Cuckoo Sandbox Book

Release 0.4.2

Cuckoo Sandbox

September 02, 2015

Contents

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

This book will explain you how to setup Cuckoo, use it and customize it.

CHAPTER 1

Having troubles?

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

1.1 FAQ

Frequently Asked Questions:

1.1.1 General Questions

Can I use Volatility with Cuckoo?

Cuckoo does not provide support for Volatility by default. If you want to perform additional forensics on the analysis machine, at the moment you'll have to implement such support by yourself. In the future we might support a full memory dump of the virtual machines, but it's not in our short term plans at the moment.

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

Despite being highly customizable, please also consider that Cuckoo has been designed for full automation. If you're planning to perform manual analysis of your malwares, probably Cuckoo is not the best choice.

1.1.2 Troubleshooting

After upgrade Cuckoo stops to work

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

Please follow the upgrade steps described in Upgrade from a previous release.

Cuckoo stumbles in some error I don't understand

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

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

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

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

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

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

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

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

Check and restore current snapshot with KVM

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

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

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

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

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

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

Choose one snapshot name and set it as current:

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

Now the virtual machine state is fixed.

Check and restore current snapshot with VirtualBox

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

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

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

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

With the following check the current snapshots state:

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

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

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

Cuckoo eats all my RAM

It's a rare case, but some sample can generate gigs of logs. These logs must be processed by Cuckoo and all reports generated, and eat some RAM. To avoid memory leaks, file bigger than analysis_size_limit in cuckoo.conf aren't processed. If you need to process them, you have to do it manually via submit.py utility.

I got a lot of errors from virtualization

Proper sizing of VM number running on a host is fundamental, if you oversize the VM number you get too much I/O. All virtualization software we have tested, under high I/O, raise error when you try to run CLI utilities. If you get some errors, try to decrease the number of running VMs.

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

CHAPTER 2

Contents

2.1 Introduction

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

2.1.1 Sandboxing

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

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

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

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

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

Using a Sandbox

Before starting installing, configuring and using Cuckoo you should take some time to think on what you want to achieve with it and how.

Some questions you should ask yourself:

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

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

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

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

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

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

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

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

2.1.2 What is Cuckoo?

Cuckoo is an open source automated malware analysis system.

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

It can retrieve the following type of results:

- Traces of win32 API calls performed by all processes spawned by the malware.
- Files being created, deleted and downloaded by the malware during its execution.
- Memory dumps of the malware processes.
- Network traffic trace in PCAP format.
- Screenshots of Windows desktop taken during the execution of the malware.

Some History

Cuckoo Sandbox started as a Google Summer of Code project in 2010 within The Honeynet Project. It was originally designed and developed by *Claudio "nex" Guarnieri*, who is still the main developer and coordinates all efforts from joined developers and contributors.

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

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

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

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

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

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

Use Cases

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

It can be used to analyze:

- Generic Windows executables
- DLL files
- PDF documents
- Microsoft Office documents
- URLs
- PHP scripts
- *Almost anything else*

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

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

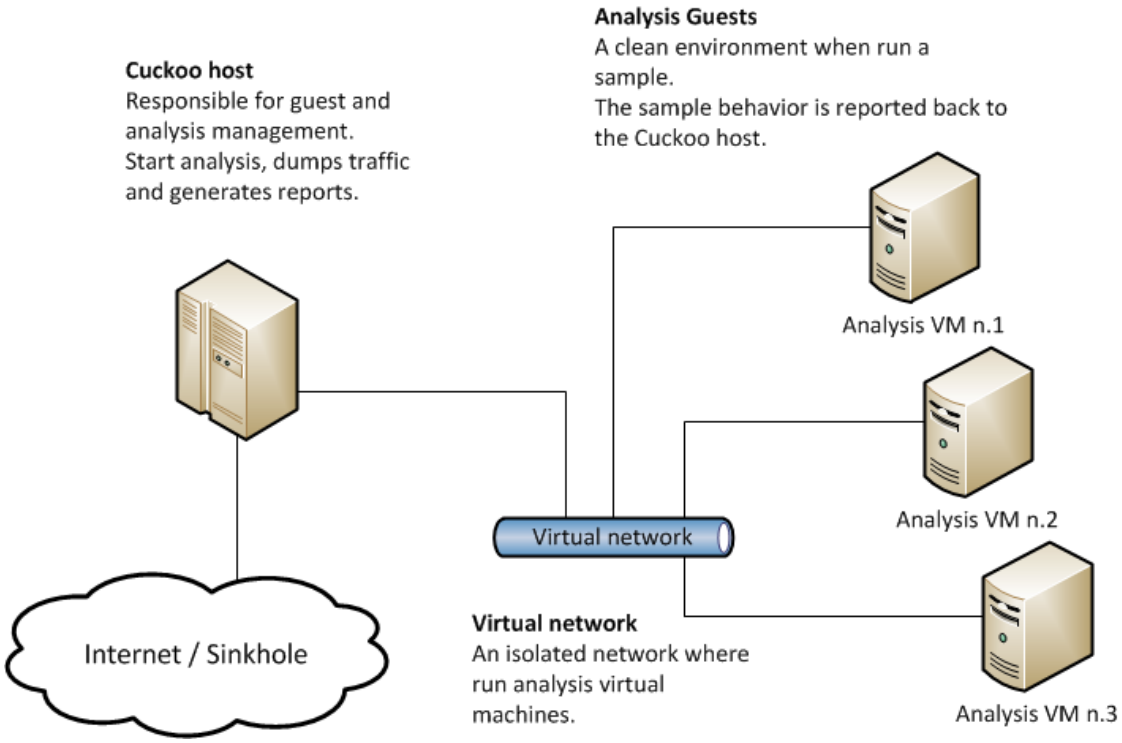
Architecture

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

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

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

The following picture explains Cuckoo's main architecture:



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

Obtaining Cuckoo

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

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

2.1.3 License

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

2.1.4 Disclaimer

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

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

2.2 Installation

This chapter explains how to install Cuckoo.

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

2.2.1 Preparing the Host

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

Requirements

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

Installing Python libraries

Cuckoo host components are completely written in Python, therefore make sure to have an appropriate version installed. For current release Python 2.6 or 2.7 are preferred.

Install Python on Ubuntu:

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

In order to properly execute, Cuckoo really just needs the default installation of Python.

However several additional features and modules require some Python libraries you will need to install in order to make them run successfully. We suggest you to install all of them so that you can take advantage of the project at its full potential.

- Magic (Highly Recommended): for identifying files' formats (otherwise use "file" command line utility)
- Dpkt (Highly Recommended): for extracting relevant information from PCAP files.
- Mako (Highly Recommended): for rendering the HTML reports and the web interface.
- Pydeep (Optional): for calculating ssdeep fuzzy hash of files.
- Pymongo (Optional): for storing the results in a MongoDB database.
- Yara and Yara Python (Optional): for matching Yara signatures.
- Libvirt (Optional): for using the KVM module.

Some of them are packaged in GNU/Linux Ubuntu and you can install them with the following command:

```
$ sudo apt-get install python-magic python-dpkt python-mako python-pymongo
```

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

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

For the rest refer to their websites.

Virtualization Software

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

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

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

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

Installing Tcpdump

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

By default Cuckoo adopts tcpdump, the prominent open source solution.

Install it on Ubuntu:

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

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

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

You can verify the results of last command with:

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

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

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

Installing Cuckoo

Proceed with download and installation.

Create a user

You either can run Cuckoo from your own user or create a new one dedicated just to your sandbox setup. Make sure that the user that runs Cuckoo is the same user that you will use to create and run the virtual machines, otherwise Cuckoo won't be able to identify and launch them.

Create a new user:

```
$ sudo adduser cuckoo
```

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

```
$ sudo useradd -G vboxusers cuckoo
```

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

```
$ sudo useradd -G libvirtd cuckoo
```

Install Cuckoo

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

Configuration

Cuckoo relies on three main configuration files:

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

cuckoo.conf

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

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

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

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

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

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

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

```
use_sniffer = 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
```

The configuration file is self-explainatory.

<machinemanager>.conf

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

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

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

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

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

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

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

# Specify a comma-separated list of available machines to be used. For each
# specified ID you have to define a dedicated section containing the details
# on the respective machine. (E.g. cuckoo1,cuckoo2,cuckoo3)
machines = cuckoo1

[cuckoo1]
# Specify the label name of the current machine as specified in your
# VirtualBox configuration.
label = cuckoo1

# Specify the operating system platform used by current machine
# [windows/darwin/linux].
platform = windows

# Specify the IP address of the current machine. Make sure that the IP address
# is valid and that the host machine is able to reach it. If not, the analysis
# will fail.
ip = 192.168.56.101
```

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

The comments for the options are self-explainatory.

reporting.conf

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

It contains the following sections:

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

[jsondump]
enabled = on

[reporthtml]
enabled = on

[pickled]
enabled = on

[metadata]
enabled = on

[maec11]
enabled = on

[mongodb]
enabled = off

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

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

2.2.2 Preparing the Guest

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

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

Creation of the Virtual Machine

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

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

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

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

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

Requirements

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

Install Python

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

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

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

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

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

Additional Softwares

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

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

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

Network Configuration

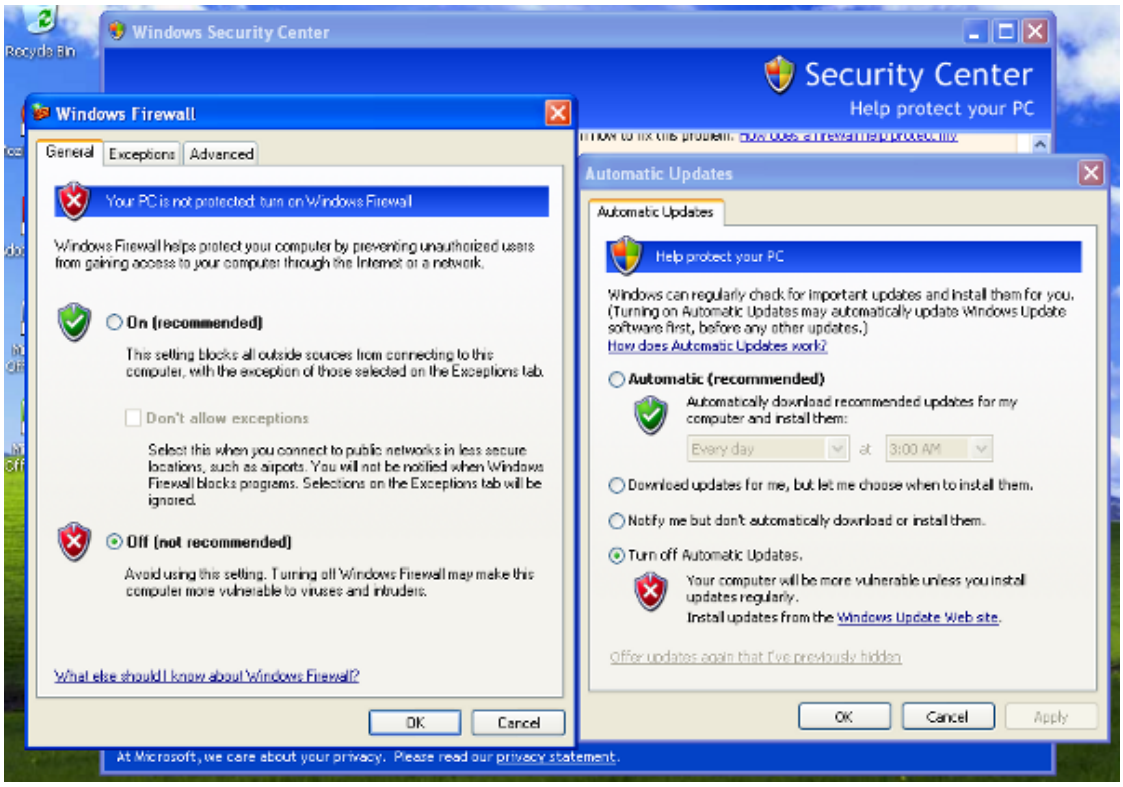
Now it's the time to setup the network configuration for your virtual machine.

Windows Settings

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

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

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



Virtual Networking

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

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

In order to make it work properly you'll have to configure your machine's network so that the Host and the Guest can communicate. Test network trying to ping a guest is a good practice, to be sure about virtual network setup.

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

Install the Agent

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

It's very simple.

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

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

Saving the Virtual Machine

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

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

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

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

VirtualBox

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

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

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

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

KVM

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

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

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

Now you have to edit your VM definition like following:

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

Find the disk section, it looks like this:

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

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

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

Now test your virtual machine, if all works prepare it for snapshotting while running Cuckoo's agent. You can finally take a snapshot with the following command:

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

VMware Workstation

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

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

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

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

Cloning the Virtual Machine

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

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

2.2.3 Upgrade from a previous release

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

The following steps are suggested:

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

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

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

- You are overwriting Python source files (.py) but Pyhton bytecode files (.pyc) are still in place.
- There are configuration files changes across the two versions.
- The part of Cuckoo which runs inside guests may change.

2.3 Usage

This chapter explains how to use Cuckoo.

2.3.1 Starting Cuckoo

To start Cuckoo use the command:

```
$ python cuckoo.py
```

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

You will get an output similar to this:

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

 Cuckoo Sandbox 0.4
 www.cuckoobox.org
 Copyright (c) 2010-2012

2012-06-11 20:17:29,694 [lib.cuckoo.core.scheduler] INFO: Loaded 1 machine/s
2012-06-11 20:17:29,694 [lib.cuckoo.core.scheduler] INFO: Waiting for analysis tasks...
```

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] [-l]

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
  -l, --logo     Show artwork
```

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

2.3.2 Submit an analysis

In order to submit a file to be analyzed you can:

- Use provided **submit.py** utility.
- Use provided **web.py** utility.

- Directly interact with the **SQLite database**.
- Use Cuckoo **Python functions** directly from Cuckoo's library.

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] [--package PACKAGE] [--timeout TIMEOUT]
                 [--options OPTIONS] [--priority PRIORITY] [--machine MACHINE]
                 [--platform PLATFORM]
                 path

positional arguments:
  path                  Path to the file or folder to analyze

optional arguments:
  -h, --help            show this help message and exit
  --package PACKAGE     Specify an analysis package
  --timeout TIMEOUT     Specify an analysis timeout
  --options OPTIONS     Specify options for the analysis package (e.g.
                        "name=value,name2=value2")
  --priority PRIORITY   Specify a priority for the analysis represented by an
                        integer
  --machine MACHINE     Specify the identifier of a machine you want to use
  --platform PLATFORM   Specify the operating system platform you want to use
                        (windows/darwin/linux)
```

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 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
```

Web Utility

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

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

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

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

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

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

Interact with SQLite

Cuckoo is designed to be easily integrated in larger solutions and to be fully automated. In order to automate analysis submission or to provide a different interface rather than the command-line (for instance a web interface), you can directly interact with the SQLite database located at *db/cuckoo.db*.

The database contains the table *tasks* which is defined as the following schema:

```
1  CREATE TABLE tasks (
2      id INTEGER PRIMARY KEY,
3      md5 TEXT DEFAULT NULL,
4      file_path TEXT NOT NULL,
5      timeout INTEGER DEFAULT NULL,
6      priority INTEGER DEFAULT 0,
7      custom TEXT DEFAULT NULL,
8      machine TEXT DEFAULT NULL,
9      package TEXT DEFAULT NULL,
10     options TEXT DEFAULT NULL,
11     platform TEXT DEFAULT NULL,
12     added_on DATE DEFAULT CURRENT_TIMESTAMP,
13     completed_on DATE DEFAULT NULL,
14     lock INTEGER DEFAULT 0,
15     status INTEGER DEFAULT 0
16 );
```

Following are the details on the fields:

- `id`: it's the numeric ID also used to name the results folder of the analysis.
- `md5`: it's the MD5 hash of the target file.
- `file_path`: it's the path pointing to the file to analyze.
- `timeout`: it's the analysis timeout, if none has been specified the field is set to NULL.
- `priority`: it's the analysis priority, if none has been specified the field is set to NULL.

- `custom`: it's a custom user-defined text that can be used for synchronization between submission and post-analysis processing.
- `machine`: it's the ID of a virtual machine the user specifically wants to use for the analysis.
- `package`: it's the name of the analysis package to be used, if non has been specified the field is set to NULL.
- `options`: it's a comma-separated list of options to pass to the analysis package.
- `platform`: it's the operating system platform to use for this analysis.
- `added_on`: it's the timestamp of when the analysis request was added.
- `completed_on`: it's the timestamp of when the analysis has been completed.
- `lock`: it's field internally used by Cuckoo to lock pending analysis.
- `status`: it's a numeric field representing the status of the analysis (0 = not completed, 1 = failed, 2 = succeeded).

Cuckoo Python Functions

In case you want to write your own Python submission script, you can use the `add()` function provided by Cuckoo, which has the following prototype:

```
    def add(self,
            file_path,
            md5=None,
            timeout=None,
            package=None,
            options=None,
            priority=None,
            custom=None,
            machine=None,
            platform=None):
```

Following is a usage example:

```
1  #!/usr/bin/env python
2  from lib.cuckoo.core.database import Database
3
4  db = Database()
5  db.add("/path/to/binary")
```

2.3.3 Analysis Packages

The **analysis packages** are a core component of Cuckoo Sandbox.

They consist in structured Python classes which, executed in the guest machines, describe how Cuckoo's analyzer component should conduct the analysis.

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

Following is the list of existing packages:

- **`exe`: default analysis package used to analyze generic Windows executables.** You can specify an "arguments" option if you want to pass arguments to the process creation (see Submit an analysis).

- **`dll`: used to run and analyze Dinamically Linked Libraries.** You can specify a "function" option that will instruct Cuckoo to execute the specified exported function. If this option is not set, Cuckoo will try to execute the regular `DllMain` function. You can also specify a "free" option that will instruct Cuckoo not to inject and hook the `rundll32` process and let the library run (not behavior results will be produced).
- **`pdf`: used to run and analyze PDF documents.** The path to Acrobat Reader is hardcoded in the package, so make sure to verify that it's matches the correct one in your guest environment.
- **`doc`: used to run and analyze Microsoft Word documents.** Same as the `pdf` package, verify Office Word path.
- **`xls`: used to run and analyze Microsoft Excel documents.** Verify Office Excel path.
- **`ie`: used to analyze Internet Explorer's behavior when opening the** given file (e.g. browser exploits).
- `bin`: used to analyze generic binary data, such as **shellcodes**.

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

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

```
$ python submit.py --package <package name> /path/to/malware
```

If none is specified, Cuckoo will try to detect the file type and select the correct analysis package accordingly. If the file type is not supported by default the analysis will be aborted, therefore you are always invited to specify the package name whenever it's possible.

2.3.4 Analysis Results

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

Following is an example of an analysis directory structure:

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

analysis.conf

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

analysis.log

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

dump.pcap

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

files/

This directory contains all the files the malware operated on and that Cuckoo was able to dump.

logs/

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

reports/

This directory contains all the reports generated by Cuckoo as explained in the Configuration chapter.

shots/

This directory contains all the screenshots of the guest's desktop taken during the malware execution.

2.3.5 Utilities

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

Cleanup utility

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

Note: Running clean.sh will delete: * Analyses * Binaries * Cuckoo task's database * Cuckoo logs

To clean your setup, run:

```
$ cd utils
$ sh clean.sh
```

Submission Utility

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

Web Utility

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

Test Report Utility

Run 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:

```
$ cd utils
$ python testreport.py ../storage/analyses/1/
```

Test Signature Utility

Run the signature engine (checks all signatures) on an already available analysis folder and see possible matches. This is used mainly in debugging and developing Cuckoo and testing new signatures. For example if you want run again the singature engine for analysis number 1:

```
$ cd utils
$ python testsignatures.py ../storage/analyses/1/
```

Community Download Utility

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

```
$ cd utils
$ python community.py
You need to enable some category!

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

optional arguments:
  -h, --help        show this help message and exit
  -a, --all         Download everything
  -s, --signatures  Download Cuckoo signatures
  -f, --force       Install files without confirmation
  -w, --rewrite     Rewrite existing files
```

Example: install all available signatures:

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

2.4 Customization

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

2.4.1 Machine Managers

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

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

A basic machine manager could look like:

```
from lib.cuckoo.common.abstracts import MachineManager
from lib.cuckoo.common.exceptions import CuckooMachineError

class MyManager(MachineManager):
    def start(self, label):
        try:
            start(label)
        except SomethingBadHappens as e:
            raise CuckooMachineError("OPS!")

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

The only requirements for Cuckoo are that:

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

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

Configuration

Every machine manager module should come with a dedicated configuration file located in *conf/<machine manager name>.conf*. For example for *modules/kvm.py* we have a *conf/kvm.conf*.

The configuration file should follow the default structure:

```
[kvm]
# Specify a comma-separated list of available machines to be used. For each
# specified ID you have to define a dedicated section containing the details
# on the respective machine. (E.g. cuckoo1,cuckoo2,cuckoo3)
machines = cuckoo1

[cuckoo1]
# Specify the label name of the current machine as specified in your
# libvirt configuration.
label = cuckoo1

# Specify the operating system platform used by current machine
# [windows/darwin/linux].
platform = windows
```

```
# Specify the IP address of the current machine. Make sure that the IP address
# is valid and that the host machine is able to reach it. If not, the analysis
# will fail.
ip = 192.168.122.105
```

A main section called `[<name of the module>]` with a `machines` field containing a comma-separated list of machines IDs.

For each machine you should specify a `label`, a `platform` and it's `ip`.

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

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

2.4.2 Analysis Packages

As explained in Analysis Packages, analysis packages are structured Python classes that describe how Cuckoo's analyzer component should conduct the analysis procedure for a given file inside the guest environment.

As you already know, you can create your own packages and add them along with the default ones. Designing new packages is very easy and requires just a minimal understanding of programming and of the Python language.

Getting started

As an example we'll take a look at the default package for analyzing generic Windows executables (located at *analyzer/windows/packages/exe.py*):

```
1  from lib.common.abstracts import Package
2  from lib.api.process import Process
3
4  class Exe(Package):
5      """EXE analysis package."""
6
7      def start(self, path):
8          p = Process()
9
10         if "arguments" in self.options:
11             p.execute(path=path, args=self.options["arguments"], suspended=True)
12         else:
13             p.execute(path=path, suspended=True)
14
15         p.inject()
16         p.resume()
17
18         return p.pid
19
20     def check(self):
21         return True
22
23     def finish(self):
24         return True
```

Let's walk through the code.

At line **1** we import the parent class `Package`, all analysis packages must inherit this abstract class otherwise Cuckoo won't be able to load them. At line **2** we import the class `Process`, which is an API module provided by Cuckoo's Windows analyzer for accessing several process-related features.

At line **4** we define our class.

At line **7** we define the `start()` function, at line **20** the `check()` function and at line **23** the `finish()` function. These three functions are required as they are used for customizing the package's operations at three different stages of the analysis.

In this case we just create a `Process` instance, check if the user specified any arguments as option and launch the malware located at `path`, which then gets injected and resumed.

start()

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

check()

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

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

For example:

```
def check(self):
    if os.path.exists("C:\\config.bin"):
        return False
    else:
        return True
```

This `check()` function will cause Cuckoo to immediately terminate the analysis whenever *C:config.bin* is created.

finish()

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

Options

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

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

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

The analysis package selected will have access to these values:

```
from lib.common.abstracts import Package

class Example(Package):

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

    def check():
        return True

    def finish():
        return True
```

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

Process API

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

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

You then initialize an instance with:

```
p = Process()
```

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

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

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

open()

This method allows you to open an handle to a running process:

```
p = Process(pid=1234)
p.open()
handle = p.h_process
```

Return: True/False in case of success or failure of the operation.

exit_code()

This method allows you to acquire the exit code of a given process:

```
p = Process(pid=1234)
code = p.exit_code()
```

If it wasn't already done before, `exit_code()` will perform a call to `open()` in order to acquire an handle to the given process.

Return: process exit code (ulong).

is_alive()

This method simply calls `exit_code()` and verify if the returned code is `STILL_ACTIVE`, meaning that the given process is still running:

```
p = Process(pid=1234)
if p.is_alive():
    print("Still running!")
```

execute()

This method simply allows you to execute a process. It accepts the following arguments:

- `path`: path to the file to execute.
- `args`: arguments to pass at process creation.
- `suspended`: (True/False) boolean saying if the process should be created in suspended mode or not (default is False)

Example:

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

Return: True/False in case of success or failure of the operation.

resume()

This method resumes a process from a suspended state.

Example:

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

terminate()

This method allows you to terminate any given process:

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

Return: True/False in case of success or failure of the operation.

inject()

This method allows you to inject a DLL file into a given process. You can specify the following arguments:

- `dll`: path to the DLL to inject, if none is specified it will use Cuckoo's default DLL.
- `apc`: True/False in case you want to use *QueueUserAPC* injection or not. Default is False, which will result in a *CreateRemoteThread* injection. If you try to inject a process that you created in suspended more, *QueueUserAPC* injection will be automatically selected.

Example:

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

Return: True/False in case of success or failure of the operation.

dump_memory()

This method allows you to take a snapshot of the given process' memory space. When invoked, it will create a result folder called *memory/<pid>/<timestamp>/* containing all the dumps sorted as *<memory region address>.dmp* (e.g. *0x12345678.dmp*).

Example:

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

Return: True/False in case of success or failure of the operation.

2.4.3 Processing Modules

Cuckoo's processing modules are Python scripts that let you define custom ways for analyzing the raw results generated by the sandbox and append some information to a global container that will be later used by 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.

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 contains all the abstracted results produced by all the modules sorted by their defined keys.

Cuckoo is already provided with 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 report.

Following is a JSON-like representation of a default global container:

```
{
    "info": {
        "started": <timestamp>,
        "ended": <timestamp>,
        "duration": <duration in seconds>,
        "version": <version of Cuckoo>
    },
    "signatures": [
        {
            "severity": <severity level>,
            "description": <signature description>
            "alert": <boolean value>,
            "references": [<any reference link>],
            "data": [<any contextual data>],
            "name": <signature name>
        }
    ],
    "behavior": {
        "processes": [
            {
                "parent_id": <parent PID>,
                "process_name": <process name>,
                "process_id": <PID>,
                "first_seen": <timestamp when the process was first seen>,
                "calls": [
                    {
                        "category": <API function category>,
                        "status": <SUCCESS or FAILURE>,
                        "return": <any returned value>,
                        "timestamp": <timestamp of the call>,
                        "repeated": <how many times it was repeated consecutively>,
                        "api": <API function>,
                        "arguments": [
                            {
                                "name": <argument name>,
                                "value": <argument value>
                            }
                        ]
                    },
                    <...>
                ],
                <...>
            }
        ],
        "processtree": [
            {
                "pid": <PID>,
                "name": <process name>,
                "children": [<recursive child entries>]
            }
        ],
        "summary": {
            "files": [<list of files accessed>],
            "keys": [<list of registry keys accessed>],
            "mutexes": [<list of mutexes accessed>]
        }
    },
    "static": {<static analysis if available for the file type>},
```

```
    "dropped": [
        {
            "size": <file size>,
            "sha1": <SHA1 hash>,
            "name": <file name>,
            "type": <file type>,
            "crc32": <CRC32 hash>,
            "ssdeep": <Ssdeep hash>,
            "sha256": <SHA256 hash>,
            "sha512": <SHA512 hash>,
            "md5": <MD5 hash>
        },
        <...>
    ],
    "file": {
        "size": <file size>,
        "sha1": <SHA1 hash>,
        "name": <file name>,
        "type": <file type>,
        "crc32": <CRC32 hash>,
        "ssdeep": <Ssdeep hash>,
        "sha256": <SHA256 hash>,
        "sha512": <SHA512 hash>,
        "md5": <MD5 hash>
    },
    "debug": {
        "log": <content of analysis.log>
    },
    "network": {
        "http": [
            {
                "body": <request body>,
                "uri": <request URI>,
                "method": <request method>,
                "host": <host name>,
                "version": <HTTP version>,
                "path": <path of the request>,
                "data": <dump of whole request>,
                "port": <port>
            },
            <...>
        ],
        "udp": [
            {
                "dport": <destination port>,
                "src": <source IP>,
                "dst": <destination IP>,
                "sport": <source port>
            },
            <...>
        ],
        "hosts": [<list of involved IP addresses>],
        "dns": [
            {
                "ip": <IP address>,
                "hostname": <domain name>
            },
        ],
```

```
        "tcp": [
            {
                "dport": <destination port>,
                "src": <source IP>,
                "dst": <destination IP>,
                "sport": <source port>
            },
            <...>
        ]
    }
}
```

Every processing module added will end up with a dedicated dictionary entry in this data structure.

Getting started

All processing modules are and should be placed in *modules/processing/*. In this directory you will find a set of default modules that are used to produce the traditional Cuckoo analysis reports.

A basic processing module could look like:

```
1  from lib.cuckoo.common.abstracts import Processing
2
3  class MyModule(Processing):
4
5      def run(self):
6          self.key = "file"
7          data = do_something()
8          return data
```

Every processing module should contain:

- A class inheriting `Processing`.
- A `run()` function.
- A `self.key` attribute defining the name to be used as a subcontainer for the returned data.
- A set of data (list, dictionary or string etc.) that will be appended to the global container.

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

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

```
1  from lib.cuckoo.common.abstracts import Processing
2
3  class MyModule(Processing):
4      order = 2
5
6      def run(self):
7          self.key = "file"
8          data = do_something()
9          return data
```

The processing modules are provided with some attributes that can be used to access the raw results for the given analysis:

- `self.analysis_path`: path to the folder containing the results (e.g. *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.

Example

A good example to understand better the mechanics behind this is the Yara module. Yara is a tool and library used to match user's defined signatures containing static binary patterns against the analyzed file.

```
1  import os
2  import logging
3
4  try:
5      import yara
6      HAVE_YARA = True
7  except ImportError:
8      HAVE_YARA = False
9
10 from lib.cuckoo.common.constants import CUCKOO_ROOT
11 from lib.cuckoo.common.abstracts import Processing
12
13 log = logging.getLogger(__name__)
14
15 class YaraSignatures(Processing):
16     """Yara signature processing."""
17
18     def run(self):
19         """Run Yara processing.
20         @return: hash with matches.
21         """
22         self.key = "yara"
23         matches = []
24
25         if HAVE_YARA:
26             try:
27                 rules = yara.compile(filepath=os.path.join(CUCKOO_ROOT, "data", "yara", "index.y
28                 for match in rules.match(self.file_path):
29                     matches.append({"name" : match.rule, "meta" : match.meta})
30             except yara.Error as e:
31                 log.warning("Unable to match Yara signatures: %s" % e[1])
32         else:
33             log.warning("Yara is not installed, skip")
34
35         return matches
```

As you can see in line #22 we defined the key name for the module. Next in the `run()` function we compile the signatures file and match every signature against the file located at `self.file_path`. The matched signatures are appended in the `matches` dictionary which is then returned and that will be included in the global container under the section "yara".

2.4.4 Signatures

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

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

Getting started

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

All signatures are and should be located in *modules/signatures/*.

A basic example signature is the following:

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

class CreatesExe(Signature):
    name = "creates_exe"
    description = "Creates a Windows executable on the filesystem"
    severity = 2

    def run(self, results):
        for file_name in results["behavior"]["summary"]["files"]:
            if file_name.endswith(".exe"):
                self.data.append({"file_name" : file_name})
                return True

        return False
```

As you can see the structure is very simple and similar to the other modules types we've seen so far.

You need to declare your class inheriting `Signature`, for which you can define some generic attributes:

- `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).
- `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)

The `run()` function takes the previously generated global container (see Processing Modules) and performs some checks against it.

In the given example it just walks through all the accessed files in the summary and checks if there is anything ending with ".exe": in that case it will return `True`, meaning that the signature was matched. Otherwise return `False`.

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

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

2.4.5 Reporting Modules

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

Getting Started

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

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

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

Every additional option you add to your section will be available to your reporting module in the `self.options` dictionary.

Following is an example of a working JSON reporting module:

```
1  import os
2  import json
3
4  from lib.cuckoo.common.abstracts import Report
5  from lib.cuckoo.common.exceptions import CuckooReportError
6
7  class JsonDump(Report):
8      """Saves analysis results in JSON format."""
9
10     def run(self, results):
11         """Writes report.
12         @param results: Cuckoo results dict.
13         @raise CuckooReportError: if fails to write report.
14         """
15         try:
16             report = open(os.path.join(self.reports_path, "report.json"), "w")
17             report.write(json.dumps(results, sort_keys=False, indent=4))
18             report.close()
19         except (TypeError, IOError) as e:
20             raise CuckooReportError("Failed to generate JSON report: %s" % e)
```

This code is very simple, it basically just receives the global container produced by the processing modules, converts it into JSON and writes it to a file.

There are few requirements for writing a valid reporting module:

- Declare your class inheriting `Report`.
- Have a `run()` function performing the main operations.
- Try to catch most exceptions and raise `CuckooReportError` to notify the issue.

All reporting modules have access to some attributes:

- `self.analysis_path`: path to the folder containing the raw analysis results (e.g. *storage/analyses/1/*)
- `self.reports_path`: path to the folder where the reports should be written (e.g. *storage/analyses/1/reports/*)
- `self.options`: a dictionary containing all the options specified in the report's configuration section in *conf/reporting.conf*.

2.5 Development

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

2.5.1 Development Notes

Git branches

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

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

Release Versioning

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

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

Ticketing system

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

Contribute

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

2.5.2 Coding Style

In order to contribute code to the project, you must diligently follow the style rules describe in this chapter. Having a clean and structured code is very important for our development lifecycle, and not compliant code will most likely be rejected.

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

Formatting

Copyright header

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

```
# Copyright (C) 2010-2012 Cuckoo Sandbox Developers.
# This file is part of Cuckoo Sandbox - http://www.cuckoosandbox.org
# See the file 'docs/LICENSE' for copying permission.
```

Indentation

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

Maximum Line Length

Limit all lines to a maximum of 79 characters.

Blank Lines

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

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

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

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

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

Imports

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

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

NOT:

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

Always specify explicitly the objects to import:

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

NOT:

```
from lib import *
```

Strings

Strings must be delimited by double quotes (").

Printing and Logging

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

In your module add:

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

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

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

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

NOT the statement:

```
print "foo"
```

Checking for keys in data structures

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

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

Exceptions

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

Following is current Cuckoo's exceptions chain:

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

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

Naming

Custom exceptions name must prefix with "Cuckoo" and end with "Error" if it represents an unexpected malfunction.

Exception handling

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

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

NOT:

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

It's a good practice use "e" instead of "e.message", as in the example above.

Documentation

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

Automated testing

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

2.6 Final Remarks

2.6.1 Links

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

2.6.2 Join the discussion

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

Mailing list how to

Cuckoo's official mailing list require registration, so you have to register your email address before sending mails. Please make sure you registered with the email address you're trying to post with.

Please respect netiquette when posting, in detail:

- Before posting read the mailing list archives, read the Cuckoo blog, read the documentation and Google about your issue. Stop posting questions that have already been answered over and over everywhere.
- Posting emails saying just like "Doesn't work, help me" are completely useless. If something is not working report the error, paste the logs, paste the config file, paste the information on the virtual machine, paste the results of the troubleshooting, give context. We are not wizards and we don't have the crystal ball.
- Use a proper title. Stuff like "Doesn't work", "Help me", "Error" is not a proper title.
- Tend to use pastebin.com or pastie.org and similar services to paste logs and configs: make the email more readable.
- Tend to upload your attachment to file upload services, we have a very low attachment size limit.
- Tend to not write HTML emails.

2.6.3 Donations

Cuckoo is software released freely to the public and developed during spare time by volunteers only. If you enjoy it and want to see it kept being developed and updated, please consider making a donation.

We receive small donations through Flattr.

If you want to make a larger donation or provide a different form of support (such as hardware, connectivity, hosting or else) you can contact us at `donations at cuckoobox dot org`.

2.6.4 People

Cuckoo Sandbox is an open source project result of the efforts and contributions of a lot of people who enjoyed volunteering some of their time for a greater good :).

Active Developers

Name	Role	Contact
Claudio "*nex*" Guarnieri	Lead Developer	`nex at cuckoobox dot org`
Alessandro "*jekil*" Tanasi	Developer	`alessandro at tanasi dot it`
Jurriaan "*skier*" Bremer	Developer	`jurriaanbremer at gmail dot com`
Mark "*rep*" Schloesser	Developer	

Inactive Developers

- Dario Fernandes

Contributors

- Thorsten Sick (Various patches and contributions)
- Adam Pridgen (Various patches and contributions)
- Mike Tu (Initial VMWare Workstation machine manager)
- Loic Jaquemet (Improvements in VirtualBox machine manager)

Bug Reporters/Advisors

- Giacomo Milani
- Stefan Hausotte
- Felix Leder
- Georg Wicherski
- Kjell Christian Nilsen
- Carsten Willems

2.6.5 Supporters

- The Honeynet Project
- The Shadowserver Foundation

2.6.6 Sponsors

- Rapid7