Cuckoo Sandbox Book

Release 2.0-rc1

Cuckoo Sandbox

January 11, 2016

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

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

CHAPTER

ONE

HAVING TROUBLES?

If you’re having troubles you might want to check out the *FAQ* as it may already have the answers to your questions.

1.1 FAQ

Frequently Asked Questions:

- *Can I analyze URLs with Cuckoo?*
- *Can I use Volatility with Cuckoo?*
- *What I need to use Cuckoo with VMware ESXi?*
- *After upgrade Cuckoo stops to work*
- *Cuckoo stumbles and produces some error I don’t understand*

1.1.1 General Questions

Can I analyze URLs with Cuckoo?

Yes you can. Since version 0.5 URLs are natively supported by Cuckoo.

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 operations, the results of a forensic analysis would be polluted by the sandbox’s components.

What I need to use Cuckoo with VMware ESXi?

To run with VMware vSphere Hypervisor (or ESXi) Cuckoo levareges on libvirt. Libivirt is currently using VMware API to take control over virtual machines, althogh these API are available ony in licensed version. In VMware vSphere free edition, these API are read only, so you are unable to use Cuckoo with it. For the minimum license needed, please have a look at VMware website.

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 and produces some error I don't understand

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

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

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

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

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

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

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

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

Check and restore current snapshot with KVM

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

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

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

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

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

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

Choose one snapshot name and set it as current:

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

Now the virtual machine state is fixed.

Check and restore current snapshot with VirtualBox

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

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

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

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

With the following check the current snapshots state:

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

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

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

Unable to bind result server error

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

```
2014-01-07 18:42:12,686 [root] CRITICAL: CuckooCriticalError: Unable to bind result server on 192.168
```

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

Otherwise you can ask the developers and/or other Cuckoo users, see *Join the discussion*.

CHAPTER

TWO

CONTENTS

2.1 Introduction

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

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 on what it does.

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

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

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

Using a Sandbox

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

Some questions you should ask yourself:

- What kind of files do I want to analyze?
- What volume of analyses do I want to be able to handle?
- Which platform do I want to use to run my analysis on?
- What kind of information I want about the file?

The creation of the isolated environment (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 software to install and which versions (particularly important when analyzing exploits).

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

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

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

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

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 functionality.

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

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

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

In March 2012 Cuckoo Sandbox wins the first round of the Magnificent7 program organized by Rapid7.

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

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

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

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

On 1st August 2013 *Claudio "nex" Guarnieri*, *Jurriaan "skier" Bremer* and *Mark "rep" Schloesser* presented Mo' Malware Mo' Problems - Cuckoo Sandbox to the rescue at Black Hat Las Vegas.

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

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

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

Use Cases

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

It can be used to analyze:

- Generic Windows executables
- DLL files
- PDF documents
- Microsoft Office documents
- URLs and HTML files
- PHP scripts
- CPL files
- Visual Basic (VB) scripts
- ZIP files
- Java JAR
- Python files
- *Almost anything else*

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

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

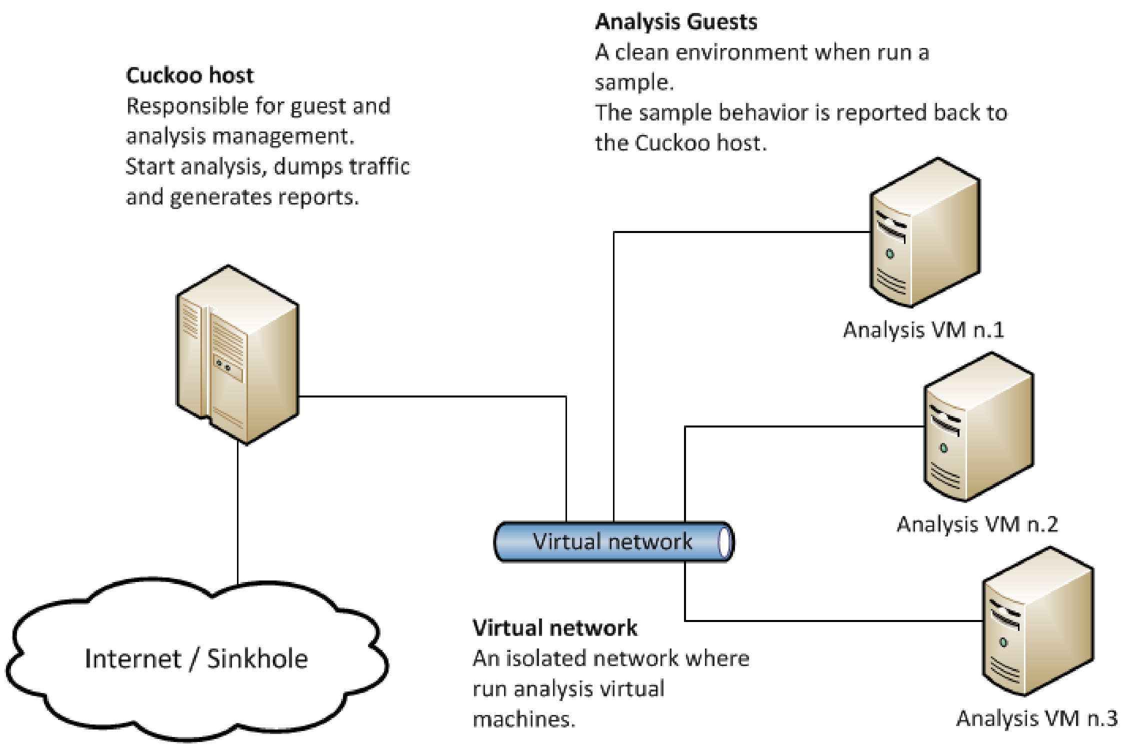
Architecture

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 malware samples get actually safely executed and analyzed.

The following picture explains Cuckoo's main architecture:


Analysis Guests
A clean environment when run a sample.
The sample behavior is reported back to the Cuckoo host.
Cuckoo host
Responsible for guest and analysis management.
Start analysis, dumps traffic and generates reports.
Analysis VM n.1
Analysis VM n.2
Virtual network
Internet / Sinkhole
Virtual network
An isolated network where run analysis virtual machines.
Analysis VM n.3

Although the recommended setup is *GNU/Linux* (Ubuntu preferably) as host and *Windows XP Service Pack 3* as guest, Cuckoo has 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 the "LICENSE" file inside the "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.

2.1.5 Cuckoo Foundation

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

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

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 software and libraries.

Installing Python libraries

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

Install the basic dependencies:

```
$ sudo apt-get install python python-pip python-dev libffi-dev libssl-dev
```

If you want to use the Django-based web interface, you'll have to install MongoDB too:

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

In order to properly function, Cuckoo requires some dependencies. They can all be installed through PyPI like this:

```
$ sudo pip install -r requirements.txt
```

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

If you want to use KVM 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 python-libvirt
```

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

```
$ sudo pip install XenAPI
```

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

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 refrain from asking for 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 outside the scope 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
```

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

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

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

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

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

Installing Volatility

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

In order to function properly, Cuckoo requires at least version 2.3 of Volatility. You can get it from the official repository.

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

Installing Cuckoo

Proceed with download and installation. Read *What is Cuckoo?* to learn where you can obtain a copy of the sandbox.

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 usermod -a -G vboxusers cuckoo
```

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

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

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 six main configuration files:

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

To get Cuckoo working you have to edit *auxiliary.conf*:, *cuckoo.conf* and *<machinery>.conf* at least.

cuckoo.conf

The first file to edit is *conf/cuckoo.conf*, it contains the generic configuration options that you might want to verify before launching Cuckoo.

The file is largely commented and self-explaining, but some of the options you might want to pay more attention to are:

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

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

auxiliary.conf

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

Following is the default *conf/auxiliary.conf* file:

```
[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

# Specify a Berkeley packet filter to pass to tcpdump.
# bpf = not arp
```

<machinery>.conf

Machinery modules are scripts 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* machinery 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 virtual 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

# (Optional) Specify the snapshot name to use. If you do not specify a snapshot
# name, the VirtualBox MachineManager will use the current snapshot.
# Example (Snapshot1 is the snapshot name):
# snapshot = Snapshot1

# (Optional) Specify the name of the network interface that should be used
# when dumping network traffic from this machine with tcpdump. If specified,
# overrides the default interface specified in cuckoo.conf
# Example (virbr0 is the interface name):
# interface = virbr0

# (Optional) Specify the IP of the Result Server, as your virtual machine sees it.
# The Result Server will always bind to the address and port specified in cuckoo.conf,
# however you could set up your virtual network to use NAT/PAT, so you can specify here
# the IP address for the Result Server as your machine sees it. If you don't specify an
# address here, the machine will use the default value from cuckoo.conf.
# Example:
# resultserver_ip = 192.168.56.1

# (Optional) Specify the port for the Result Server, as your virtual machine sees it.
# The Result Server will always bind to the address and port specified in cuckoo.conf,
# however you could set up your virtual network to use NAT/PAT, so you can specify here
# the port for the Result Server as your machine sees it. If you don't specify a port
# here, the machine will use the default value from cuckoo.conf.
# Example:
# resultserver_port = 2042

# (Optional) Set your own tags. These are comma separated and help to identify
# specific VMs. You can run samples on VMs with tag you require.
# tags = windows_xp_sp3,32_bit,acrobat_reader_6
```

You can use this same configuration structure for any other machinery module, although existing ones might have some variations or additional configuration options.

The comments for the options are self-explainatory.

Following is the default *conf/kvm.conf* file:

```
[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 virtual 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. You may want to configure your network settings in
# /etc/libvirt/<hypervisor>/networks/
ip = 192.168.122.105

# (Optional) Specify the snapshot name to use. If you do not specify a snapshot
# name, the KVM MachineManager will use the current snapshot.
# Example (Snapshot1 is the snapshot name):
# snapshot = Snapshot1

# (Optional) Specify the name of the network interface that should be used
# when dumping network traffic from this machine with tcpdump. If specified,
# overrides the default interface specified in cuckoo.conf
# Example (virbr0 is the interface name):
# interface = virbr0

# (Optional) Specify the IP of the Result Server, as your virtual machine sees it.
# The Result Server will always bind to the address and port specified in cuckoo.conf,
# however you could set up your virtual network to use NAT/PAT, so you can specify here
# the IP address for the Result Server as your machine sees it. If you don't specify an
# address here, the machine will use the default value from cuckoo.conf.
# Example:
# resultserver_ip = 192.168.122.101

# (Optional) Specify the port for the Result Server, as your virtual machine sees it.
# The Result Server will always bind to the address and port specified in cuckoo.conf,
# however you could set up your virtual network to use NAT/PAT, so you can specify here
# the port for the Result Server as your machine sees it. If you don't specify a port
# here, the machine will use the default value from cuckoo.conf.
# Example:
# resultserver_port = 2042

# (Optional) Set your own tags. These are comma separated and help to identify
# specific VMs. You can run samples on VMs with tag you require.
# tags = windows_xp_sp3,32_bit,acrobat_reader_6
```

memory.conf

The Volatility tool offers a large set of plugins for memory dump analysis. Some of them are quite slow. In volatility.conf lets you to enable or disable the plugins of your choice. To use Volatility you have to follow two steps:

- Enable it before in processing.conf
- Enable memory_dump in cuckoo.conf

In the memory.conf's basic section you can configure the Volatility profile and the deletion of memory dumps after processing:

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

After that every plugin has an own section for configuration:

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

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

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

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

```
# Masks. Data that should not be logged
# Just get this information from your plain VM Snapshot (without running malware)
# This will filter out unwanted information in the logs
[mask]
# pid_generic: a list of process ids that already existed on the machine before the malware was star
pid_generic = 4, 680, 752, 776, 828, 840, 1000, 1052, 1168, 1364, 1428, 1476, 1808, 452, 580, 652, 24
```

processing.conf

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

You will find a section for each processing module:

```
# Enable or disable the available processing modules [on/off].
# If you add a custom processing 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.
```

```
[analysisinfo]
enabled = yes

[behavior]
enabled = yes

[debug]
enabled = yes

[dropped]
enabled = yes

[memory]
enabled = no

[network]
enabled = yes

[procmemory]
enabled = yes

[static]
enabled = yes

[strings]
enabled = yes

[targetinfo]
enabled = yes

[virustotal]
enabled = yes
# Add your VirusTotal API key here. The default API key, kindly provided
# by the VirusTotal team, should enable you with a sufficient throughput
# and while being shared with all our users, it shouldn't affect your use.
key = a0283a2c3d55728300d064874239b5346fb991317e8449fe43c902879d758088
```

You might want to configure the VirusTotal key if you have an account of your own.

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 = yes

[reporthtml]
enabled = yes

[mongodb]
```

```
enabled = no
host = 127.0.0.1
port = 27017
```

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

Configuration (Android Analysis)

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

avd.conf

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

The file is largely commented and self-explaining, but some of the options you might want to pay more attention to are:

- `emulator_path`: this defines the Android emulator path (it is located inside Android SDK)
- `adb_path`: this defines the ADB path (it is located inside Android SDK)
- `avd_path`: this defines where AVD images are located

2.2.2 Preparing the Guest

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

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

Creation of the Virtual Machine

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

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

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

Note: 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.

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

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

Requirements

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

Install Python

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

You can download the proper Windows installer from the official website. Also in this case Python 2.7 is preferred.

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

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

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

Additional Software

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

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

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

Network Configuration

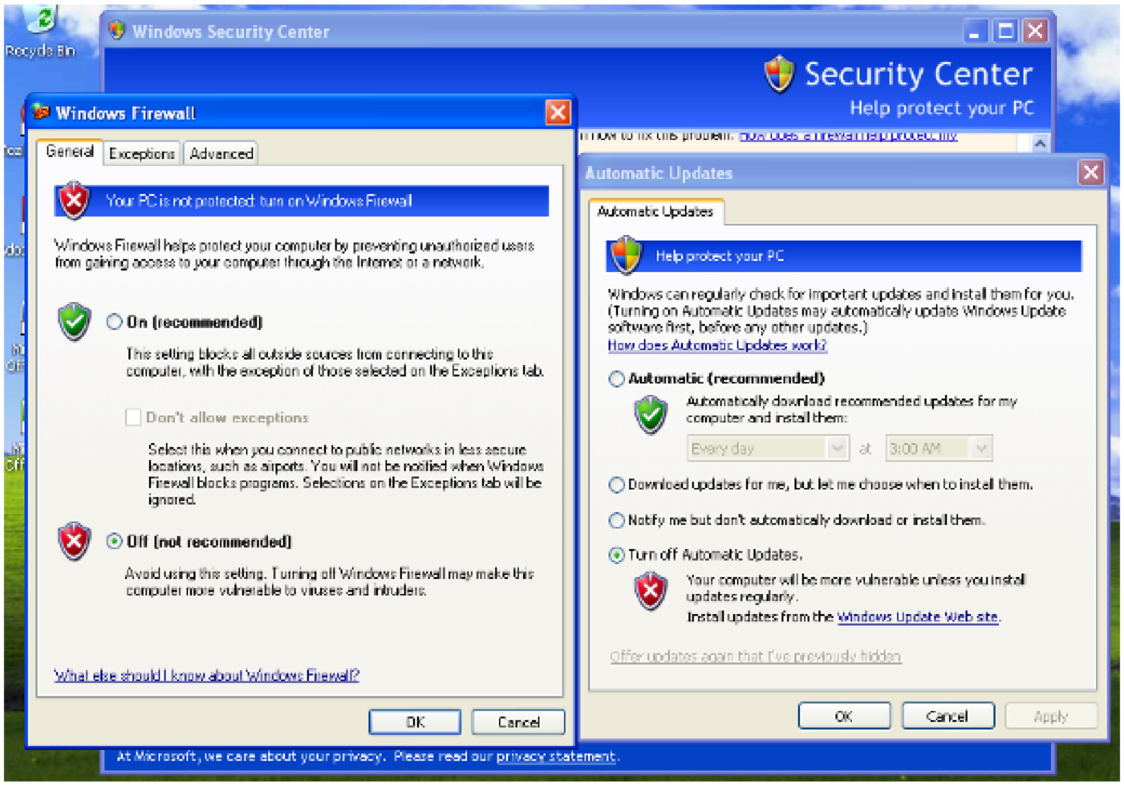
Now it's time to setup the network for your virtual machine.

Windows Settings

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

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

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



Windows Security Center

Security Center
Help protect your PC

Windows Firewall

General | Exceptions | Advanced

Your PC is not protected: turn on Windows Firewall

Windows Firewall helps protect your computer by preventing unauthorized users from gaining access to your computer through the Internet or a network.

( ) On (recommended)

This setting blocks all outside sources from connecting to this computer, with the exception of those selected on the Exceptions tab.

[ ] Don't allow exceptions

Select this when you connect to public networks in less secure locations, such as airports. You will not be notified when Windows Firewall blocks programs. Selections on the Exceptions tab will be ignored.

[✓ Off (not recommended)]

Avoid using this setting. Turning off Windows Firewall may make this computer more vulnerable to viruses and intruders.

What else should I know about Windows Firewall?

OK | Cancel

Automatic Updates

Automatic Updates

Help protect your PC

Windows can regularly check for important updates and install them for you. (Turning on Automatic Updates may automatically update Windows Update software first, before any other updates.)
How does Automatic Updates work?

( ) Automatic (recommended)

Automatically download recommended updates for my computer and install them:

Every day at 3:00 AM

( ) Download updates for me, but let me choose when to install them.

( ) Notify me but don't automatically download or install them.

[✓ Turn off Automatic Updates.]

Your computer will be more vulnerable unless you install updates regularly.
Install updates from the Windows Update Web site.

Offer updates again that I've previously hidden

OK | Cancel | Apply

At Microsoft, we care about your privacy. Please read our privacy statement.

Virtual Networking

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

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

In order to make it work properly you'll have to configure your machine's network so that the Host and the Guest can communicate. Testing the network access by pinging a guest is a good practice, to make sure the virtual network was set up correctly. Use only static IP addresses for your guest, as 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. If you aren't sure about that check your virtualization software documentation and test connectivity with ping and telnet.

The recommended setup is using a Host-Only networking layout with proper forwarding 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 (assuming that eth0 is your outgoing interface, vboxnet0 is your virtual interface and 192.168.56.0/24 is your subnet address):

```
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 forwarding:

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

Installing the Agent

From release 0.4 Cuckoo adopts a custom agent that runs inside the Guest and that handles the communication and the exchange of data with the Host. This agent is designed to be cross-platform, therefore you should be able to use it on Windows 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.

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

Saving the Virtual Machine

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

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

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

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

VirtualBox

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

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

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

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

KVM

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

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

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

Now you have to edit your VM definition as follows:

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

Find the disk section, it looks like this:

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

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

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

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

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

Having multiple snapshots can cause errors.

ERROR: No snapshot found for virtual machine VM-Name

VM snapshots can be managed using the following commands.

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

VMware Workstation

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

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

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

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

XenServer

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cloning the Virtual Machine

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

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

2.2.3 Preparing the Guest (Physical Machine)

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

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

Creation of the Physical Machine

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

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

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

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

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

Requirements

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

Install Python

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

You can download the proper Windows installer from the official website. Also in this case Python 2.7 is preferred.

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

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

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

Additional Software

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

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

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

Additional Host Requirements

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

On Debian/Ubuntu:

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

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

Network Configuration

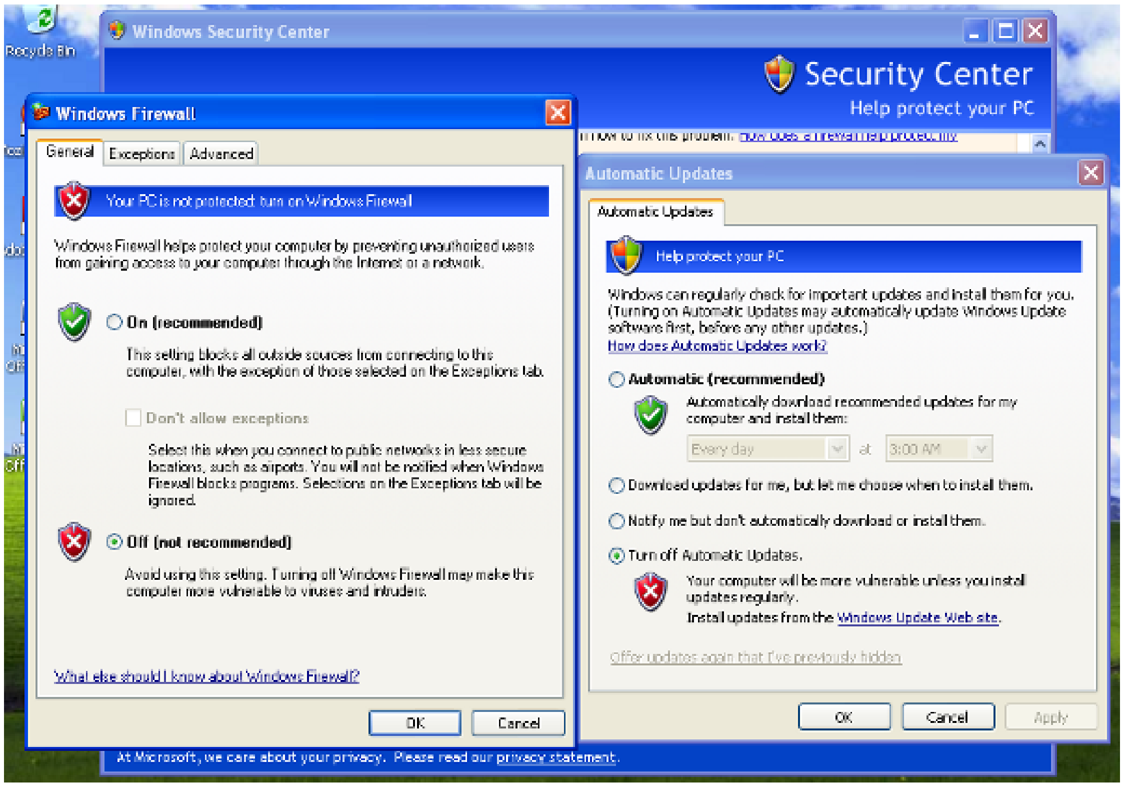
Now it's time to setup the network for your physical machine.

Windows Settings

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

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

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



Your PC is not protected: turn on Windows Firewall

Windows Firewall helps protect your computer by preventing unauthorized users from gaining access to your computer through the Internet or a network.

On (recommended)

This setting blocks all outside sources from connecting to this computer, with the exception of those selected on the Exceptions tab.

Don't allow exceptions

Select this when you connect to public networks in less secure locations, such as airports. You will not be notified when Windows Firewall blocks programs. Selections on the Exceptions tab will be ignored.

[✓ Off (not recommended)]

Avoid using this setting. Turning off Windows Firewall may make this computer more vulnerable to viruses and intruders.

What else should I know about Windows Firewall?

Help protect your PC

Windows can regularly check for important updates and install them for you. (Turning on Automatic Updates may automatically update Windows Update software first, before any other updates.)

How does Automatic Updates work?

Automatic (recommended)

Automatically download recommended updates for my computer and install them:

Every day at 3:00 AM

Download updates for me, but let me choose when to install them.

Notify me but don't automatically download or install them.

[✓ Turn off Automatic Updates.]

Your computer will be more vulnerable unless you install updates regularly.

Install updates from the Windows Update Web site.

Offer updates again that I've previously hidden

At Microsoft, we care about your privacy. Please read our privacy statement.

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

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

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

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

Networking

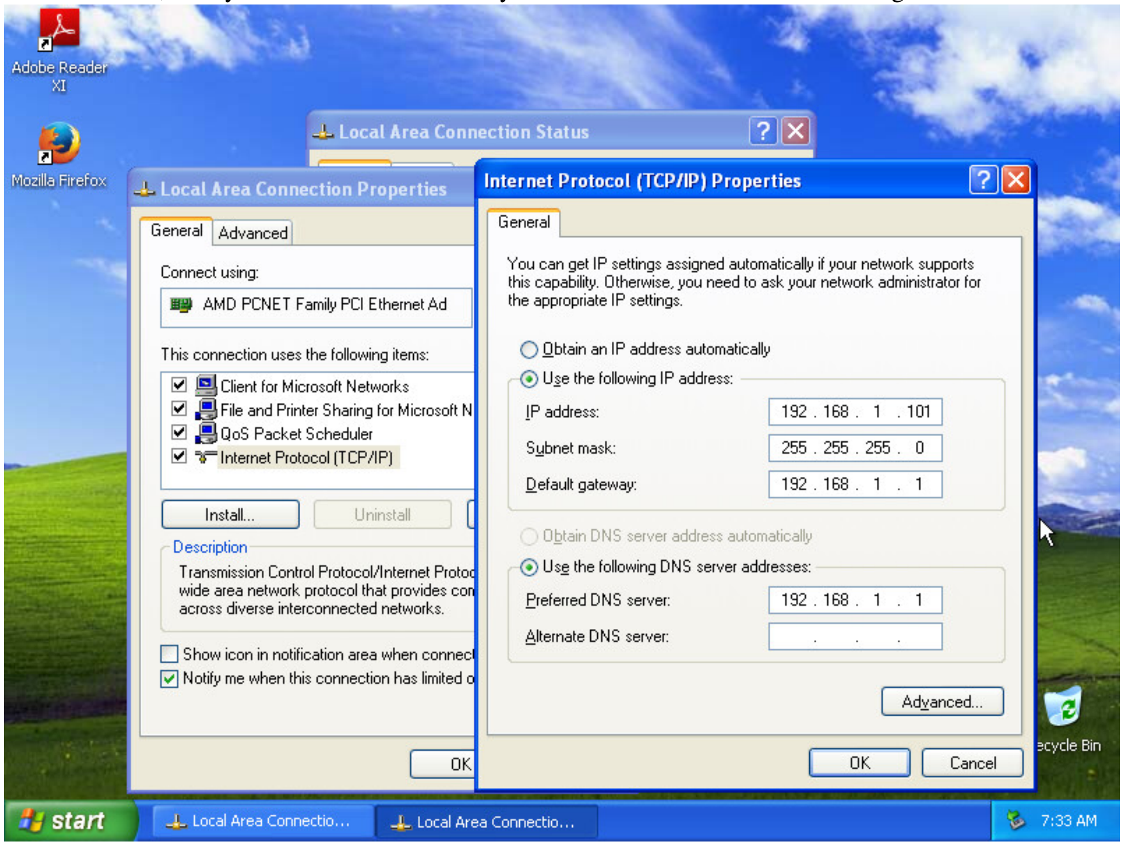
Now you need to decide how to make your physical machine able to access Internet or your local network.

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

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

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

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


Local Area Connection Status
Local Area Connection Properties
General
Advanced
Connect using:
AMD PCNET Family PCI Ethernet Ad
This connection uses the following items:
Client for Microsoft Networks
File and Printer Sharing for Microsoft N
QoS Packet Scheduler
Internet Protocol (TCP/IP)
Install...
Uninstall
Description
Show icon in notification area when connec
Notify me when this connection has limited c
OK
Internet Protocol (TCP/IP) Properties
Advanced...
Cancel
start
7:33 AM

Transmission Control Protocol/Internet Protoc wide area network protocol that provides con across diverse interconnected networks.

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

Obtain an IP address automatically [✓ Use the following IP address:]

IP address: 192 . 168 . 1 . 101
Subnet mask: 255 . 255 . 255 . 0
Default gateway: 192 . 168 . 1 . 1

Obtain DNS server address automatically [✓ Use the following DNS server addresses:]

Preferred DNS server: 192 . 168 . 1 . 1
Alternate DNS server: . . .

Installing the Agent

From release 0.4 Cuckoo adopts a custom agent that runs inside the Guest and that handles the communication and the exchange of data with the Host. This agent is designed to be cross-platform, therefore you should be able to use it on Windows 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.

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

Saving the Guest

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

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

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

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

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

Fog

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

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

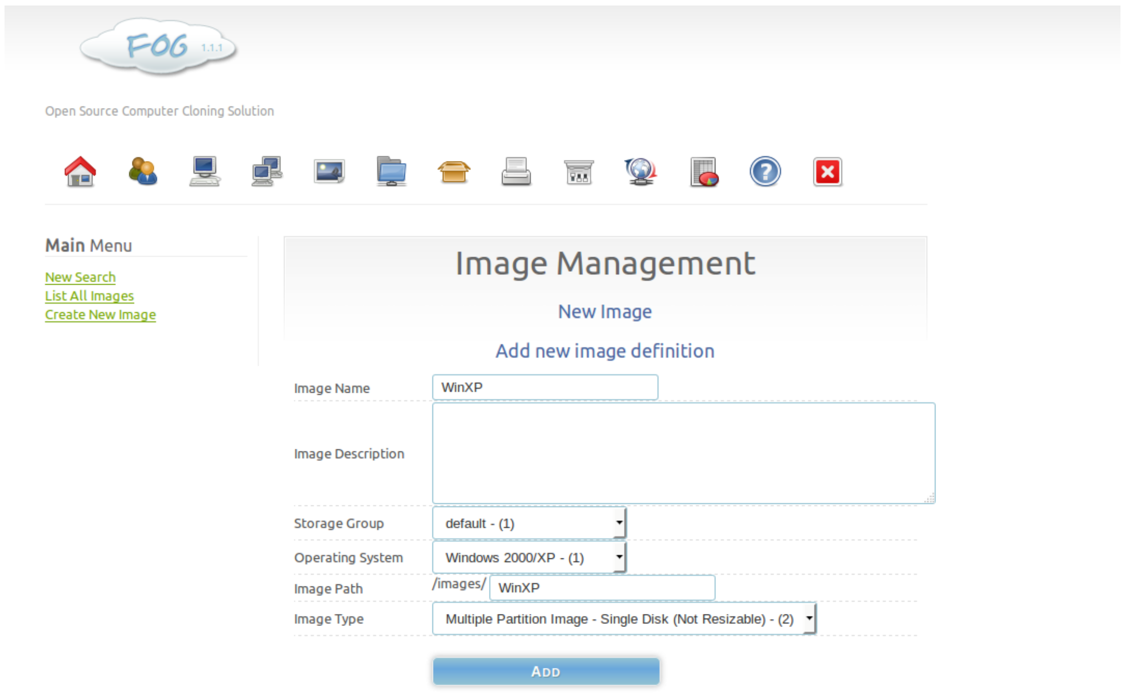
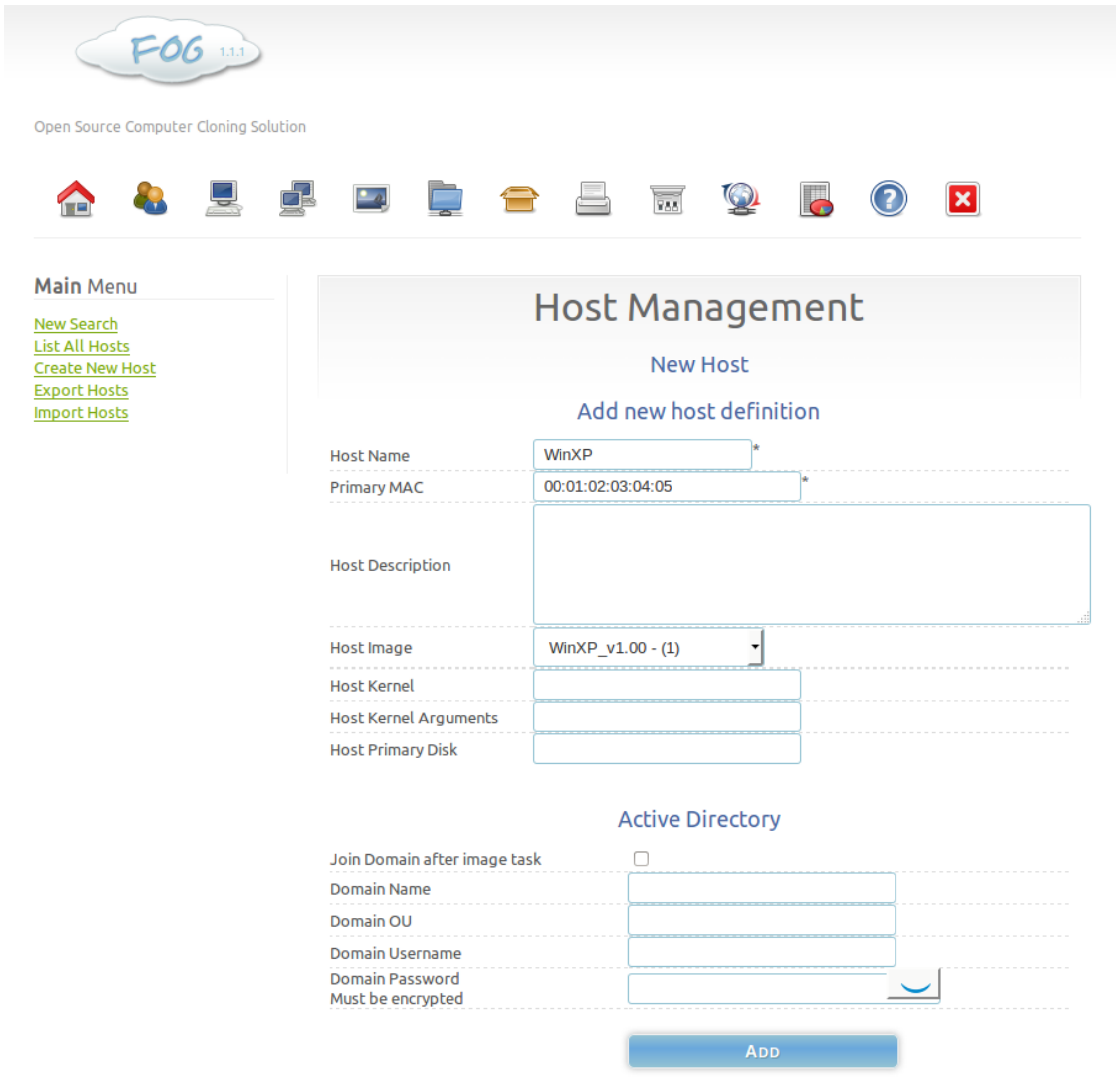


Image Name	WinXP
Image Description	
Storage Group	default - (1)
Operating System	Windows 2000/XP - (1)
Image Path	/images/ WinXP
Image Type	Multiple Partition Image - Single Disk (Not Resizable) - (2)

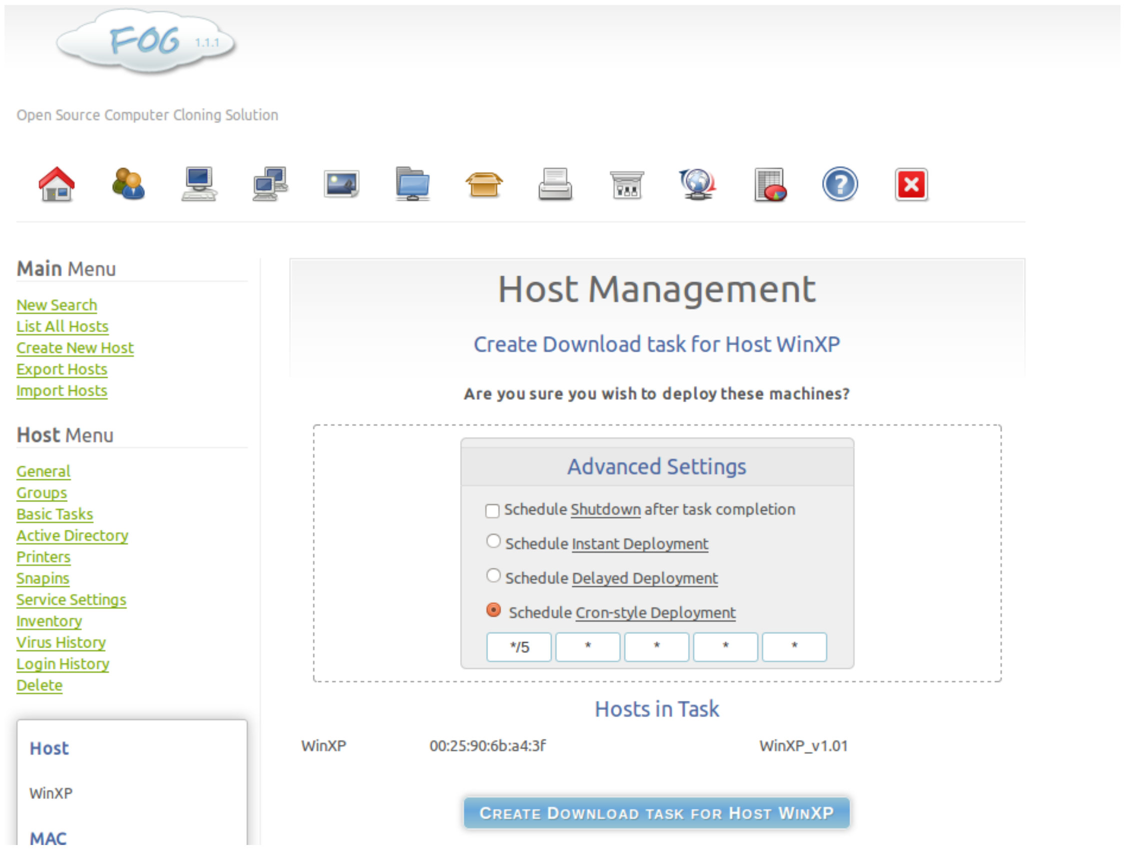
Next you will need to add the host you plan to re-image to Fog. To add a host, open a web browser and navigate to the Host Management page of Fog (http://<your_fog_server>/fog/management/index.php?node=host). Click "Create New Host." Provide the proper inputs for your host configuration. Be sure to select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



Host Name	WinXP *
Primary MAC	00:01:02:03:04:05 *
Host Description	
Host Image	WinXP_v1.00 - (1)
Host Kernel	
Host Kernel Arguments	
Host Primary Disk	
Join Domain after image task	☐
Domain Name	
Domain OU	
Domain Username	
Domain Password Must be encrypted	

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

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



Are you sure you wish to deploy these machines?

Advanced Settings

[ ] Schedule Shutdown after task completion

( ) Schedule Instant Deployment

( ) Schedule Delayed Deployment

[✓ Schedule Cron-style Deployment]

*/5	*	*	*	*

WinXP	00:25:90:6b:a4:3f	WinXP_v1.01

Setup using VMWare (Bonus!)

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

2.2.4 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. There are two ways to upgrade your Cuckoo: start from scratch or migrate your "old" setup (migration is supported only starting from Cuckoo 0.6). The suggested way to upgrade Cuckoo is to start from a fresh setup because it's easier and faster than migrate your old setup.

Upgrade starting from scratch

To start from scratch you have 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. If you are using an external database instead of the default or you are using the MongoDb reporting module is suggested to start all databases from scratch, due to possible schema changes between Cuckoo releases.
7. Test it!

If something goes wrong you probably failed to do 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 Python 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 (agent.py) may change.
- If you are using an external database like the reporting module for MongoDb a change in the data schema may corrupt your database.

Migrate your Cuckoo

Data migration is shipped starting from Cuckoo 1.1 and supports migration starting from Cuckoo 0.6. If your Cuckoo release is older than 0.6 you can't migrate your data.

The following steps are suggested as requirement to migrate your data:

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. Download and extract the latest Cuckoo.
5. Reconfigure Cuckoo as explained in this book (copying old configuration files is not safe because options can change between releases), and update agent in your virtual machines.
6. Copy from your backup "storage" and "db" folders. (Reports and analyses already present in "storage" folder will keep the old format.)

Now setup Alembic (the framework used for migrations) and dateutil with:

```
pip install alembic
pip install python-dateutil
```

Enter the alembic migration directory in "utils/db_migration" with:

```
cd utils/db_migration
```

Before starting the migration script you must set your database connection in "cuckoo.conf" if you are using a custom one. Alembic migration script will use the database connection parameters configured in cuckoo.conf.

Again, please remember to backup before launching migration tool! A wrong configuration may corrupt your data, backup should save kittens!

Run the database migrations with:

```
alembic upgrade head
```

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:

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

 Cuckoo Sandbox 1.2
 www.cuckoosandbox.org
 Copyright (c) 2010-2015

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

2013-04-07 15:57:17,459 [lib.cuckoo.core.scheduler] INFO: Using "virtualbox" machine manager
2013-04-07 15:57:17,861 [lib.cuckoo.core.scheduler] INFO: Loaded 1 machine/s
2013-04-07 15:57:17,862 [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] [-t] [-m MAX_ANALYSIS_COUNT]
                 [-u USER] [--clean]

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
  -t, --test            Test startup
  -m MAX_ANALYSIS_COUNT, --max-analysis-count MAX_ANALYSIS_COUNT
                        Maximum number of analyses
  -u USER, --user USER  Drop user privileges to this user
  --clean               Remove all tasks and samples and their associated data
```

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

2.3.2 Submit an Analysis

- *Submission Utility*

- *API*
- *Distributed Cuckoo*
- *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] [-d] [--remote REMOTE] [--url] [--package PACKAGE]
                 [--custom CUSTOM] [--owner OWNER] [--timeout TIMEOUT]
                 [-o OPTIONS] [--priority PRIORITY] [--machine MACHINE]
                 [--platform PLATFORM] [--memory] [--enforce-timeout]
                 [--clock CLOCK] [--tags TAGS] [--max MAX] [--pattern PATTERN]
                 [--shuffle] [--unique] [--quiet]
                 target

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

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Enable debug logging
  --remote REMOTE       Specify IP:port to a Cuckoo API server to submit
                        remotely
  --url                 Specify whether the target is an URL
  --package PACKAGE     Specify an analysis package
  --custom CUSTOM       Specify any custom value
  --owner OWNER         Specify the task owner
  --timeout TIMEOUT     Specify an analysis timeout
  -o OPTIONS, --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
  --clock CLOCK         Set virtual machine clock
  --tags TAGS           Specify tags identifier of a machine you want to use
  --max MAX             Maximum samples to add in a row
  --pattern PATTERN     Pattern of files to submit
  --shuffle             Shuffle samples before submitting them
  --unique              Only submit new samples, ignore duplicates
  --quiet               Only print text on failure
```

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
```

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

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

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

```
$ ./utils/submit.py --memory --options free=True /path/to/binary
```

API

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

Distributed Cuckoo

Detailed usage of the Distributed Cuckoo API interface is described in *Distributed Cuckoo*.

Python Functions

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

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

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

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

Parameters

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

Return type integer

Example usage:

```
>>> 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*[, *owner=""*[, *machine=None*[, *platform=None*[, *memory=False*[, *enforce_timeout=False*], *clock=None[]*]]]]]]]]])

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

Parameters

- **url** (*string*) – URL to analyze

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

Return type integer

Example Usage:

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

2.3.3 Web interface

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

Configuration

The web interface pulls data from a Mongo database, so having the Mongo reporting module enabled in `reporting.conf` is mandatory for this interface. If that's not the case, the application won't start and it will raise an exception.

The interface can be configured by editing `local_settings.py` under `web/web/`:

```
# If you want to customize your cuckoo path set it here.
# CUCKOO_PATH = "/where/cuckoo/is/placed/"

# Maximum upload size.
MAX_UPLOAD_SIZE = 26214400

# Override default secret key stored in secret_key.py
# Make this unique, and don't share it with anybody.
# SECRET_KEY = "YOUR_RANDOM_KEY"
# Local time zone for this installation. Choices can be found here:
# http://en.wikipedia.org/wiki/List_of_tz_zones_by_name
# although not all choices may be available on all operating systems.
# On Unix systems, a value of None will cause Django to use the same
# timezone as the operating system.
# If running in a Windows environment this must be set to the same as your
# system time zone.
TIME_ZONE = "America/Chicago"

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

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

MANAGERS = ADMINS

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

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

Usage

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

```
$ python manage.py runserver
```

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

```
$ python manage.py runserver 0.0.0.0:PORT
```

You can serve Cuckoo's web interface using WSGI interface with common web servers: Apache, Nginx, Unicorn and so on. Please refer both to the documentation of the web server of your choice as well as Django documentation.

2.3.4 REST API

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

On Debian/Ubuntu with pip:

```
$ pip install flask
```

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 do this:

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

Web deployment

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

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

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

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

To begin, create a uWSGI configuration file at `/etc/uwsgi/apps-available/cuckoo-api.ini`:

```
[uwsgi]
plugins = python
chdir = /home/cuckoo/cuckoo
file = utils/api.py
uid = cuckoo
gid = cuckoo
```

This configuration inherits a number of settings from the distribution's default uWSGI configuration, loading `api.py` from the Cuckoo installation directory. If Cuckoo is installed in a different path, adjust the configuration (the *chdir* setting, and perhaps the *uid* and *gid* settings) accordingly.

Enable the app configuration and start the server:

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

Note: Logs for the application may be found in the standard directory for distribution app instances, i.e.:

`/var/log/uwsgi/app/cuckoo-api.log`

The UNIX socket is created in a conventional location as well:

`/run/uwsgi/app/cuckoo-api/socket`

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

To begin, create a Nginx configuration file at `/etc/nginx/sites-available/cuckoo-api`:

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

# HTTP server
#
server {
    listen 8090;
    listen [::]:8090 ipv6only=on;

    # REST API app
    location / {
        uwsgi_pass  _uwsgi_cuckoo_api;
        include     uwsgi_params;
    }
}
```

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

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

Enable the server configuration and start the server:

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

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

Resources

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

Resource	Description
`POST` */tasks/create/file*	Adds a file to the list of pending tasks to be processed and analyzed.
`POST` */tasks/create/url*	Adds an URL to the list of pending tasks to be processed and analyzed.
`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/delete*	Removes the given task from the database and deletes the results.
`GET` */tasks/report*	Returns the report generated out of the analysis of the task associated with the specified ID. You can optionally specify which report format to return, if none is specified the JSON report will be returned.
`GET` */tasks/screenshots*	Retrieves one or all screenshots associated with a given analysis task ID.
`GET` */memory/list*	Returns a list of memory dump files associated with a given analysis task ID.
`GET` */memory/get*	Retrieves one memory dump file associated with a given analysis task ID.
`GET` */files/view*	Search the analyzed binaries by MD5 hash, SHA256 hash or internal ID (referenced by the tasks details).
`GET` */files/get*	Returns the content of the binary with the specified SHA256 hash.
`GET` */pcap/get*	Returns the content of the PCAP associated with the given task.
`GET` */machines/list*	Returns the list of analysis machines available to Cuckoo.
`GET` */machines/view*	Returns details on the analysis machine associated with the specified name.
`GET` */cuckoo/status*	Returns the basic cuckoo status, including version and tasks overview

/tasks/create/file

POST /tasks/create/file

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

Example request:

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

Example request using Python:

```
import requests
import json

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

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

# Add your code to error checking for request.status_code.
```

```
json_decoder = json.JSONDecoder()
task_id = json_decoder.decode(request.text)["task_id"]

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

Example response:

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

Form parameters:

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

Status codes:

- `200` - no error

/tasks/create/url

POST /tasks/create/url

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

Example request:

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

Example request using Python:

```
import requests
import json

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

multipart_url = {"url": ("", SAMPLE_URL)}
request = requests.post(REST_URL, files=multipart_url)

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

json_decoder = json.JSONDecoder()
task_id = json_decoder.decode(request.text)["task_id"]

# Add your code toerror checking if task_id is None.
```

Example response:

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

Form parameters:

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

Status codes:

- `200` - no error

/tasks/list

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

Returns list of tasks.

Example request:

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

Example response:

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

Parameters:

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

Status codes:

- `200` - no error

/tasks/view

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

Returns details on the task associated with the specified ID.

Example request:

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

Example response:

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

Note: possible value for key `status`:

- `pending`
- `running`

- `completed`
- `reported`

Parameters:

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

Status codes:

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

/tasks/delete

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

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

Example request:

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

Parameters:

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

Status codes:

- `200` - no error
- `404` - task not found
- `500` - unable to delete the task

/tasks/report

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

Returns the report associated with the specified task ID.

Example request:

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

Parameters:

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

Status codes:

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

/tasks/screenshots

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

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

Example request:

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

Parameters:

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

Status codes:

- `404` - file or folder not found

/memory/list

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

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

Example request:

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

Parameters:

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

Status codes:

- `404` - file or folder not found

/memory/get

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

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

Example request:

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

Parameters:

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

Status codes:

- `404` - file or folder not found

/files/view

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

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

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

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

Example request:

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

Example response:

```
{
    "sample": {
        "sha1": "da39a3ee5e6b4b0d3255bfef95601890afd80709",
        "file_type": "empty",
        "file_size": 0,
        "crc32": "00000000",
        "ssdeep": "3::",
        "sha256": "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

/pcap/get

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

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

Example request:

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

Status codes:

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

/machines/list

GET /machines/list

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

Example request:

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

Example response:

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

Status codes:

- `200` - no error

/machines/view

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

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

Example request:

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

Example response:

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

Status codes:

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

/cuckoo/status

GET /cuckoo/status/

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

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

Diskspace directories:

- `analyses` - $CUCKOO/storage/analyses/
- `binaries` - $CUCKOO/storage/binaries/
- `temporary` - `tmppath` as specified in `conf/cuckoo.conf`

Example request:

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

Example response:

```
{
    "tasks": {
        "reported": 165,
        "running": 2,
        "total": 167,
        "completed": 0,
        "pending": 0
    },
    "diskspace": {
        "analyses": {
            "total": 491271233536,
            "free": 71403470848,
            "used": 419867762688
        },
        "binaries": {
            "total": 491271233536,
            "free": 71403470848,
            "used": 419867762688
        },
        "temporary": {
            "total": 491271233536,
            "free": 71403470848,
            "used": 419867762688
        }
    },
    "version": "1.0",
    "protocol_version": 1,
    "hostname": "Patient0",
    "machines": {
        "available": 4,
        "total": 5
    }
}
```

Status codes:

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

2.3.5 Distributed Cuckoo

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

A typical setup thus includes a machine on which the distributed script is run and one or more machines running an instance of the Cuckoo daemon (`./cuckoo.py`) and the *Cuckoo REST API*.

A few notes;

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

Dependencies

The distributed script uses a few Python libraries which can be installed through the following command (on Debian/Ubuntu):

```
$ sudo pip install flask flask-sqlalchemy requests
```

Starting the Distributed REST API

The Distributed REST API requires a few commandline options in order to run. Following is a listing of all available commandline options:

```
$ ./distributed/app.py -h

usage: app.py [-h] [-s SETTINGS] [-v] [host] [port]

positional arguments:
    host                  Host to listen on.
    port                  Port to listen on.

optional arguments:
    -h, --help            show this help message and exit
    -s SETTINGS, --settings SETTINGS
                          Settings file.
    -v, --verbose         Enable verbose logging.
```

The various configuration options are described in the configuration file, but following we have more in-depth descriptions as well.

Report Formats

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

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

Samples Directory

The samples directory denotes the directory where the submitted samples will be stored temporarily, until they're passed on to a Cuckoo node and processed.

Reports Directory

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

RESTful resources

Following are all RESTful resources. Also make sure to check out the *Quick usage* section which documents the most commonly used commands.

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

GET /api/node

Returns all enabled nodes. For each node its associated name, API url, and machines are returned:

```
$ curl http://localhost:9003/api/node
{
    "success": true,
    "nodes": {
        "localhost": {
            "machines": [
                {
                    "name": "cuckoo1",
                    "platform": "windows",
                    "tags": []
                }
            ],
            "name": "localhost",
            "url": "http://localhost:8090/"
        }
    }
}
```

POST /api/node

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

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

GET /api/node/<name>

Get basic information about a particular Cuckoo node:

```
$ curl http://localhost:9003/api/node/localhost
{
    "success": true,
    "nodes": [
        {
            "name": "localhost",
            "url": "http://localhost:8090/"
            "machines": [
                {
                    "name": "cuckoo1",
                    "platform": "windows",
                    "tags": []
                }
            ]
        }
    ]
}
```

PUT /api/node/<name>

Update basic information of a Cuckoo node:

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

DELETE /api/node/<name>

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

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

GET /api/task

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

```
$ curl http://localhost:9003/api/task?limit=1
{
    "success": true,
    "tasks": {
        "1": {
            "clock": null,
            "custom": null,
            "owner": "",
            "enforce_timeout": null,
            "machine": null,
            "memory": null,
            "options": null,
            "package": null,
            "path": "/tmp/dist-samples/tmphal8mS",
            "platform": "windows",
            "priority": 1,
            "tags": null,
            "task_id": 1,
            "timeout": null
        }
    }
}
```

POST /api/task

Submit a new file or URL to be analyzed:

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

GET /api/task/<id>

Get basic information about a particular task:

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

DELETE /api/task/<id>

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

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

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

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

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

Quick usage

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

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

```
$ curl http://localhost:9003/api/node \
    -F name=localhost -F url=http://localhost:8090/
```

Disable a Cuckoo node:

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

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

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

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

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

Proposed setup

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

Configuration settings

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

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

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

Update `connection` to use something *not* sqlite3. Preferably PostgreSQL or MySQL. SQLite3 doesn't support multi-threaded applications that well and this will give errors at random if used.

You should create your own empty database for the distributed cuckoo setup. Do not be tempted to use any existing cuckoo database in order to avoid update problems with the DB scripts. In the config use the new database name, the remaining stuff like usernames , servers can be the same as for your cuckoo install.

conf/processing.conf You may want to disable some processing modules, such as `virustotal`.

conf/reporting.conf Depending on which report(s) are required for integration with your system it might make sense to only make those report(s) that you're going to use. Thus disable the other ones.

conf/virtualbox.conf Assuming `VirtualBox` is the Virtual Machine manager of choice, the `mode` will have to be changed to `headless` or you will have some restless nights.

Setup Cuckoo

On each machine the following three scripts should be ran:

```
./cuckoo.py
./utils/api.py -H 1.2.3.4  # IP accessible by the Distributed script.
./utils/process.py auto
```

One way to do this is by placing each script in its own `screen(1)` session as follows, this allows one to check back on each script to ensure it's (still) running successfully:

```
$ screen -S cuckoo  ./cuckoo.py
$ screen -S api     ./utils/api.py
$ screen -S process ./utils/process.py auto
```

Setup Distributed Cuckoo

On the first machine (so the say the "management machine") start a few separate `screen(1)` sessions for the Distributed Cuckoo scripts with all the required parameters (see the rest of the documentation on the parameters for this script):

```
$ screen -S distributed ./distributed/app.py
$ SCREEN -S dist_scheduler ./distributed/instance.py dist.scheduler
$ SCREEN -S dist_status ./distributed/instance.py dist.status
$ SCREEN -S cuckoo1 ./distributed/instance.py -v cuckoo1
```

The -v parameter enables verbose output and the cuckoo1 parameter is the name assigned to the actual cuckoo instance running the virtual machine while registering the node as outlined below.

Register Cuckoo nodes

As outlined in *Quick usage* the Cuckoo nodes have to be registered with the Distributed Cuckoo script:

```
$ curl http://localhost:9003/node -F name=cuckoo0 -F url=http://localhost:8090/
$ curl http://localhost:9003/node -F name=cuckoo1 -F url=http://1.2.3.4:8090/
```

Having registered the Cuckoo nodes all that's left to do now is to submit tasks and fetch reports once finished. Documentation on these commands can be found in the *Quick usage* section. In case you are not using localhost, replace localhost with the IP of the node where there distributed.py is running and the -F url parameter points to the nodes running the actual virtual machines.

2.3.6 Analysis Packages

The **analysis packages** are a core component of Cuckoo Sandbox. They consist in structured Python classes which, when executed in the guest machines, describe how Cuckoo's analyzer component should conduct the analysis.

Cuckoo provides some default analysis packages that you can use, but you are able to create your own or modify the existing ones. You can find them at *analyzer/windows/modules/packages/*.

As described in *Submit an Analysis*, you can specify some options to the analysis packages in the form of `key1=value1,key2=value2`. The existing analysis packages already include some default options that can be enabled.

Following is 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.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `bin`: used to analyze generic binary data, such as **shellcodes**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `cpl`: used to analyze **Control Panel Applets**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `dll`: used to run and analyze **Dynamically 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`.
  - `arguments`: specify arguments to pass to the DLL through commandline.

  - `loader`: specify a process name to use to fake the DLL launcher name instead of rundll32.exe (this is used to fool possible anti-sandboxing tricks of certain malware)
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `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.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `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.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `generic`: used to run and analyze **generic samples** via cmd.exe.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `html`: used to analyze **Internet Explorer**‘s behavior when opening the given HTML file.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `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.

  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `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.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `msi`: used to run and analyze **MSI windows installer**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `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.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `ppt`: used to run and analyze **Microsoft PowerPoint documents**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `ps1`: used to run and analyze **PowerShell scripts**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `python`: used to run and analyze **Python scripts**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `vbs`: used to run and analysis **VBScript files**.

  Options:

  - `free` *[yes/no]*: if enabled, no behavioral logs will be produced and the malware will be executed freely.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `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.
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

- `zip`: used to run and analyze **Zip archives**.

  Options:

  - `file`: specify the name of the file contained in the archive to execute. If none is specified, Cuckoo will try to execute *sample.exe*.
  - `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.
  - `password`: specify the password of the archive. If none is specified, Cuckoo will try to extract the archive without password or use the password "*infected*".
  - `procmemdump` *[yes/no]*: if enabled, take memory dumps of all actively monitored processes.
  - `dll`: specify the name of an optional DLL to be used as a replacement for cuckoomon.dll.

You can find more details on how to start creating new analysis packages in the *Analysis Packages* customization chapter.

As you already know, you can select which analysis package to use by specifying its name at submission time (see *Submit an Analysis*) as follows:

```
$ ./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 we encourage to specify the package name whenever possible.

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

```
$ ./utils/submit.py --package dll --options function=FunctionName,loader=explorer.exe /path/to/malwar
```

2.3.7 Analysis Results

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

Following is an example of an analysis directory structure:

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

analysis.conf

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

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.

2.3.8 Clean all Tasks and Samples

Since Cuckoo 1.2 a built-in **–clean** feature has been added, it drops all associated information of the tasks and samples in the database. If you submit a task after running **–clean** then you'll start with `Task #1` again.

To clean your setup, run:

```
$ ./cuckoo.py --clean
```

To sum up, this command does the following:

- Delete analysis results.
- Delete submitted binaries.
- Delete all associated information of the tasks and samples in the configured database.
- Delete all data in the configured MongoDB (if configured and enabled in reporting.conf).

Warning: If you use this command you will delete permanently all data stored by Cuckoo in all storages: file system, SQL database and MongoDB database. Use it only if you are sure you would clean up all the data.

2.3.9 Utilities

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

Submission Utility

Submits samples 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 1
```

If you want to re-generate the reports:

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

Following are the usage options:

```
$ ./utils/process.py -h
usage: process.py [-h] [-d] [-r] [-p PARALLEL] [-u USER] [-m MODULES] id

positional arguments:
  id                    ID of the analysis to process (auto for continuous
                        processing of unprocessed tasks).

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Display debug messages
  -r, --report          Re-generate report
  -p PARALLEL, --parallel PARALLEL
                        Number of parallel threads to use (auto mode only).
  -u USER, --user USER  Drop user privileges to this user
  -m MODULES, --modules MODULES
                        Path to signature and reporting modules - overrides
                        default modules path.
```

As best practice we suggest to adopt the following configuration if you are running Cuckoo with many virtual machines:

- Run a stand alone process.py in auto mode (you choose the number of parallel threads)
- Disable Cuckoo reporting in cuckoo.conf (set process_results to off)

This could increase the performance of your system because the reporting is not yet demanded to Cuckoo.

With Cuckoo 2 a new processing utility was introduced, it is more stable and with better performance. It is dubbed *process2.py*, following are the usage options:

```
$ ./utils/process2.py -h
usage: process2.py [-h] [-d] [-u USER] [-m MODULES] instance

positional arguments:
  instance              Task processing instance.

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Display debug messages
  -u USER, --user USER  Drop user privileges to this user
  -m MODULES, --modules MODULES
```

```
                        Path to signature and reporting modules - overrides
                        default modules path.
```

Community Download Utility

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

```
$ ./utils/community.py -h

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

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, --machinery       Download machine managers
  -n, --analyzer        Download analyzer modules
  -g, --agent           Download agent modules
  -r, --reporting       Download reporting modules
  -f, --force           Install files without confirmation
  -w, --rewrite         Rewrite existing files
  -b BRANCH, --branch BRANCH
                        Specify a different branch
```

Example: install all available signatures:

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

Database migration utility

This utility is developed to migrate your data between Cuckoo's release. It's developed on top of the Alembic framework and it should provide data migration for both SQL database and Mongo database. This tool is already described in *Upgrade from a previous release*.

Stats utility

This is a really simple utility which prints some statistics about processed samples:

```
$ ./utils/stats.py

1 samples in db
1 tasks in db
pending 0 tasks
running 0 tasks
completed 0 tasks
recovered 0 tasks
reported 1 tasks
failed_analysis 0 tasks
failed_processing 0 tasks
roughly 32 tasks an hour
roughly 778 tasks a day
```

Machine utility

The machine.py utility is designed to help you automatize the configuration of virtual machines in Cuckoo. It takes a list of machine details as arguments and write them in the specified configuration file of the machinery module enabled in *cuckoo.conf*. Following are the available options:

```
$ ./utils/machine.py -h
usage: machine.py [-h] [--debug] [--add] [--delete] [--ip IP]
                  [--platform PLATFORM] [--tags TAGS] [--interface INTERFACE]
                  [--snapshot SNAPSHOT] [--resultserver RESULTSERVER]
                  vmname

positional arguments:
  vmname                Name of the Virtual Machine.

optional arguments:
  -h, --help            show this help message and exit
  --debug               Debug log in case of errors.
  --add                 Add a Virtual Machine.
  --delete              Delete a Virtual Machine.
  --ip IP               Static IP Address.
  --platform PLATFORM   Guest Operating System.
  --tags TAGS           Tags for this Virtual Machine.
  --interface INTERFACE
                        Sniffer interface for this machine.
  --snapshot SNAPSHOT   Specific Virtual Machine Snapshot to use.
  --resultserver RESULTSERVER
                        IP:Port of the Result Server.
```

Distributed scripts

There are a couple of shell scripts used to automate distributed utility:

- “start-distributed” is used to start distributed Cuckoo
- “stop-distributed” is used to stop distributed Cuckoo

Mac OS X Bootstrap scripts

A couple of bootstrap scripts used for Mac OS X analysis are located in *utils/darwin* folder, they are used to bootstrap the guest and host system for Mac OS X malware analysis. Some settings are defined as constants inside them, so it is suggested to have a look at them and configure them for your needs.

SMTP Sinkhole

The smtp_sinkhole.py utility is designed to provide an easy to use SMTP sinkhole to catch all the emails going out of virtual machines network. This is typically used to dump all emails when you run an analysis of sample used for spam purposes. You can use it also to prevent sending spam on internet. Following are the available options:

```
$ ./utils/smtp_sinkhole.py -h
usage: smtp_sinkhole.py [host [port]]

SMTP Sinkhole

positional arguments:
  host
  port

optional arguments:
  -h, --help  show this help message and exit
  --dir DIR   Directory used to dump emails.
```

By default, if you run it without arguments, it will listen for incoming mails on localhost port 1025. Yoy can bind it on different address and port, as in the following example:

```
$ ./utils/smtp_sinkhole.py 192.168.56.1 1025
```

If you want to save the dumped emails to disk, just use the *–dir* argument and specify an existent directory where save them, as in the following example:

```
$ ./utils/smtp_sinkhole.py --dir /home/dumpmail
```

You have to use iptables to route all mails generated from your analysis virtual machine network to the sinkhole script, for example if 192.168.56.0/24 is the address of your virtual network and smtp_sinkhole.py is listening on 192.168.56.1 port 1025 you can use the following command:

```
$ sudo iptables -t nat -A PREROUTING -i vboxnet0 -p tcp -m tcp --dport 25 -j REDIRECT --to-ports 1025
```

Setup script

Cuckoo setup script is a tool to setup a whole Cuckoo environment on a Debian based OS (i.e. Ubuntu or Debian). Actually it is a working in progress, but it is suggested to give it a try! It is located in *utils/setup.sh* and it is configured by some constants, so you should edit it if you want to customize the behaviour.

2.4 Customization

This chapter explains how to customize Cuckoo. Cuckoo is written in a modular architecture built to be as customizable as it can, to fit the needs of all users.

2.4.1 Auxiliary Modules

Auxiliary modules define some procedures that need to be executed in parallel to every single analysis process. All auxiliary modules should be placed under the *modules/auxiliary/* directory.

The skeleton of a module would look something like this:

```python
from lib.cuckoo.common.abstracts import Auxiliary

class MyAuxiliary(Auxiliary):

    def start(self):
        # Do something.

    def stop(self):
        # Stop the execution.
```

The function `start()` will be executed before starting the analysis machine and effectively executing the submitted malicious file, while the `stop()` function will be launched at the very end of the analysis process, before launching the processing and reporting procedures.

For example, an auxiliary module provided by default in Cuckoo is called *sniffer.py* and takes care of executing **tcpdump** in order to dump the generated network traffic.

2.4.2 Machinery Modules

Machinery modules define how Cuckoo should interact with your virtualization software (or potentially even with physical disk imaging solutions). Since we decided to not enforce any particular vendor, from release 0.4 you are able to use your preferred solution and, in case it's not supported by default, write a custom Python module that defines how to make Cuckoo use it.

Every machinery module should be located inside *modules/machinery/*.

A basic machinery module would look like this:

```python
from lib.cuckoo.common.abstracts import Machinery
from lib.cuckoo.common.exceptions import CuckooMachineError

class MyMachinery(Machinery):
    def start(self, label):
        try:
            revert(label)
            start(label)
        except SomethingBadHappens as e:
            raise CuckooMachineError("OPS!")

    def stop(self, label):
        try:
            stop(label)
        except SomethingBadHappens as e:
            raise CuckooMachineError("OPS!")
```

The only requirements for Cuckoo are that:

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

As you understand, the machinery module is a core part of a Cuckoo setup, therefore make sure to spend enough time debugging your code and make it solid and resistant to any unexpected error.

Configuration

Every machinery module should come with a dedicated configuration file located in *conf/<machinery module name>.conf*. For example for *modules/machinery/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 its `ip`.

These fields are required by Cuckoo in order to use the already embedded `initialize()` function that generates the list of available machines.

If you plan to change the configuration structure you should override the `initialize()` function (inside your own module, no need to modify Cuckoo's core code). You can find its original code in the `Machinery` abstract inside *lib/cuckoo/common/abstracts.py*.

LibVirt

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

```python
from lib.cuckoo.common.abstracts import LibVirtMachinery

class MyMachinery(LibVirtMachinery):
    # 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.

2.4.3 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

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

    def start(self, path):
        args = self.options.get("arguments")
        return self.execute(path, args)
```

It seems really easy, thanks to all method inherited by Package object. Let's have a look as some of the main methods an analysis package inherits from Package object:

```
1  from lib.api.process import Process
2  from lib.common.exceptions import CuckooPackageError
3
4  class Package(object):
5      def start(self):
6          raise NotImplementedError
7
8      def check(self):
9          return True
10
11     def execute(self, path, args):
12         dll = self.options.get("dll")
13         free = self.options.get("free")
14         suspended = True
15         if free:
16             suspended = False
17
18         p = Process()
19         if not p.execute(path=path, args=args, suspended=suspended):
20             raise CuckooPackageError("Unable to execute the initial process, "
21                                      "analysis aborted.")
22
23         if not free and suspended:
24             p.inject(dll)
25             p.resume()
26             p.close()
27             return p.pid
28
29     def finish(self):
30         if self.options.get("procmemdump"):
31             for pid in self.pids:
32                 p = Process(pid=pid)
33                 p.dump_memory()
34         return True
```

Let's walk through the code:

- Line **1**: import the `Process` API class, which is used to create and manipulate Windows processes.

- Line **2**: import the `CuckooPackageError` exception, which is used to notify issues with the execution of the package to the analyzer.
- Line **4**: define the main class, inheriting `object`.
- Line **5**: define the `start()` function, which takes as argument the path to the file to execute. It should be implemented by each analysis package.
- Line **8**: define the `check()` function.
- Line **13**: acquire the `free` option, which is used to define whether the process should be monitored or not.
- Line **18**: initialize a `Process` instance.
- Line **19**: try to execute the malware, if it fails it aborts the execution and notify the analyzer.
- Line **23**: check if the process should be monitored.
- Line **24**: inject the process with our DLL.
- Line **25**: resume the process from the suspended state.
- Line **27**: return the PID of the newly created process to the analyzer.
- Line **29**: define the `finish()` function.
- Line **30**: check if the `procmemdump` option was enabled.
- Line **31**: loop through the currently monitored processes.
- Line **32**: open a `Process` instance.
- Line **33**: take a dump of the process memory.

start()

In this function you have to place all the initialization operations you want to run. This may include running the malware process, launching additional applications, taking memory snapshots and more.

check()

This function is executed by Cuckoo every second while the malware is running. You can use this function to perform any kind of recurrent operation.

For example if in your analysis you are looking for just one specific indicator to be created (e.g. a file) you could place your condition in this function and if it returns `False`, the analysis will terminate 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.

execute()

Wraps the malware execution and deal with DLL injection.

finish()

This function is simply called by Cuckoo before terminating the analysis and powering off the machine. By default, this function contains an optional feature to dump the process memory of all the monitored processes.

Options

Every package have automatically access to a dictionary containing all user-specified options (see *Submit an Analysis*).

Such options are made available in the attribute `self.options`. For example let's assume that the user specified the following string at submission:

```
foo=1,bar=2
```

The analysis package selected will have access to these values:

```
from lib.common.abstracts import Package

class Example(Package):

    def start(self, path):
        foo = self.options["foo"]
        bar = self.options["bar"]

    def check():
        return True

    def finish():
        return True
```

These options can be used for anything you might need to configure inside your package.

Process API

The `Process` class provides access to different process-related features and functions. You can import it in your analysis packages with:

```
from lib.api.process import Process
```

You then initialize an instance with:

```
p = Process()
```

In case you want to open an existing process instead of creating a new one, you can specify multiple arguments:

- `pid`: PID of the process you want to operate on.
- `h_process`: handle of a process you want to operate on.
- `thread_id`: thread ID of a process you want to operate on.
- `h_thread`: handle of the thread of a process you want to operate on.

This class implements several methods that you can use in your own scripts.

Methods

`Process.open()`

Opens an handle to a running process. Returns `True` or `False` in case of success or failure of the operation.

Return type boolean

Example Usage:

```
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:

```python
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 operation.

Return type boolean

Example Usage:

```python
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 instead of `CreateRemoteThread()`, beware that if the process is in suspended mode, Cuckoo will always use `QueueUserAPC()`

Return type boolean

Example Usage:

```python
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:

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

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

```
[foobar]
enabled = on
```

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 transformations 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.
- **Memory** *(modules/processing/memory.py)* - executes Volatility on a full memory dump.
- **NetworkAnalysis** *(modules/processing/network.py)* - parses the PCAP file and extracts some network information, such as DNS traffic, domains, IPs, HTTP requests, IRC and SMTP traffic.
- **ProcMemory** *(modules/processing/procmemory.py)* - performs analysis of process memory dump. **Note**: the module is able to process user defined Yara rules from data/yara/memory/index_memory.yar. Just edit this file to add your Yara rules.
- **StaticAnalysis** *(modules/processing/static.py)* - performs some static analysis of PE32 files.
- **Strings** *(modules/processing/strings.py)* - extracts strings from the analyzed binary.
- **TargetInfo** *(modules/processing/targetinfo.py)* - includes information on the analyzed file, such as hashes.
- **VirusTotal** *(modules/processing/virustotal.py)* - searches on VirusTotal.com for antivirus signatures of the analyzed file. **Note**: the file is not uploaded on VirusTotal.com, if the file was not previously uploaded on the website no results will be retrieved.

Getting started

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

A basic processing module could look like:

```python
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 sub container for the returned data.
- A set of data (list, dictionary, string, etc.) that will be appended to the global container.

You can also specify an `order` value, which allows you to run the available processing modules in an ordered sequence. By default all modules are set with an `order` value of `1` and are executed in alphabetical order.

If you want to change this value your module would look like:

```python
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`:

```python
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.
- `self.pmemory_path`: path to the process memory dumps, if created.

With these attributes you should be able to easily access all the raw results stored by Cuckoo and perform your analytic operations on them.

As a last note, a good practice is to use the `CuckooProcessingError` exception whenever the module encounters an issue you want to report to Cuckoo. This can be done by importing the class like this:

```
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
```

2.4.5 Signatures

With Cuckoo you're able to create some customized signatures that you can run against the analysis results in order to identify some predefined pattern that might represent a particular malicious behavior or an indicator you're interested in.

These signatures are very useful to give a context to the analyses: both because they simplify the interpretation of the results as well as for automatically identifying malware samples of interest.

Some examples of what you can use Cuckoo's signatures for:

- Identify a particular malware family you're interested in by isolating some unique behaviors (like file names or mutexes).
- Spot interesting modifications the malware performs on the system, such as installation of device drivers.
- Identify particular malware categories, such as Banking Trojans or Ransomware by isolating typical actions commonly performed by those.
- Classify samples into the categories malware/unknown (it is not possible to identify clean samples)

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

Getting started

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

First things first, all signatures must be located inside *modules/signatures/*.

The following is a basic example signature:

```python
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 = "1.2"
10
11     def on_complete(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 1.2 Cuckoo provides some helper functions that make the process of creating signatures much easier.

In this example we just walk through all the accessed files in the summary and check if there is anything ending with "*.exe*": in that case it will return `True`, meaning that the signature matched, otherwise return `False`.

the function `on_complete` is called at the end of the cuckoo signature process. Other function will be called before on specific events and help you to write more sophisticated and faster signatures.

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

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

Creating your new signature

In order to make you better understand the process of creating a signature, we are going to create a very simple one together and walk through the steps and the available options. For this purpose, we're simply going to create a signature that checks whether the malware analyzed opened a mutex named "i_am_a_malware".

The first thing to do is import the dependencies, create a skeleton and define some initial attributes. These are the ones you can currently set:

- `name`: an identifier for the signature.
- `description`: a brief description of what the signature represents.
- `severity`: a number identifying the severity of the events matched (generally between 1 and 3).

- `categories`: a list of categories that describe the type of event being matched (for example “*banker*”, “*injection*” or “*anti-vm*”).
- `families`: a list of malware family names, in case the signature specifically matches a known one.
- `authors`: a list of people who authored the signature.
- `references`: a list of references (URLs) to give context to the signature.
- `enable`: if set to False the signature will be skipped.
- `alert`: if set to True can be used to specify that the signature should be reported (perhaps by a dedicated reporting module).
- `minimum`: the minimum required version of Cuckoo to successfully run this signature.
- `maximum`: the maximum required version of Cuckoo to successfully run this signature.

In our example, we would create the following skeleton:

```python
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 = "1.2" # We specify that in order to run the signature, the user will need at least

def on_complete(self):
    return
```

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

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

```python
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 = "1.2"

def on_complete(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 way:

```python
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 = "1.2"

def on_complete(self):
    for process in self.get_processes_by_pid():
        if "summary" in process and "mutexes" in process["summary"]:
            for mutex in process["summary"]["mutexes"]:
                if mutex == "i_am_a_malware":
                    return True

    return False
```

Evented Signatures

Since version 1.0, Cuckoo provides a way to write more high performance signatures. In the past every signature was required to loop through the whole collection of API calls collected during the analysis. This was necessarily causing some performance issues when such collection would be of a large size.

Since 1.2 Cuckoo only supports the so called "evented signatures". The old signatures based on the `run` function can be ported to using `on_complete`. The main difference is that with this new format, all the signatures will be executed in parallel and a callback function called `on_call()` will be invoked for each signature within one single loop through the collection of API calls.

An example signature using this technique is the following:

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

class SystemMetrics(Signature):
    name = "generic_metrics"
    description = "Uses GetSystemMetrics"
    severity = 2
    categories = ["generic"]
    authors = ["Cuckoo Developers"]
    minimum = "1.2"

    # Evented signatures can specify filters that reduce the amount of
    # API calls that are streamed in. One can filter Process name, API
    # name/identifier and category. These should be sets for faster lookup.
    filter_processnames = set()
    filter_apinames = set(["GetSystemMetrics"])
    filter_categories = set()

    # This is a signature template. It should be used as a skeleton for
    # creating custom signatures, therefore is disabled by default.
    # The on_call function is used in "evented" signatures.
    # These use a more efficient way of processing logged API calls.
    enabled = False

    def on_complete(self):
        # In the on_complete method one can implement any cleanup code and
        #  decide one last time if this signature matches or not.
        #  Return True in case it matches.
        return False

    # This method will be called for every logged API call by the loop
    # in the RunSignatures plugin. The return value determines the "state"
    # of this signature. True means the signature matched and False it did not this time.
    # Use self.deactivate() to stop streaming in API calls.
    def on_call(self, call, pid, tid):
        # This check would in reality not be needed as we already make use
        # of filter_apinames above.
        if call["api"] == "GetSystemMetrics":
            # Signature matched, return True.
            return True

        # continue
        return None
```

The inline comments are already self-explanatory.

Another event is triggered when a signature matches.

```
def on_signature(self, matched_sig):
    required = ["creates_exe", "badmalware"]
    for sig in required:
        if not sig in self.list_signatures():
            return
    return True
```

This kind of signature can be used to combine several signatures identifying anomalies into one signature classifying the sample (malware alert).

Two more events can be used to write more complex signatures. They are called when new processes are processed or new threads. `on_process` and `on_thread`. They can be used to reset the state of flags when a new process is entered. Allowing to write signatures that take into account if a series of events happened in one thread/process or globally.

```
def on_process(self, pid):
    pass

def on_thread(self, pid, tid):
    pass
```

Quickout

Additionally to the filters you can use a quickout function to determine if the signature can be matched at all. For example if a signature is written to identify behavior of malicious PDFs you could test for the file type to be PDF. Returning `True` will remove the signature from the list of potential candidates (reducing False Positives and processing time).

You can find many more example of signatures in our community repository.

Matches

Starting from version 1.2, signatures are able to log exactly what triggered the signature. This allows users to better understand why this signature is present in the log, and to be able to better focus malware analysis.

Two helpers have been included in order to specify matching data.

`Signature.`**`add_match`**`(`*process*, *type*, *match*`)`
Adds a match to the signature. Can be called several times for the same signature.

Parameters

- **process** (*dict*) – process dictionary (same as the `on_call` argument). Should be `None` except when used in evented signatures.
- **type** (*string*) – nature of the matching data. Can be anything (ex: `'file'`, `'registry'`, etc.). If match is composed of api calls (when used in evented signatures), should be `'api'`.
- **match** – matching data. Can be a single element or a list of elements. An element can be a string, a dict or an API call (when used in evented signatures).

Example Usage, with a single element:

```python
self.add_match(None, "url", "http://malicious_url_detected.com")
```

Example Usage, with a more complex signature, needing several API calls to be triggered:

```python
self.signs = []
self.signs.append(first_api_call)
self.signs.append(second_api_call)
self.add_match(process, 'api', self.signs)
```

`Signature.`**`has_matches`**`()`

Checks whether the current signature has any matching data registered. Returns `True` in case it does, otherwise returns `False`.

This can be used to easily add several matches for the same signature. If you want to do so, make sure that all the api calls are scanned by making sure that `on_call` never returns `True`. Then, use `on_complete` with `has_matches` so that the signature is triggered if any match was previously added.

Return type boolean

Example Usage, from the *network_tor* signature:

```python
def on_call(self, call, process):
    if self.check_argument_call(call,
                                pattern="Tor Win32 Service",
                                api="CreateServiceA",
                                category="services"):
        self.add_match(process, "api", call)

def on_complete(self):
    return self.has_matches()
```

Helpers

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

With Cuckoo 1.2 the amount of information extracted from a sample grew another step and the resulting data format got more complex. To avoid having to port your signatures with every extension and to reduce errors we strongly suggest to use the helpers to access data.

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*[, *actions=["regkey_written", "regkey_opened", "regkey_read"]*[, *pid=None*]]])

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
- **actions** (*list*) – a list of key-access-actions to search the key in
- **pid** (*int*) – process id to filter for

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_call`**`(`*call, pattern[, name=Name[, api=None[, category=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

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

Return type boolean

Example Usage:

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

`Signatures.`**`list_signatures`**`()`

Returns a list of signature names that matched so far. It can be used to write meta-signatures combining several signatures on anomalies into a classification.

Return type list

Example Usage:

```
def on_signature(self, matched_sig):
    required = ["creates_exe", "badmalware"]
    for sig in required:
        if not sig in self.list_signatures():
            return
    return True
```

`Signatures.`**`get_processes`**`(`[*name=None*]`)`

An iterator returning the processes monitored. If name is given, they will be filtered for the name

Parameters **name** (*string*) – Name of the process to filter for

Return type iterator

Example Usage:

```
for process in self.get_processes("foo"):
    pass
```

`Signatures.`**`get_processes_by_pid`**`(`[*pid=None*]`)`

An iterator returning the processes monitored. If pid is given, they will be filtered for the process id

Parameters **pid** (*int*) – Process ID of the process to filter for

Return type iterator

```
for process in self.get_processes_by_pid(4):
    pass
```

`Signatures.`**`get_threads`**`(`[*pid=None*]`)`

An iterator returning the threads monitored. If pid is given, they will be filtered for the process id.

Parameters **pid** (*int*) – Name of the process to filter for

Return type iterator

```
for thread in self.get_threads():
    pass
```

Signatures.**get_files**([*pid=None*[, *actions=None*]])
Iterates over the files accessed by a process (or all processes). Access type can be a list of "file_written", "file_read", "file_deleted". Default is all.

Parameters

- **pid** (*int*) – Name of the process to filter for
- **actions** (*list*) – access types of the files to return

Return type iterator

```
for afile in self.get_files():
    pass
```

Signatures.**get_keys**([*pid=None*[, *actions=None*]])
Iterates over the registry keys accessed by a process (or all processes). Access type can be a list of "regkey_written", "regkey_opened", "regkey_read". Default is all.

Parameters

- **pid** (*int*) – Name of the process to filter for
- **actions** (*list*) – access types of the registry keys to return

Return type iterator

```
for akey in self.get_keys():
    pass
```

Signatures.**get_mutexes**([*pid=None*])
Returns a list of mutexes. Optionally filtered by process id

Parameters **pid** (*int*) – Name of the process to filter for

Return type list

```
for mutex in self.get_mutexes():
    pass
```

Signature.**get_net_hosts**()
Returns a list of hosts from the network sniffing part of the collected data

Return type list

```
for host in self.get_net_hosts():
    pass
```

Signature.**get_net_domains**()
Returns a list of domains from the network sniffing part of the collected data

Return type list

```
for domain in self.get_net_domains():
    pass
```

Signature.**get_net_http**()
Returns a list of http information blocks from the network sniffing part of the collected data

Return type list

```
for http_data in self.get_net_http():
    pass
```

`Signature.`**`get_net_udp`**`()`

Returns a list of udp information blocks from the network sniffing part of the collected data

Return type list

```
for udp_data in self.get_net_udp():
    pass
```

`Signature.`**`get_net_icmp`**`()`

Returns a list of icmp information blocks from the network sniffing part of the collected data

Return type list

```
for icmp_data in self.get_net_icmp():
    pass
```

`Signature.`**`get_net_irc`**`()`

Returns a list of irc information blocks from the network sniffing part of the collected data

Return type list

```
for irc_data in self.get_net_irc():
    pass
```

`Signature.`**`get_net_smtp`**`()`

Returns a list of smtp information blocks from the network sniffing part of the collected data

Return type list

```
for smtp_data in self.get_net_smtp():
    pass
```

`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)
```

`Signature.flags.`**`set`**`(`*name*[, *pid=None*[, *tid=None*[, *timestamp=None*]]]`)`

Flags can be used to collect information in the on_call section of a signature and react (decide to alert) in the on_complete part if all required flags are set. Flags are signature specific. They are identified by their name. PID, TID and timestamp can be later used to identify if a flag was set in a specific process/thread or at a specific time.

Parameters

- **name** (*string*) – Name of the flag to set
- **pid** (*int*) – process id
- **tid** (*int*) – thread id
- **timestamp** (*int*) – timestamp in the log to mark this flag for

```
def on_call(self, call, pid, tid):
    self.flags.set("foo", 1, 2, 2345)
```

`Signature.flags.`**`find`**`(`*[name=None[, pid=None[, tid=None[, before=None[, after=None]]]]]*`)`

Returns a list of flags matching the given criteria

Parameters

- **name** (*string*) – Name of the flag look for
- **pid** (*int*) – process id to filter for
- **tid** (*int*) – thread id to filter for
- **before** (*int*) – flag timestamp must be <= before-timestamp
- **after** (*int*) – flag timestamp must be >= after-timestamp

```
def on_complete(self):
    if self.flags.find("foo"):
        self.data.append({"Flag found matching name": "foo"})
        return True
```

`Signature.`**`mark_start`**`()`

Mark the start of a api-call region relevant for the signature. This way the report can contain a link to the API call that triggered the signature. As soon as the signature returns `True` this mark will be stored in the report. Subsequent start marks will overwrite the old one till it is stored in the results with the triggering of the signature. So you can set a start mark "on suspicion" and overwrite it several times till the signature triggers.

It is marking the api call. So the only reasonable signatures to use it is in on_call evented ones

```python
def on_call(self, call, pid, tid):
    if self.check_argument_call(call, pattern=".*cuckoo.*", category="filesystem", regex=True):
        self.mark_start()
        return True
```

`Signature.`**`mark_end`**`()`

A complementary function to mark_start. It is optional and marks the end of a api call range that triggered the signature. It should be called before returning the `True` result.

Without a prior mark_start, the end-mark will not be stored in the result.

```python
def on_call(self, call, pid, tid):
    if self.check_argument_call(call, pattern=".*foo.*", category="filesystem", regex=True):
        self.mark_start()
        return None
    if self.check_argument_call(call, pattern=".*cuckoo.*", category="filesystem", regex=True):
        self.mark_end()
        return True
```

`Signature.`**`deactivate`**`()`

Deactivate a signature. Deactivated signatures will not be notified in `on_call` events. This can be used after a signature triggered (just before the return) to match this signature only once.

```python
def on_call(self, call, pid, tid):
    if call["api"] == "LdrGetProcedureAddress":
        self.deactivate()
        return True
```

`Signature.`**`activate`**`()`

Re-activates a signature after it has been de-activated.

```python
def on_process(self, pid):
    self.activate()
```

2.4.6 Reporting Modules

After the raw analysis results have been processed and abstracted by the processing modules and the global container is generated (ref. *Processing Modules*), it is passed over by Cuckoo to all the reporting modules available, which will make use of it and will make it accessible and consumable in different formats.

Getting Started

All reporting modules must be placed inside the directory *modules/reporting/*.

Every module must 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:

```python
import os
import json
import codecs
```

```python

from lib.cuckoo.common.abstracts import Report
from lib.cuckoo.common.exceptions import CuckooReportError

class JsonDump(Report):
    """Saves analysis results in JSON format."""

    def run(self, results):
        """Writes report.
        @param results: Cuckoo results dict.
        @raise CuckooReportError: if fails to write report.
        """
        try:
            report = codecs.open(os.path.join(self.reports_path, "report.json"), "w", "utf-8")
            json.dump(results, report, sort_keys=False, indent=4)
            report.close()
        except (UnicodeError, TypeError, IOError) as e:
            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 from `Report`.
- Have a `run()` function performing the main operations.
- Try to catch most exceptions and raise `CuckooReportError` to notify the issue.

All reporting modules have access to some attributes:

- `self.analysis_path`: path to the folder containing the raw analysis results (e.g. *storage/analyses/1/*)
- `self.reports_path`: path to the folder where the reports should be written (e.g. *storage/analyses/1/reports/*)
- `self.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.

Up until version 1.0 we used to coordinate all ongoing development in a dedicated "development" branch and we've been exclusively merging pull requests in such branch. Since version 1.1 we moved development to the traditional "master" branch and we make use of GitHub's tags and release system to reference development milestones in time.

Release Versioning

Cuckoo releases are named using three numbers separated by dots, such as 1.2.3, where the first number is the release, the second number is the major version, the third number is the bugfix version. The testing stage from git ends with "-beta" and development stage with "-dev".

Warning: If you are using a "beta" or "dev" stage, please consider that it's not meant to be an official release, therefore we don't guarantee its functioning and we don't generally provide support. If you think you encountered a bug there, make sure that the nature of the problem is not related to your own misconfiguration and collect all the details to be notified to our developers. Make sure to specify which exact version you are using, eventually with your current git commit id.

Ticketing system

To submit bug reports or feature requests, please use GitHub's Issue tracking system.

Contribute

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

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) 2010-2013 Claudio Guarnieri.
```

# Copyright (C) 2014-2015 Cuckoo Foundation. # This file is part of Cuckoo Sandbox - http://www.cuckoosandbox.org # See the file 'docs/LICENSE' for copying permission.

Indentation

The code must have a 4-spaces-tabs indentation. Since Python enforce the indentation, make sure to configure your editor properly or your code might cause malfunctioning.

Maximum Line Length

Limit all lines to a maximum of 79 characters.

Blank Lines

Separate the class definition and the top level function with one blank line. Methods definitions inside a class are separated by a single blank line:

```
class MyClass:
    """Doing something."""

    def __init__(self):
        """Initialize"""
        pass

    def do_it(self, what):
        """Do it.
        @param what: do what.
        """
        pass
```

Use blank lines in functions, sparingly, to isolate logic sections. Import blocks are separated by a single blank line, import blocks are separated from classes by one blank line.

Imports

Imports must be on separate lines. If you're importing multiple objects from a package, use a single line:

```
from lib import a, b, c
```

NOT:

```
from lib import a
from lib import b
from lib import c
```

Always specify explicitly the objects to import:

```
from lib import a, b, c
```

NOT:

```
from lib import *
```

Strings

Strings must be delimited by double quotes (").

Printing and Logging

We discourage the use of `print()`: if you need to log an event please use Python's `logging` which is already initialized by Cuckoo.

In your module add:

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

And use the `log` handle, for more details refer to the Python 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.

The following is the current Cuckoo exceptions chain:

```
.-- CuckooCriticalError
|   |-- CuckooStartupError
|   |-- CuckooDatabaseError
|   |-- CuckooMachineError
|   `-- CuckooDependencyError
|-- CuckooOperationalError
|   |-- CuckooAnalysisError
|   |-- CuckooProcessingError
|   `-- CuckooReportError
`-- CuckooGuestError
```

Beware that the use of `CuckooCriticalError` and its child exceptions will cause Cuckoo to terminate.

Naming

Custom exception names must start with "Cuckoo" and end with "Error" if it represents an unexpected malfunction.

Exception handling

When catching an exception and accessing its handle, use `as e`:

```
try:
    foo()
except Exception as e:
    bar()
```

NOT:

```
try:
    foo()
except Exception, something:
    bar()
```

It's a good practice use "e" instead of "e.message".

Documentation

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

Automated testing

We believe in automated testing to provide high quality code and avoid dumb bugs. When possible, all code must be committed with proper unit tests. Particular attention must be placed when fixing bugs: it's good practice to write unit tests to reproduce the bug. All unit tests and fixtures are placed in the tests folder in the Cuckoo root. We adopted Nose as unit testing framework.

2.6 Final Remarks

2.6.1 Links

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

2.6.2 Join the discussion

You can get in contact with the Cuckoo 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, the Cuckoo blog, the documentation and Google about your issue. **DO NOT** post questions that have already been answered over and over everywhere.
- Posting messages saying just something like "Doesn't work, help me" are completely useless. If something is not working report the error, paste the logs, the config file, the information on the virtual machine, the results of the troubleshooting, etc. Give context. We are not wizards and we don't have a crystal ball.
- Use a proper title. Stuff like "Doesn't work", "Help me", "Error" are not proper titles.
- Try to use pastebin.com, pastie.org or similar services to paste logs and configs: makes the message more readable.
- **The community website uses Markdown syntax**. So please read the 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 any sort. If you're interested in cooperating, 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

Role	Name Contact	
Claudio nex Guarnieri	Lead Developer	`nex at nex dot sx`
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

It's hard at this point to keep track of all individual contributions. Following is the list of people who contributed code to our GitHub repository:

```
$ git shortlog -s -n
  1058  Nex
   960  jekil
   286  Jurriaan Bremer
   242  rep
   185  nex
    72  Ivan Kirillov
    70  Thorsten Sick
    35  Alessandro Tanasi
    24  Mark Schloesser
    24  Pietro Delsante
    22  David Maciejak
    15  Adam Meily
    14  Justin Roberts
    13  Greg Back
    11  r3comp1le
     9  Christopher Schmitt
     9  Script Kiddie
     7  Hugh Pearse
     7  SpoonBoy
     6  Tal Jerome
     6  init99
     5  David Francos
     5  jamu
     5  lehmz
     4  Adam Pridgen
     4  Ben Small
     3  Allen Swackhamer
     3  Espen Fjellvær Olsen
     3  Jerome Marty
     3  KillerInstinct
     3  Nagy Ferenc László
     3  Stephen DiCato
     3  mak
     3  robertsjw
     3  wzr
     3  z0mbiehunt3r
     2  Claudio Guarnieri
     2  Gael Muller
     2  Mario Vilas
     2  Max Taube
     2  Neriberto C.Prado
     2  Richard Harman
     2  Roberto Abdelkader Martínez Pérez
     2  SecTecRes
     2  Thomas Penteker
     2  Will Metcalf
     2  bcyrill
     2  kholbrook1303
     2  mcpacosy
     2  mt00at
     2  upsidedwn
     1  =
     1  Aitor Gómez
     1  Alexander J
     1  Andrea De Pasquale
     1  Ben Lyon
     1  Benjamin Vanheuverzwijn
     1  Crashman1983
     1  Henrique Menezes
     1  John Davison
     1  Mark Woan
     1  Micha Lenk
     1  Nitzan Carmel
     1  Ryan Peck
     1  SnakeByte Lab
     1  Valter Santos
     1  bladeswords
     1  chimerhapsody
     1  chort
     1  chrestme
     1  jvoisin
     1  sabri
     1  shendo
     1  vacmf
```

There is a number of friends who provided feedback, ideas and support during the years of development of this project, including:

- Felix Leder
- Tillmann Werner
- Georg Wicherski
- David Watson

- Christian Seifert

2.6.5 Supporters

- The Honeynet Project
- The Shadowserver Foundation