  -m, --machinemanagers
                        Download machine managers
  -r, --reporting       Download reporting modules
  -f, --force           Install files without confirmation
  -w, --rewrite         Rewrite existing files
```

Example: install all available signatures:

```
$ ./utils/community.py --signatures --force
```

2.4 Customization

This chapter explains how to customize Cuckoo. Cuckoo is written in a modular architecture built to be as much customizable it can, to fit all user's needs.

2.4.1 Machine Managers

Machine managers are modules that 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 and, in case is not supported by default, write a custom Python module that define how to make Cuckoo use it.

Every machine manager module is and should be located inside *modules/machinemanagers/*.

A basic machine manager could look like:

```
1  from lib.cuckoo.common.abstracts import MachineManager
2  from lib.cuckoo.common.exceptions import CuckooMachineError
3
4  class MyManager(MachineManager):
5      def start(self, label):
6          try:
7              revert(label)
8              start(label)
```

```
9           except SomethingBadHappens as e:
10              raise CuckooMachineError("OPS!")
11
12      def stop(self, label):
13          try:
14              stop(label)
15          except SomethingBadHappens as e:
16              raise CuckooMachineError("OPS!")
```

The only requirements for Cuckoo are that:

- The class inherits `MachineManager`.
- You have a `start()` and `stop()` functions.
- You preferably raise `CuckooMachineError` when something fails.

As you understand, the machine manager 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 machine manager module should come with a dedicated configuration file located in *conf/<machine manager name>.conf*. For example for *modules/machinemanagers/kvm.py* we have a *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 it's `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 it's original code in the `MachineManager` abstract inside *lib/cuckoo/common/abstracts.py*.

LibVirt

Starting with Cuckoo 0.5 developing new machine managers based on LibVirt is easy. Inside *lib/cuckoo/common/abstracts.py* you can find `LibVirtMachineManager` that already provides all the functionalities for a LibVirt machine manager. Just inherit this base class and specify your connection string, as in the example below:

```
from lib.cuckoo.common.abstracts import LibVirtMachineManager

        class MyMachineManager(LibVirtMachineManager):
            # Set connection string.
            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.

2.4.2 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 *analyzer/windows/packages/exe.py*):

```
from lib.common.abstracts import Package
from lib.api.process import Process
from lib.common.exceptions import CuckooPackageError

class Exe(Package):
    """EXE analysis package."""

    def start(self, path):
        free = self.options.get("free", False)
        args = self.options.get("arguments", None)
        suspended = True
        if free:
```

```
13              suspended = False
14
15          p = Process()
16          if not p.execute(path=path, args=args, suspended=suspended):
17              raise CuckooPackageError("Unable to execute initial process, analysis aborted")
18
19          if not free and suspended:
20              p.inject()
21              p.resume()
22              return p.pid
23          else:
24              return None
25
26      def check(self):
27          return True
28
29      def finish(self):
30          return True
```

Let's walk through the code:

- Line **1**: import the base `Package` class, it's needed to define our analysis package class.
- Line **2**: import the `Process` API class, which is used to create and manipulate Windows processes.
- Line **3**: import the `CuckooPackageError` exception, which is used to notify issues with the execution of the package to the analyzer.
- Line **5**: define the main class, inheriting `Package`.
- Line **8**: define the `start()` function, which takes as argument the path to the file to execute.
- Line **9**: acquire the `free` option, which is used to define whether the process should be monitored or not.
- Line **10**: acquire the `arguments` option, which is passed to the creation of the initial process.
- Line **15**: initialize a `Process` instance.
- Line **16** and **17**: try to execute the malware, if it fails it aborts the execution and notify the analyzer.
- Line **19**: check if the process should be monitored.
- Line **20**: inject the process with our DLL.
- Line **21**: resume the process from the suspended state.
- Line **22**: return the PID of the newly created process to the analyzer.
- Line **26**: define the `check()` function.
- Line **29**: define the `finish()` function.

start()

In this function you have to place all the initialization operations you want to run. This might 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 straight 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.

finish()

This function is simply called by Cuckoo before terminating the analysis and powering off the machine. There's no predefined use for this function and it's not going to affect Cuckoo's execution whatsoever, so you could simply use it to perform any last operation on the system.

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:

```
p = Process(pid=1234)
p.open()
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:

```
p = Process(pid=1234)
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:

```
p = Process(pid=1234)
if p.is_alive():
    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:

```
p = Process(pid=1234)
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:

```
p = Process()
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:

```
p = Process()
p.execute(path="C:\\WINDOWS\\system32\\calc.exe", args="Something", suspended=True)
p.resume()
```

`Process.`**`terminate`**`()`

Terminates the opened process. Returns `True` or `False` in case of success or failure of the operaton.

Return type boolean

Example Usage:

```
p = Process(pid=1234)
if p.terminate():
    print("Process terminated!")
else:
    print("Could not terminate the process!")
```

`Process.`**`inject`**([*dll*[, *apc=False*]])

Injects a DLL (by default "dll/cuckoomon.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 istead of `CreateRemoteThread()`, beware that if the process is in suspended mode, Cuckoo will always use `QueueUserAPC()`

Return type boolean

Example Usage:

```
p = Process()
p.execute(path="C:\\WINDOWS\\system32\\calc.exe", args="Something", suspended=True)
p.inject()
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:

```
p = Process(pid=1234)
p.dump_memory()
```

2.4.3 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 *modules/processing/* directory. Any additional module you decide to create, must be placed inside that directory.

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** *(modules/processing/analysisinfo.py)* - generates some basic information on the current analysis, such as timestamps, version of Cuckoo and so on.
- **BehaviorAnalysis** *(modules/processing/behavior.py)* - parses the raw behavioral logs and perform some initial trasnformations and interpretations, including the complete processes tracing, a behavioral summary and a process tree.
- **Debug** *(modules/processing/debug.py)* - includes errors and the *analysis.log* generated by the analyzer.
- **Dropped** *(modules/processing/dropped.py)* - includes information on the files dropped by the malware and dumped by Cuckoo.
- **NetworkAnalysis** *(modules/processing/network.py)* - prases the PCAP file and extract some network information, such as DNS traffic, domains, IPs, HTTP requests, IRC and SMTP traffic.
- **StaticAnalysis** *(modules/processing/static.py)* - performs some static analysis of PE32 files.
- **Strings** *(modules/processing/static.py)* - extracts strings from the analyzer binary.
- **TargetInfo** *(modules/processing/targetinfo.py)* - includes information on the analyzed file, such as hashes.

- **VirusTotal** *(modules/processing/virustotal.py)* - lookup 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.
- **YaraSignatures** *(modules/processing/yarasignatures.py)* - matches the Yara signatures available under *data/yara/* against the analyzed file.

Getting started

In order to make them available to Cuckoo, all processing modules are and should be placed inside the folder at *modules/processing/*.

A basic processing module could look like:

```
from lib.cuckoo.common.abstracts import Processing

class MyModule(Processing):

    def run(self):
        self.key = "key"
        data = do_something()
        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 subcontainer for the returned data.
- A set of data (list, dictionary or 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:

```
from lib.cuckoo.common.abstracts import Processing

class MyModule(Processing):
    order = 2

    def run(self):
        self.key = "key"
        data = do_something()
        return data
```

You can also manually disable a processing module by setting the `enabled` attribute to `False`:

```
from lib.cuckoo.common.abstracts import Processing

class MyModule(Processing):
    enabled = False

    def run(self):
        self.key = "key"
        data = do_something()
        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. *storage/analysis/1*)
- `self.log_path`: path to the *analysis.log* file.
- `self.conf_path`: path to the *analysis.conf* 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.

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 following:

```
from lib.cuckoo.common.exceptions import CuckooProcessingError
from lib.cuckoo.common.abstracts import Processing

class MyModule(Processing):

    def run(self):
        self.key = "key"

        try:
            data = do_something()
        except SomethingFailed:
            raise CuckooProcessingError("Failed")

        return data
```

2.4.4 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 malwares of interest.

Some examples 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.

You can find signatures created by us and by other Cuckoo users on our Community repository.

Getting started

Creation of signatures is a very simple process and requires just a decent understanding of Python programming.

First thing first, all signatures are and should be located inside *modules/signatures/*.

A basic example signature is the following:

```
1  from lib.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 = "0.5"
10
11     def run(self):
12         return self.check_file(pattern=".*\\.exe$",
13                                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 as you can see at line **12** from version 0.5 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`.

In case the signature gets matched, a new entry in the "signatures" section will be added to the global container like following:

```
"signatures": [
    {
        "severity": 2,
        "description": "Creates a Windows executable on the filesystem",
        "alert": false,
        "references": [],
        "data": [
            {
                "file_name": "C:\\d.exe"
            }
        ],
        "name": "creates_exe"
    }
]
```

We could rewrite the exact same signature by accessing the **global container** directly:

```
1  from lib.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 = "0.5"
10
11     def run(self):
```

```
        for file_path in self.results["behavior"]["summary"]["files"]:
            if file_path.endswith(".exe"):
                return True

        return False
```

This obviously requires you to know the structure of the **global container**, which you can observe represented in the JSON report of your analyses.

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 going to simply 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:

```
from lib.cuckoo.common.abstracts import Signature

class BadBadMalware(Signature): # We initialize the class inheriting Signature.
    name = "badbadmalware" # We define the name of the signature
    description = "Creates a mutex known to be associated with Win32.BadBadMalware" # We provide
    severity = 3 # We set the severity to maximum
    categories = ["trojan"] # We add a category
    families = ["badbadmalware"] # We add the name of our fictional malware family
    authors = ["Me"] # We specify the author
    minimum = "0.5" # We specify that in order to run the signature, the user will need at least

def run(self):
    return
```

This is a perfectly valid signature. It doesn't really do anything as of yet, now we need to define the conditions for the signature to be matched.

As we said, we want to match a pecurial mutex name, so we proceed as follows:

```
from lib.cuckoo.common.abstracts import Signature

class BadBadMalware(Signature):
    name = "badbadmalware"
    description = "Creates a mutex known to be associated with Win32.BadBadMalware"
    severity = 3
    categories = ["trojan"]
    families = ["badbadmalware"]
    authors = ["Me"]
    minimum = "0.5"

def run(self):
    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:

```
from lib.cuckoo.common.abstracts import Signature

class BadBadMalware(Signature):
    name = "badbadmalware"
    description = "Creates a mutex known to be associated with Win32.BadBadMalware"
    severity = 3
    categories = ["trojan"]
    families = ["badbadmalware"]
    authors = ["Me"]
    minimum = "0.5"

def run(self):
    for mutex in self.results["behavior"]["summary"]["mutexes"]:
        if mutex == "i_am_a_malware":
            return True

    return False
```

Helpers

As anticipated, from version 0.5 the `Signature` base class also provides some helper methods that simplify the creation of signatures and avoid you from directly accessing the global container (at least most of the times).

Following is a list of available methods.

`Signature.`**`check_file`**(*pattern*[, *regex=False*])

Checks whether the malware opened or created a file matching the specified pattern. Returns `True` in case it did, otherwise returns `False`.

Parameters

- **`pattern`** (*string*) – file name or file path pattern to be matched
- **`regex`** (*boolean*) – enable to compile the pattern as a regular expression

Return type boolean

Example Usage:

```
self.check_file(pattern=".*\.exe$", regex=True)
```

Signature.**check_key**(*pattern*[, *regex=False*])
Checks whether the malware opened or created a registry key matching the specified pattern. Returns `True` in case it did, otherwise returns `False`.

Parameters

- **pattern** (*string*) – registry key pattern to be matched
- **regex** (*boolean*) – enable to compile the pattern as a regular expression

Return type boolean

Example Usage:

```
self.check_key(pattern=".*CurrentVersion\\Run$", regex=True)
```

Signature.**check_mutex**(*pattern*[, *regex=False*])
Checks whether the malware opened or created a mutex matching the specified pattern. Returns `True` in case it did, otherwise returns `False`.

Parameters

- **pattern** (*string*) – mutex pattern to be matched
- **regex** (*boolean*) – enable to compile the pattern as a regular expression

Return type boolean

Example Usage:

```
self.check_mutex("mutex_name")
```

Signature.**check_api**(*pattern*[, *process=None*[, *regex=False*]])
Checks whether Windows function was invoked. Returns `True` in case it was, otherwise returns `False`.

Parameters

- **pattern** (*string*) – function name pattern to be matched
- **process** (*string*) – name of the process performing the call
- **regex** (*boolean*) – enable to compile the pattern as a regular expression

Return type boolean

Example Usage:

```
self.check_api(pattern="URLDownloadToFileW", process="AcroRd32.exe")
```

Signature.**check_argument**(*pattern[, name=Name[, api=None[, category=None[, process=None[, regex=False]]]]]*)
Checks whether the malware invoked a function with a specific argument value. Returns `True` in case it did, otherwise returns `False`.

Parameters

- **pattern** (*string*) – argument value pattern to be matched
- **name** (*string*) – name of the argument to be matched
- **api** (*string*) – name of the Windows function associated with the argument value
- **category** (*string*) – name of the category of the function to be matched
- **process** (*string*) – name of the process performing the associated call

- **regex** (*boolean*) – enable to compile the pattern as a regular expression

Return type boolean

Example Usage:

```
self.check_argument(pattern=".*cuckoo.*", category="filesystem", regex=True)
```

`Signature.`**`check_ip`**(*pattern*[, *regex=False*])

Checks whether the malware contacted the specified IP address. Returns `True` in case it did, otherwise returns `False`.

Parameters

- **pattern** (*string*) – IP address to be matched
- **regex** (*boolean*) – enable to compile the pattern as a regular expression

Return type boolean

Example Usage:

```
self.check_ip("123.123.123.123")
```

`Signature.`**`check_domain`**(*pattern*[, *regex=False*])

Checks whether the malware contacted the specified domain. Returns `True` in case it did, otherwise returns `False`.

Parameters

- **pattern** (*string*) – domain name to be matched
- **regex** (*boolean*) – enable to compile the pattern as a regular expression

Return type boolean

Example Usage:

```
self.check_domain(pattern=".*cuckoosandbox.org$", regex=True)
```

`Signature.`**`check_url`**(*pattern*[, *regex=False*])

Checks whether the malware performed an HTTP request to the specified URL. Returns `True` in case it did, otherwise returns `False`.

Parameters

- **pattern** (*string*) – URL pattern to be matched
- **regex** (*boolean*) – enable to compile the pattern as a regular expression

Return type boolean

Example Usage:

```
self.check_url(pattern="^.+\/load\.php\?file=[0-9a-zA-Z]+$", regex=True)
```

2.4.5 Reporting Modules

After the analysis raw 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 some use of it and will make it accessible and consumable in different formats.

Getting Started

All reporting modules are and should be placed inside the directory *modules/reporting/*.

Every module should also have a dedicated section in the file *conf/reporting.conf*: for example if you create a module *module/reporting/foobar.py* you will have to append the following section to *conf/reporting.conf*:

```
[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 lib.cuckoo.common.abstracts import Report
6  from lib.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 basically just 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 `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.conf_path`: path to the *analysis.conf* file of the current analysis (e.g. *storage/analyses/1/analysis.conf*)
- `self.options`: a dictionary containing all the options specified in the report's configuration section in *conf/reporting.conf*.

2.5 Development

This chapter explains how to write Cuckoo's code and how to contribute.

2.5.1 Development Notes

Git branches

Cuckoo Sandbox source code is available in our official git repository. You'll find multiple branches which are used for different stages of our development lifecycle.

- **Development**: This is where our developers commit their ongoing work for the upcoming releases. As a development branch, this can be really unstable and sometimes even broken and not usable. Users are discouraged to adopt this branch, this is aimed only to developers or guys with a deep knowledge into our technologies.
- **Testing**: When work on development branch is in a usable state and some new features or fixes are completed, the development branch in merged into testing. This is the branch where users can get a taste of the next release. If you want to be always up-to-date this branch is for you.
- **Stable**: When unstable branch is widely tested and bugs free and if all planned features has been completed, a new stable version will be released and available here.

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

We use a ticketing system to track all Cuckoo's developments, bugs and features. Official tracking system is public available, if you want to contribute to Cuckoo please take a look to it before reporting bugs or asking for new features.

Contribute

To submit your patch just create a Pull Request from yuor GitHub fork. If you don't now how to create a Pull Request take a look to GitHub help.

2.5.2 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, and not compliant code will most likely be rejected.

Essentially Cuckoo's code style is based on PEP 8 - Style Guide for Python Code and PEP 257 – Docstring Conventions.

Formatting

Copyright header

All source code files must start with the following copyright header:

```
# Copyright (C) 2010-2012 Cuckoo Sandbox Developers.
# 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 separeted 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 Cucoko.

In your module add:

```
import logging
log = logging.getLogger(__name__)
```

And use the `log` handle, refer to Python's documentation.

In case you really need to print a string to standard output, use the `print()` function:

```
print("foo")
```

NOT the statement:

```
print "foo"
```

Checking for keys in data structures

When checking for a key in a data structure use the clause "in" instead of methods like "has_key()", for example:

```
if "bar" in foo:
    do_something(foo["bar"])
```

Exceptions

Custom exceptions must be defined in the *lib/cuckoo/common/exceptions.py* file or in the local module if the exception should not be global.

Following is current Cuckoo's 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 exceptions name must prefix 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", as in the example above.

Documentation

All code must be documented in docstring format, see PEP 257 – Docstring Conventions. Additional comments may be added in logical blocks will be results hard to understand.

Automated testing

We belive 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 adopt Nose as unit testing framework.

2.6 Final Remarks

2.6.1 Links

- www.cuckoosandbox.org
- community.cuckoosandbox.org
- github.com/cuckoobox
- www.malwr.com

2.6.2 Join the discussion

You can get in contact with Cuckoo's developers and users through the official mailing list kindly provided by The Honeynet Project or on IRC at the official #cuckoosandbox channel.

Our mailing list is mostly intended for development discussions and sharing of ideas and bug reports. 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 mailing list archives, read the Cuckoo blog, read the documentation and Google about your issue. Stop posting questions that have already been answered over and over everywhere.
- Posting messages saying just like "Doesn't work, help me" are completely useless. If something is not working report the error, paste the logs, paste the config file, paste the information on the virtual machine, paste the results of the troubleshooting, give context. We are not wizards and we don't have the crystal ball.
- Use a proper title. Stuff like "Doesn't work", "Help me", "Error" is not a proper title.
- Tend to use pastebin.com or pastie.org and similar services to paste logs and configs: make the message more readable.
- **The community website uses Markdown syntax**. So please read Markdown documentation before posting.

2.6.3 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 other sort. If you're interested in cooperating with us, feel free to contact us.

2.6.4 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	Lead Developer	`nex at cuckoobox dot org`
Alessandro jekil Tanasi	Developer	`alessandro at tanasi dot it`
Jurriaan skier Bremer	Developer	`jurriaanbremer at gmail dot com`
Mark rep Schloesser	Developer	`ms at mwcollect dot org`

Contributors

- Thorsten Sick
- Adam Pridgen
- Mike Tu
- Loic Jaquemet
- Pietro Delsante

- JoseMi Holguin

2.6.5 Supporters

- The Honeynet Project
- The Shadowserver Foundation

2.6.6 Sponsors

- Rapid7

A

P

S