Cuckoo Sandbox Book

Release 0.3

Cuckoo Sandbox

December 26, 2011

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 Windows environment.

This book explains what Cuckoo is, how it works and what you can do with it, from setup and run Cuckoo to how to customize it and extend it.

CHAPTER

ONE

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 Usage questions

How to start an analysis?

You can simply start an analysis via command-line utility `submit.py`. Check *Submit an analysis*.

How to change Cuckoo default behaviour?

Depending on what you mean, you can edit Cuckoo's configuration files (see *Configuration*) or work on the analysis packages (see *Analysis Packages*).

1.1.2 General questions

Can I redistribute Cuckoo Sandbox?

Yes, you can. Cuckoo Sandbox is distributed under the GNU General Public License version 3. See *License*.

Can I include Cuckoo Sandbox in my closed source commercial product?

Generally no, you can't. Cuckoo Sandbox is distributed under the GNU General Public License version 3. See *License*.

I want to help Cuckoo, what can I do?

Your help is very appreciated, you can help Cuckoo Sandbox in several ways, from coding to send bug reports. See *Final Remarks*.

I want to help but I don't have time

There are many ways to help Cuckoo: coding, testing, reporting bugs, donating money or hardware, reviewing code and documentation or submitting feature requests or feedback. Just do whatever you feel could help the project with your possibilities.

Otherwise you can ask to the developers and to other Cuckoo users in the #cuckoobox IRC channel.

CHAPTER

TWO

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.
- Network traffic trace in PCAP format.
- Screenshots of Windows desktop taken during the execution of the malware.
- Traces of assembly instructions performed by 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 still mantains it 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 as 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.

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 submission and processing automation capabilities.

It can be used to analyze:

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

Thanks to its scripting and customization capabilities there's basically no limit to what you can achieve with Cuckoo, for example automating malware unpacking or automating the dump of configuration files and web-injects from banking trojans.

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 and execution process, while the Guests are the isolated environments where the malwares get actually safely executed and analyzed.

The following picture explains Cuckoo's architecture:

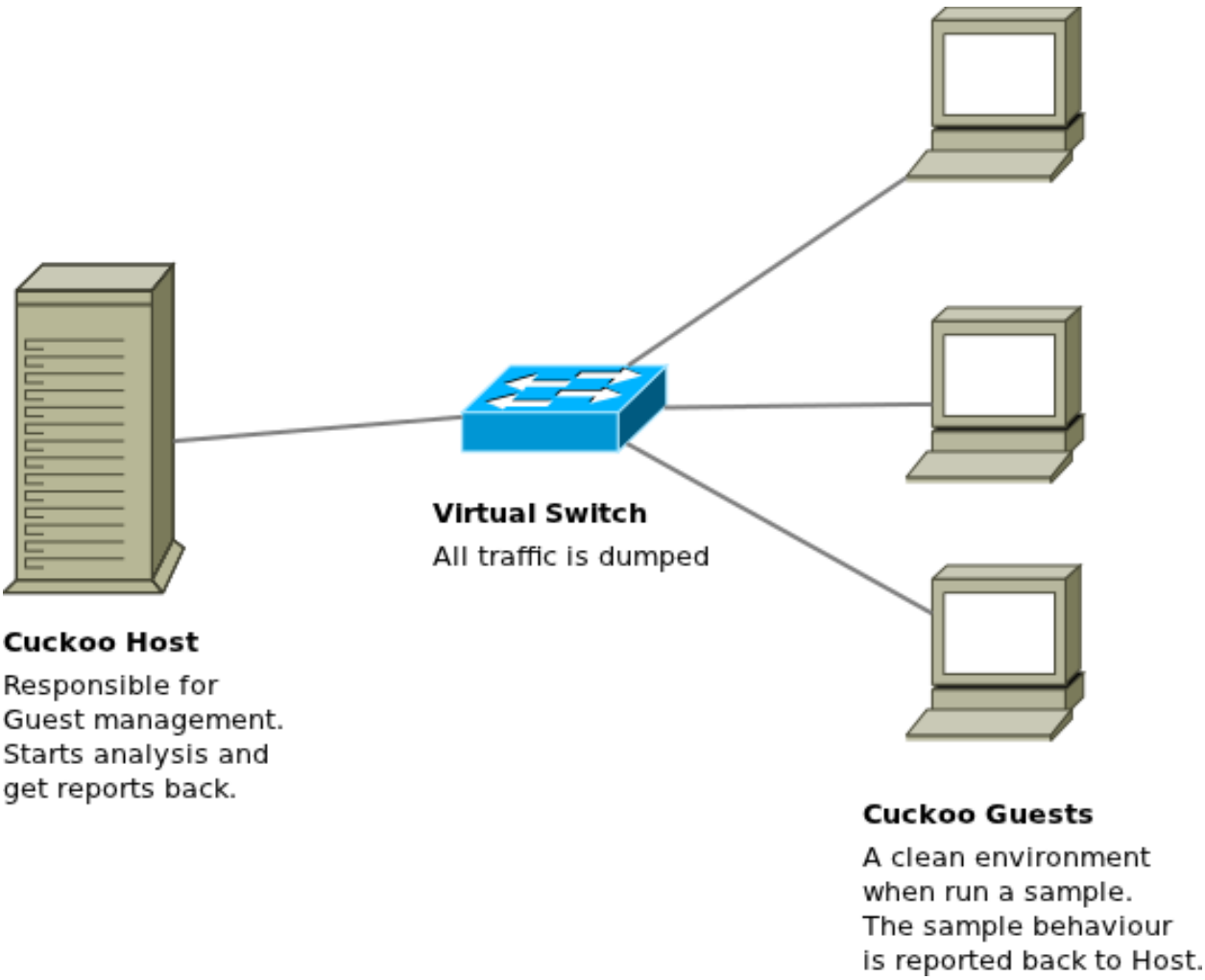
Although recommended setup is *GNU/Linux* (Ubuntu preferrably) as host and *Windows XP Service Pack 3* as guest, Cuckoo proved to work smoothly also on *Mac OS X* as host and *Windows Vista* and *Windows 7* as guests.

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


Virtual Switch
All traffic is dumped
Cuckoo Host
Responsible for
Guest management.
Starts analysis and
get reports back.
Cuckoo Guests
A clean environment
when run a sample.
The sample behaviour
is reported back to Host.

Cuckoo Sandbox is free software: you can redistribute it and/or modify it under the terms of GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your opinion) any later version.

See the GNU General Public License for more details.

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 software 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
```

Cuckoo makes use of several libraries which include:

- Magic: for detecting file types.
- Dpkt: for extracting relevant information from PCAP files.
- Mako: for rendering the HTML reports and the web interface.

On Ubuntu you can install all of them with the following command:

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

On different distributions refer to the provided official homepage to retrieve other installers or sources.

Other optional libraries, which do not affect Cuckoo's execution, include:

- Pyssdeep: for calculating ssdeep fuzzy hash of files.

Installing VirtualBox

At current stage, Cuckoo heavily relies on VirtualBox as it's unique virtualization engine.

Despite being often packaged by all GNU/Linux distributions, you are encouraged to download and install the latest version from the official website. The reason behind this choice is that packaged versions of VirtualBox (called OSE) generally have some limitations or adjustments in order to meet requirements of the GNU GPL license.

You can get the proper package for your distribution at the official download page.

The installation of VirtualBox is not in purposes of this documentation, if you are not familiar with it please refer to the official documentation.

Installing Tcpdump

By default Cuckoo makes use of VirtualBox's embedded network tracing functionalities, but in some cases or some network configurations you might need to adopt an external network sniffer.

If you intend to use VirtualBox's own network trace, you can skip this section.

The best choice for packet interception is tcpdump of course.

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
```

Installing Cuckoo

Proceed with download and installation.

Create a user

Even if you obviously can run Cuckoo with your current user, creating a dedicated one is always a good practice.

Create a new user:

```
$ sudo adduser cuckoo
```

Make sure the new user belongs to the "vboxusers" group (or the group you used to run VirtualBox):

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

Download Cuckoo

You can get your copy of Cuckoo from the official website or from our git repository.

Please notice that the archives to be downloaded from the website are core releases, while the version on git has to be considered an **under development** stage, therefore possibly unstable and not yet fully documented.

Install it

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

Configuration

Cuckoo relies on two main configuration files:

- *cuckoo.conf*: for configuring general behavior and analysis options.
- *reporting.conf*: for enabling or disabling report formats.

cuckoo.conf

We'll first start editing *conf/cuckoo.conf* walking through every section and option available.

Logging Following is the logging section:

```
[Logging]
# Enable/Disable additional debugging messages. This messages won't wrote to
# log file but just printed on screen. [on/off]
debug = off
```

The **debug** option enables or disables debug messages that will be both printed on standard output as well as stored in the log file.

Analysis Following is the analysis section:

```
[Analysis]
# This is the actual analysis timeout (expressed in seconds). This represents
# the default timeout performed by analysis core if none is specified.
analysis_timeout = 200
# Watchdog timeout (expressed in seconds) for analysis execution to complete,
# when this timeout gets hit, current execution is aborted and virtual machine
# is restored and freed.
watchdog_timeout = 600
# Specify here the path where analysis results shall be stored.
results_path = analysis/
```

This section defines two analysis time boundaries:

- **analysis timeout**: this timeout represent the maximum time an analysis should last, it can be overridden when submitting a file to analyze.

- **watchdog timeout**: this is the time limit for which Cuckoo host should wait for the guest component (*analyzer*) to terminate its operations.

The *analysis timeout* should be smaller than the *watchdog timeout*. If by mistake it's configured differently, Cuckoo will force the *analysis timeout* to a smaller value.

Consider that the *watchdog timeout* should be raised just under critical circumstances, where the analyzer or virtual machine are not responding and therefore need to be killed. When this happens, you'll most likely lose any analysis results from that run.

The **results_path** option defines where to store the analysis results.

Processing Following is the processing section:

```
[Processing]
# Specify here the interpreter path to be used to launch the script.
interpreter = /usr/bin/python
# Specify here the path to the analysis results processing script.
processor = processor.py
```

This section defines where the post-analysis processing script is located and how it should be executed.

This script should be your interface to the analysis results and you should use it and customize it at your will in order to consume the data generated by Cuckoo. We'll get more into details on this in the *Customization* chapter.

By default Cuckoo provides a Python processing script that invokes some Python classes used to process the results and to generate human readable analysis reports (text, HTML, JSON).

The **interpreter** option defines the path to the application to be used to execute the script.

The **processor** option defines the path to the script to be executed.

Sniffer Following is the sniffer section:

```
[Sniffer]
# Enable or disable the following option by assigning a True or False value.
# In case you decide to disable it, you're supposed to either not have any
# network dump or to used VirtualBox's (or any other virtualization engine
# you are using) to handle the network monitoring instead of using an external
# sniffer such as tcpdump. [on/off]
sniffer = off
# Path to the sniffer (tcpdump) binary.
path = /usr/sbin/tcpdump
# This specifies the network interface where the sniffer will bind to in order
# to monitor virtual machines' generated traffic.
interface = eth0
```

This section should be considered and edited just in the case you decided to use an external sniffer (assuming that you *properly installed* it already).

If otherwise you don't plan to use an external sniffer, you can skip this section.

First you'll need to enable the **snifer** option by setting it to "*on*".

The **path** option defines where the sniffer (tcpdump) binary is located. It should be generally correct by default.

The **interface** option defines which network interface the sniffer should monitor. This obviously depends on your network configuration and on how you are planning to configure your virtual machines' networking. It's up to you.

Virtual Machines Following is the Virtual Machines section:

```
[VirtualMachines]
# Virtualization product.
engine = VirtualBox
# List virtual machines IDs separated by commas.
enabled = cuckoo1
# Set to "gui" if you want Cuckoo to spawn virtual machines' GUIs or set to
# "headless" if you don't.
mode = gui
# Path to local Python installation on guest machines. Please be sure to have
# correctly set this value as it's critical to Cuckoo's proper execution.
python = C:\Python27\python.exe
```

This is probably the most important section in the configuration file, as it defines the core options for the virtualization engine.

The **engine** option defines which virtualization module to use. At current stage only VirtualBox is supported, therefore you shouldn't modify this option unless you really know what you're doing.

The **enabled** option defines a comma-separated list of enabled virtual machines.

Note: The virtual machines' IDs used by Cuckoo are user-defined names that are exclusively used internally by Cuckoo. They are **not** the names used to label the virtual machines inside VirtualBox. Even if they could have the same values (*not recommended*), it's important to understand that they are not the same thing.

The **mode** option defines if the virtualization software should spawn the machines in *gui* mode (with regular window) or in *headless*, which will not create any graphical interface.

The **python** option defines the location of the Python interpreter *inside the virtualized Windows environment*. This is critical to proper execution of Cuckoo, so take care to use the path you define here when installing Python on Windows or to come back here later and modify this value accordingly.

Virtual machines details For each virtual machine you specified in the comma-separated list in the **enabled** option of the previous section, you have to create a dedicated section named with the ID value you assigned in the list.

An example of such section is:

```
[cuckoo1]
name = Cuckoo1
username = Me
password = cuckoo
# Please notice that the shared folder name must coincide with the current
# virtual machine id, which is the name you assigned between the square
# brackets (e.g. [cuckoo1]).
share = shares/cuckoo1
```

As you notice the section name **[cuckoo1]** has to contain the ID you assigned to the virtual machine.

The **name** option is the name you're going to use to create the virtual machine in VirtualBox.

The **username** option defines the name of the Windows account you're going to create.

The **password** option defines the password for such Windows account.

Note: The Windows account is mandatory. It is needed to allow the host to execute commands inside the guest operating system, therefore the *username* and *password* options must containd valid values.

The **share** option defines the path to the shared folder you're going to assign to this specific virtual machine. This folder has to exist, therefore make sure to create it. The name of such folder must coincide with the ID you assigned to current virtual machine. In the example given, the current virtual machine ID is "*cuckoo1*", so the shared folder is named "*cuckoo1*" as well.

If for example you defined more than one virtual machine in the *enabled* option (e.g. "cuckoo1,cuckoo2") you'll have to create multiple details sections like:

```
[cuckoo1]
name = Cuckoo1
username = Me
password = cuckoo
share = shares/cuckoo1

[cuckoo2]
name = Cuckoo2
username = Me
password = cuckoo
share = shares/cuckoo2
```

reporting.conf

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

It contains the following section:

```
[Tasks]
# Enable/Disable reporting tasks.
# Here you can choose what report enable or disable.
# By default all available reporting tasks are enabled.
# Available values are [on/off]
jsondump = on
reporttxt = on
reporthtml = on
```

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* VirtualBox you can proceed on creating all the virtual machines you need.

To do so you can either use the graphical user interface or the powerful command-line utility *VBoxManage* provided by VirtualBox.

Consider that the use of VirtualBox is not in the purposes of this documentation so please refer to the official documentation for it.

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.

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 Guest Additions

VirtualBox's Guest Additions provide some additional functionalities that allow the host and the guests to interact easily.

They are required for:

- Time synchronization.
- Shared folders.
- Executing processes in the guest.

You can get details on how to install them from the dedicated chapter in the official documentation.

Install Python

Python is a strict requirement for the Cuckoo guest component (*analyzer*) 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 optionals 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.
- WinAppDbg: it's used by the *tracer package* to dump assembly instructions.

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

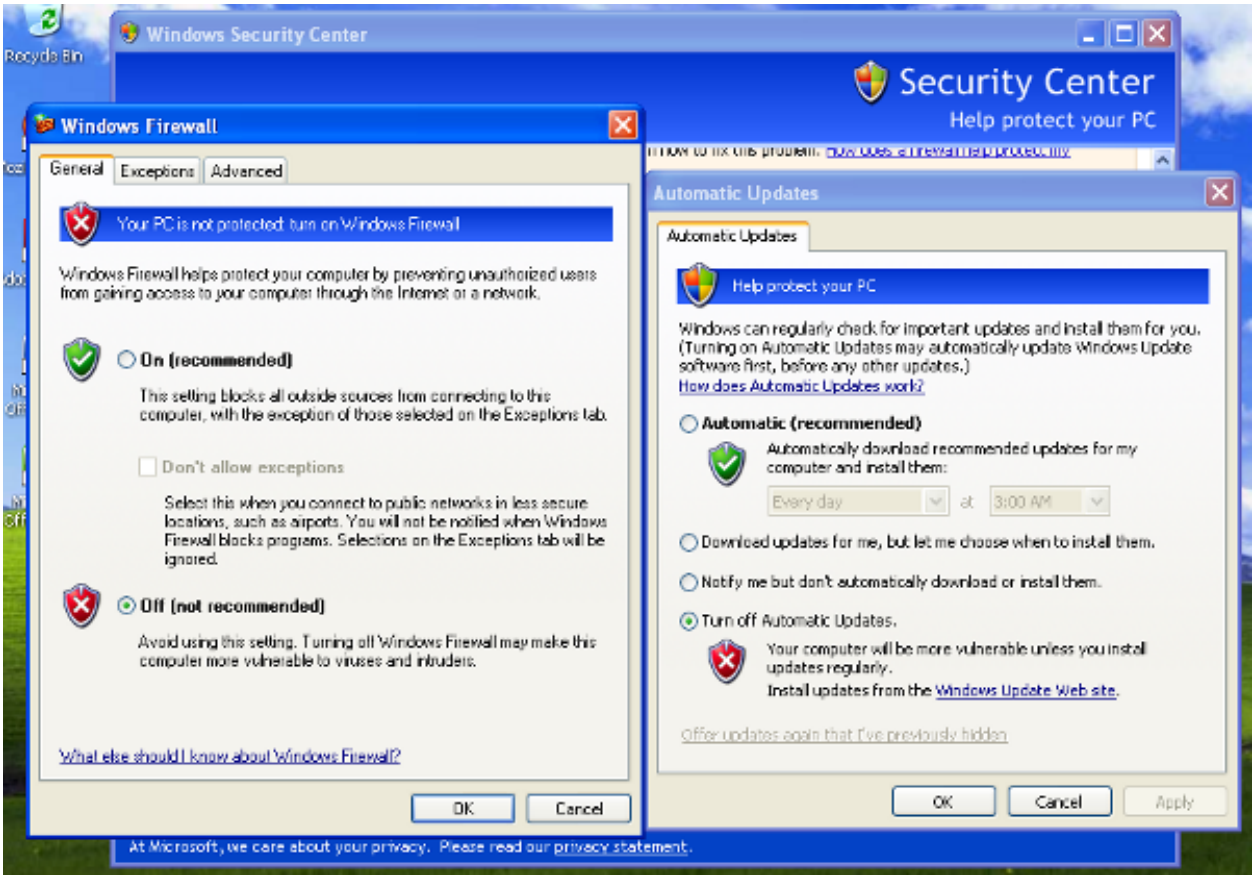
Now it's the time to setup the network configuration for your virtual machine.

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.

By default VirtualBox adopts Network Address Translation (NAT) which in most cases will be good enough for any needs. This is also the configuration we'll adopt in this documentation.

If you have particular needs and want to use some different networking, please refer to VirtualBox's virtual networking documentation.

Network Tracing

Unless you decided to use an external sniffer (as previously discussed in *Configuration*), you can proceed configuring the network trace functionality provided by VirtualBox as explained here.

First you need to power off your virtual machine:

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

Then you enable network trace:

```
$ VBoxManage modifyvm "Name of VM" --nictrace1 on --nictracefile1 /path/to/cuckoo/shares/<VM ID>/dump
```

The last argument specifies the path where the PCAP file will be stored. It has to be an absolute path and include the file name as well. In order to make Cuckoo able to find the file you'll have to specify the shared folder you created for current virtual machine and "*dump.pcap*".

Shared Folders

Cuckoo exchanges data between the host and the guest using VirtualBox's Shared Folders.

In order to have them enabled on your virtual machine you should have installed the Guest Additions as specified in *Requirements*.

You will have to add two shared folders:

- **shares/setup**: which is used to get Cuckoo analyzer's components to be run inside virtualized Windows.
- **shares/<VM ID>**: the unique shared folder associated with your current Virtual Machine, which is used to store the analysis results.

You can do so from VirtualBox's graphical user interface or from the command line:

```
$ VBoxManage sharedfolder add "<Name of VM>" --name "setup" --hostpath "/path/to/cuckoo/shares/setup"
$ VBoxManage sharedfolder add "<Name of VM>" --name "<VM ID>" --hostpath "/path/to/cuckoo/shares/<VM
```

Where "*<Name of VM>*" is the label you gave to the virtual machine in VirtualBox and "*<VM ID>*" is the ID you assigned to the Virtual Machine in Cuckoo.

Using the GUI, you should see something similar to this:

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 and with your Windows user logged in. Once you are sure that the operating system is fully booted and ready to be snapshotted you can proceed.

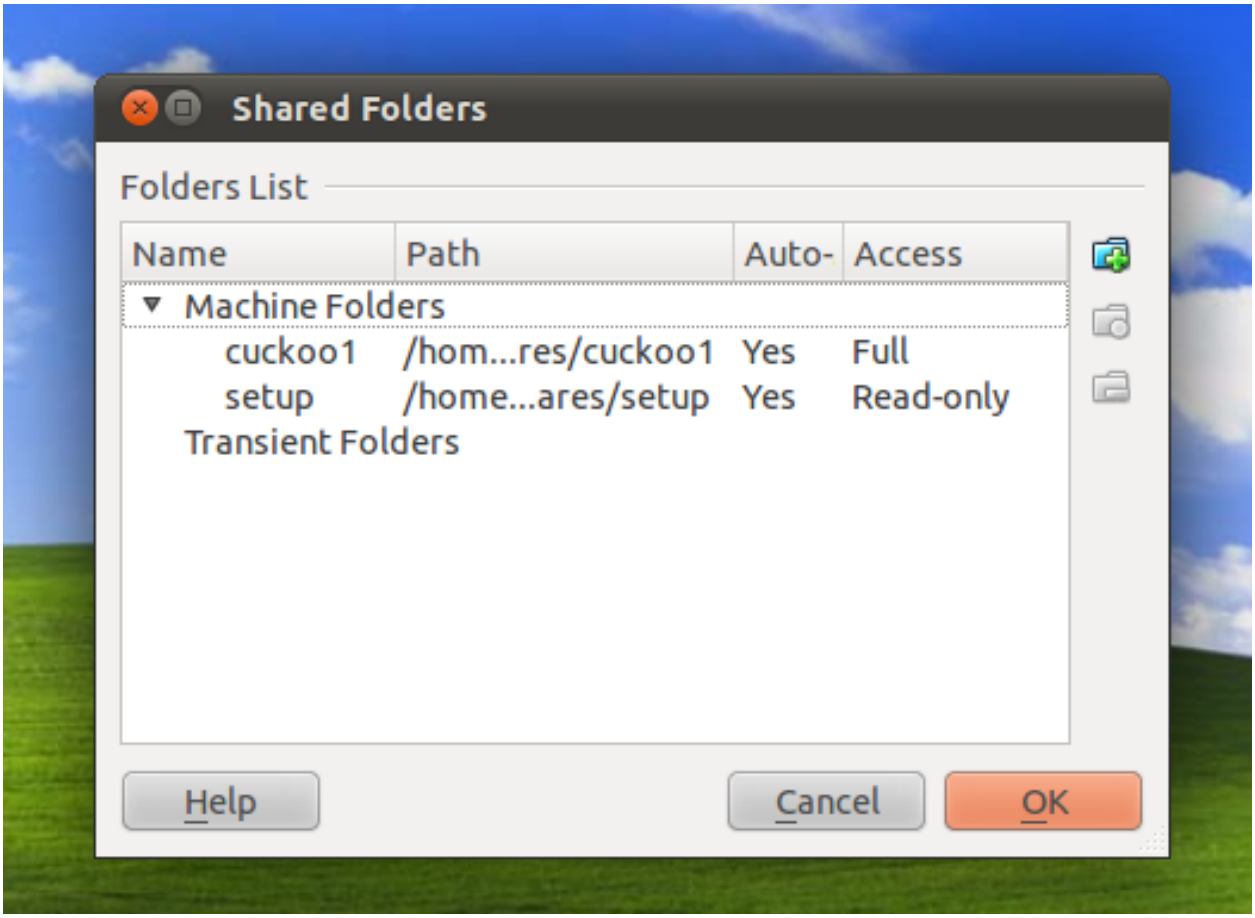
You can take the snapshot from the graphical user interface or from the command line:

```
$ 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
```

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


Shared Folders
Folders List
Name
Path
Auto-
Access
Machine Folders
cuckoo1
/hom...res/cuckoo1
Yes
Full
setup
/home...ares/setup
Yes
Read-only
Transient Folders
Help
Cancel
OK

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.

To clone a machine you can use the graphical user interface (at least in the most recent versions of VirtualBox) or from the command line:

```
$ VBoxManage clonevm "<Name of original VM>" --name "<Name of new VM>" --registervm
```

Now you have an exact copy of your original virtual machine saved with the new name you specified.

Obviously the new virtual machine will 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*, *Shared Folders* and *Saving the Virtual Machine* for this new machine.

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:

```
                       _
     _____ _   _  _____| |  _ ____   ____
    / ___) | | |/ ___) |_/ ) _ \ / _ \
   ( (___| |_| ( (___|  _ ( |_| | |_| |
    \_____)_____/ \_____)_| \_)___/ \___/ v0.3

 www.cuckoobox.org
 Copyright (C) 2010-2011

[2011-12-18 17:43:06,343] [Core.Init] INFO: Started.
[2011-12-18 17:43:06,834] [VirtualMachine.Check] INFO: Your VirtualBox version is: "4.1.6", good!
[2011-12-18 17:43:06,834] [Core.Init] INFO: Populating virtual machines pool...
[2011-12-18 17:43:06,840] [VirtualMachine.Infos] INFO: Virtual machine "Cuckoo1" information:
[2011-12-18 17:43:06,841] [VirtualMachine.Infos] INFO:      \_| Name: Cuckoo1
[2011-12-18 17:43:06,841] [VirtualMachine.Infos] INFO:        | ID: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx
[2011-12-18 17:43:06,841] [VirtualMachine.Infos] INFO:        | CPU Count: 1 Core/s
[2011-12-18 17:43:06,841] [VirtualMachine.Infos] INFO:        | Memory Size: 192 MB
[2011-12-18 17:43:06,841] [VirtualMachine.Infos] INFO:        | VRAM Size: 16 MB
[2011-12-18 17:43:06,841] [VirtualMachine.Infos] INFO:        | State: Saved
[2011-12-18 17:43:06,842] [VirtualMachine.Infos] INFO:        | Current Snapshot: "Clean"
[2011-12-18 17:43:06,842] [VirtualMachine.Infos] INFO:        | MAC Address: 08:00:27:XX:XX:XX
[2011-12-18 17:43:06,842] [Core.Init] INFO: 1 virtual machine/s added to pool.
[2011-12-18 17:43:07,045] [Database.Init] INFO: Generated database "db/cuckoo.db" which didn't exist
```

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

2.3.2 Submit an analysis

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

- Use provided **submit.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 [options] filepath

Options:
  -h, --help            show this help message and exit
  -t TIMEOUT, --timeout=TIMEOUT
                        Specify analysis execution time limit
  -p PACKAGE, --package=PACKAGE
                        Specify custom analysis package name
  -r PRIORITY, --priority=PRIORITY
                        Specify an analysis priority expressed in integer
  -c CUSTOM, --custom=CUSTOM
                        Specify any custom value to be passed to postprocessing
  -d, --download        Specify if the target is an URL to be downloaded
  -u, --url             Specify if the target is an URL to be analyzed
  -m MACHINE, --machine=MACHINE
                        Specify a virtual machine you want to specifically use for this 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:

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

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

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

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

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

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

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

Example: submit an URL to be downloaded locally and analyzed:

```
$ python submit.py --download http://www.website.tld/file.exe
```

Example: submit an URL to be analyzed within Internet Explorer:

```
$ python submit.py --url http://maliciousurl.tld/exploit.php
```

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

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

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 *queue* which is defined as the following schema:

```
CREATE TABLE queue (
  id INTEGER PRIMARY KEY,
  md5 TEXT DEFAULT NULL,
  target TEXT NOT NULL,
  timeout INTEGER DEFAULT NULL,
  priority INTEGER DEFAULT 0,
  added_on DATE DEFAULT CURRENT_TIMESTAMP,
  completed_on DATE DEFAULT NULL,
  package TEXT DEFAULT NULL,
  lock INTEGER DEFAULT 0,
  status INTEGER DEFAULT 0,
  custom TEXT DEFAULT NULL,
  vm_id TEXT DEFAULT NULL
);
```

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.
- `target`: 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.
- `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.
- `package`: it's the name of the analysis package to be used, if non has been specified the field is set to NULL.
- `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 = completed successfully, 2 = failed).
- `custom`: it's a custom user-defined text that can be used for synchronization between submission and post-analysis processing.
- `vm_id`: it's the ID (as defined in cuckoo.conf) of a virtual machine the user specifically wants to use for the analysis.

Cuckoo Python Functions

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

```
    def add_task(self, target, md5 = None, timeout = None, package = None, priority = None, custom =
```

Following is a usage example:

```
#!/usr/bin/python
from cuckoo.core.db import CuckooDatabase

db = CuckooDatabase()
db.add_task("/path/to/binary")
```

2.3.3 Analysis Packages

The **analysis packages** are a key component in Cuckoo Sandbox.

They consist in structured Python scripts which are executed inside the virtual machine and that define how Cuckoo should conduct the analysis.

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

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

If none is specified, Cuckoo will try to detect the type of the file and choose the proper analysis package accordingly. If the file type is not supported and no package was specified, the analysis will be aborted and marked as failed in the database.

This functionality allows you not only to use existing analysis packages, but also create some of your own and customize your Cuckoo setup. Ths topic will be dealt in details in the *Analysis Packages* customization chapter.

Cuckoo provides some default analysis packages which include:

- `exe`: default analysis package used to analyze generic Windows executables.
- `dll`: used to analyze Dynamic Linked Libraries.
- `pdf`: used to analyze Adobe Reader while opening the given PDF file.
- `doc`: used to analyze Microsoft Office while opening documents.
- `php`: used to analyze PHP scripts.
- `ie`: used to analyze Internet Explorer while opening the given URL.
- `firefox`: used to analyze Mozilla Firefox while opening the given URL.
- `tracer`: used to trace assembly instructions performed by the given file.

2.3.4 Execution

When Cuckoo receives an analysis request, you'll see something like this:

```
[2011-12-18 18:20:16,242] [Core.Dispatcher] INFO: Acquired analysis task for target "/tmp/malwar
[2011-12-18 18:20:16,424] (Task #1) [Core.Analysis.Run] INFO: Acquired virtual machine "cuckoo1"
[2011-12-18 18:20:17,005] [VirtualMachine.Restore] INFO: Virtual machine "Cuckoo1" successfully
[2011-12-18 18:20:19,779] [VirtualMachine.Start] INFO: Virtual machine "Cuckoo1" starting in "gu
[2011-12-18 18:20:24,429] [VirtualMachine.Execute] INFO: Cuckoo analyzer running with PID 1732 c
[2011-12-18 18:20:54,871] [VirtualMachine.Execute] INFO: Cuckoo analyzer exited with code 0 on v
[2011-12-18 18:20:54,878] (Task #1) [Core.Analysis.SaveResults] INFO: Analysis results successfu
[2011-12-18 18:20:55,124] (Task #1) [Core.Analysis.Processing] INFO: Analysis results processor
[2011-12-18 18:20:56,307] [VirtualMachine.Stop] INFO: Virtual machine "Cuckoo1" powered off succ
```

```
10  [2011-12-18 18:20:56,308] (Task #1) [Core.Analysis.FreeVM] INFO: Virtual machine "cuckoo1" relea
11  [2011-12-18 18:20:56,309] (Task #1) [Core.Analysis.Run] INFO: Analyis completed.
```

Let's get through what happened.

At line **1** Cuckoo's tasks dispatcher acquired a new submission for the target */tmp/malware.exe*. At line **2** it acquired the free virtual machine *cuckoo1*. At line **3** Cuckoo restored the virtual machine to current snapshot and at line **4** it started it in graphical mode.

In the meanwhile it prepared all required files and configurations for the analysis.

At line **5** Cuckoo analyzer component started inside the virtualized Windows environment with process ID *1732*. At line **6**, after the 60 seconds of the specified timeout, the analyzer terminates its execution and exits. At line **7** the analysis results are stored to *analysis/1/* and this same path is specified to the processor script which is invoked at line **8** with process ID *8571*. At line **9** the virtual machine is successfully powered off and released at line **10**. At line **11** finally Cuckoo considers the analysis as completed.

At this point you should have complete analysis results into *analysis/1/* and, depending on the options you enabled in *reporting.conf* (*Configuration*), some automatically generated reports at *analysis/1/reports/*.

2.3.5 Analysis Results

Once an analysis is completed, several files are stored in a dedicated directory. Unless you configured differently, all of the analysis are saved into *analysis/* with a subdirectory named after the numerical ID assigned to the analysis into the database.

Following is an example of analysis results:

```
.
|-- analysis.conf
|-- analysis.log
|-- dump.pcap
|-- files
|   |-- dropped.tmp
|   `-- dropped.exe
|-- logs
|   |-- 1232.csv
|   |-- 1540.csv
|   `-- 1118.csv
|-- reports
|   |-- report.html
|   |-- report.json
|   `-- report.txt
|-- malware.exe
|-- shots
|   |-- shot_1.jpg
|   |-- shot_2.jpg
|   |-- shot_3.jpg
|   `-- shot_4.jpg
`-- trace
```

analysis.conf

analysis.conf is a configuration file automatically generated by Cuckoo to specify some parameters to the guest component (analyzer). It's generally not relevant for an end-users as it's exclusively used internally by Cuckoo.

analysis.log

analysis.log is a log file generated by Cuckoo analyzer and that keeps track of analysis execution and might report errors occurred during the analysis.

dump.pcap

dump.pcap is the trace dump containing all the network activity generated by the virtual machine during the malware execution.

files/

The *files/* directory contains all files created or deleted by the malware and that were successfully dumped by Cuckoo.

logs/

The *logs/* directory contains the raw CSV-like logs generated by the monitored processes and that contains the concrete behavioral tracing results.

reports/

The *reports/* directory contains the abstract analysis reports generated automatically by Cuckoo. The number and the format of such reports depends on the the configuration explained in the *Configuration* chapter.

shots/

The *shots/* directory contains the screenshots of Windows desktop taken during the analysis execution.

trace/

The *trace/* directory is only used in specific cases (for instance when using the *tracer* analysis package) and contains the assembly instruction traces of the monitored processes.

2.3.6 Web Interface

Cuckoo Sandbox comes with a very simple and handy web server which is used to navigate and view analysis reports.

Start the Web Server

The web server has following options:

```
Usage: web.py [options]

Options:
  -h, --help            show this help message and exit
  -t HOST, --host=HOST  Specify the host to bind the server on (default localhost)
  -p PORT, --port=PORT  Specify the port to bind the server on (default 8080)
```

In order to start it, just launch:

```
$ python web.py
```

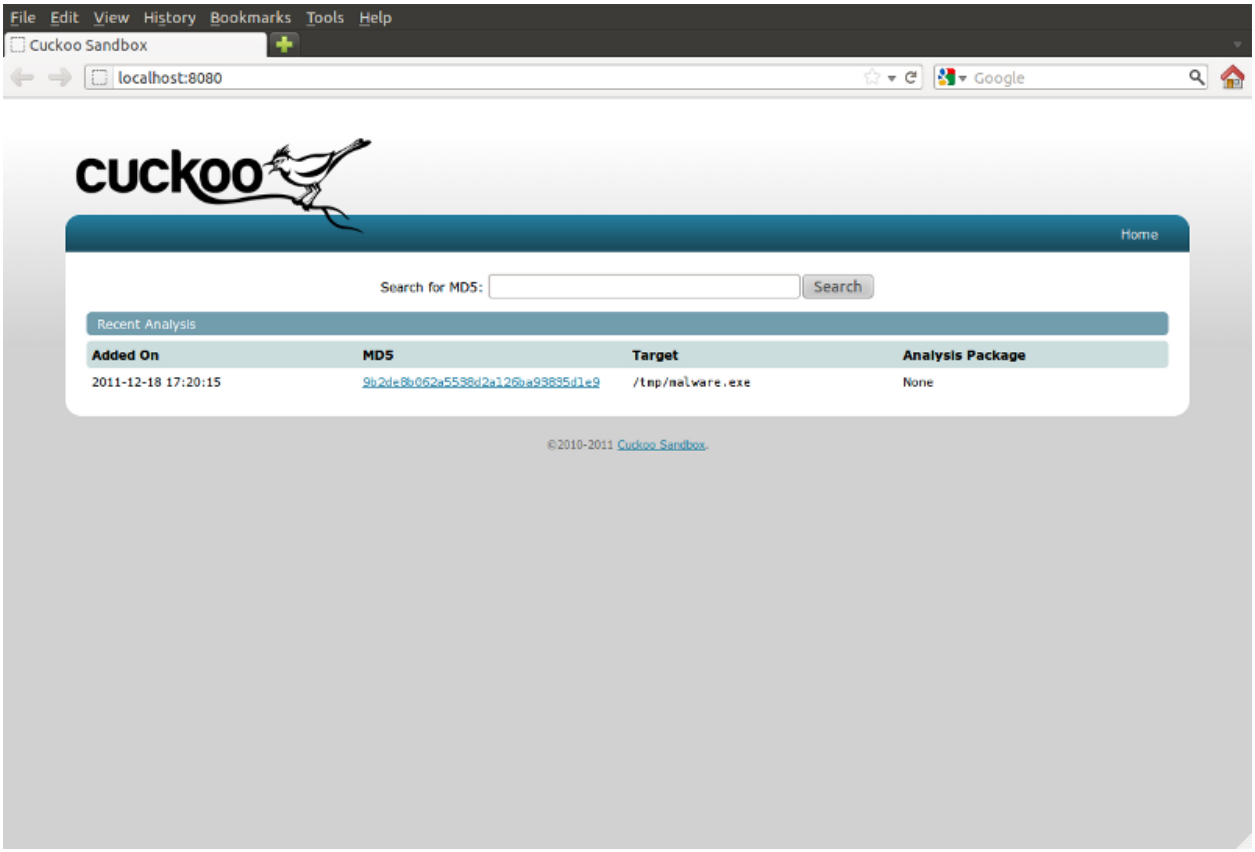
You should see displayed:

```
Starting web server on localhost:8080
```

You can change the host and the port on which to bind the web server by specifying them with the appropriate options.

Recent Analysis

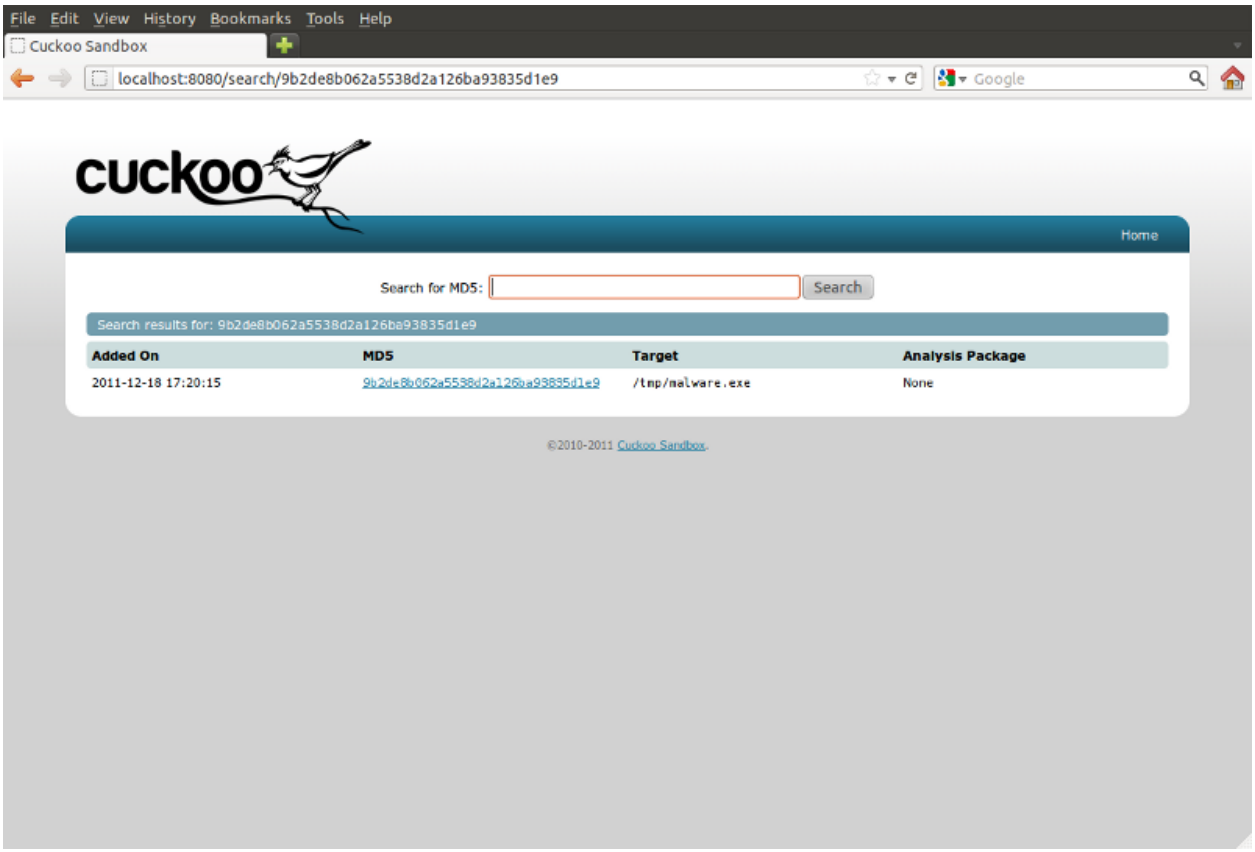
If you now open your browser and go to *http://localhost:8080/* you will see Cuckoo's index page, which includes a list of the 30 most recent analysis and a web form through which you can search for analysis by MD5 hash.



By filling the form and submitting it you'll be prompted with search results. By clicking on the MD5 of one of the analysis, you'll be prompted with the report for that specific analysis.

Search for MD5

After submitting a valid MD5 in the search form, you'll be prompted with all the analysis performed on the file matching that hash.



View Report

When requesting the report of a specific analysis, the web server will returns you the appropriate HTML report. Please notice that if in your Cuckoo Setup you disabled the generation of HTML reports (see *Configuration*), a 404 error will be returned.

2.4 Customization

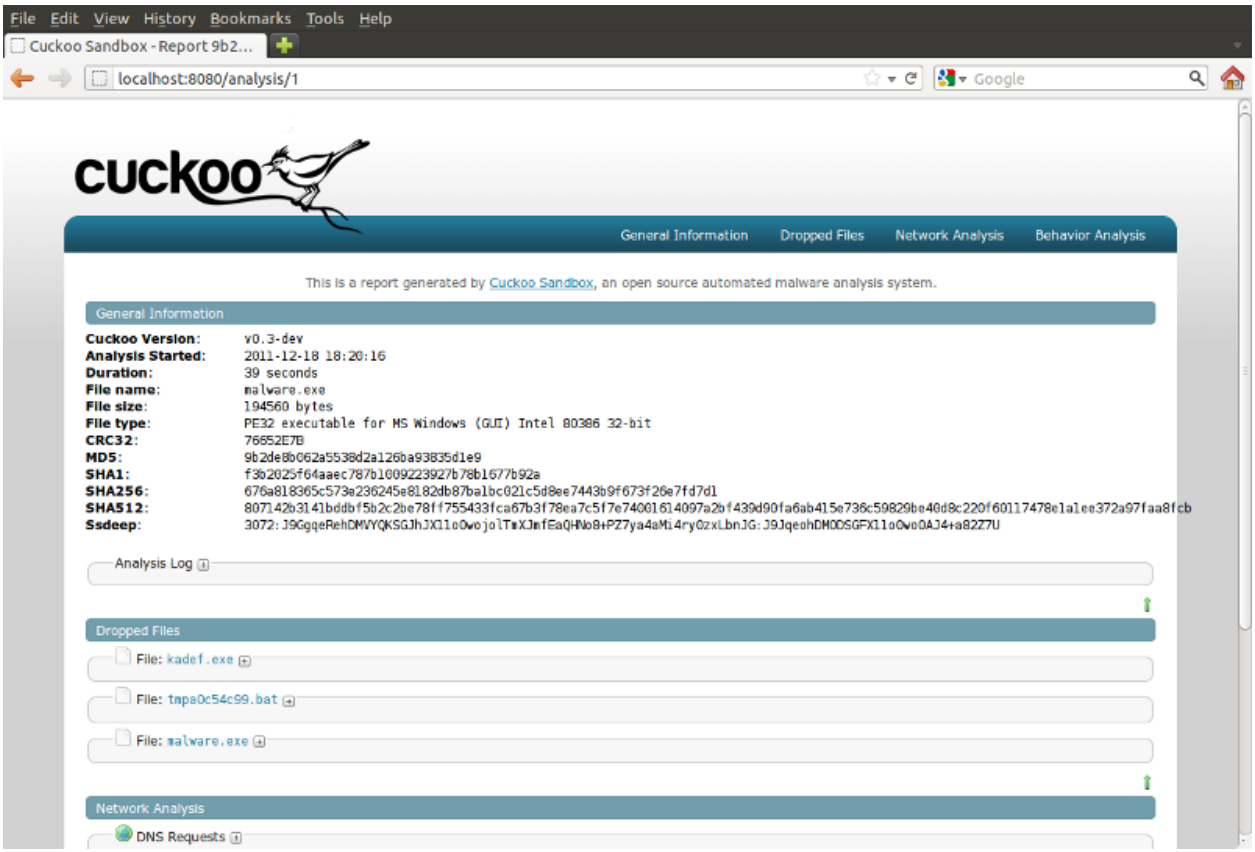
This chapter explains how to customize Cuckoo. Cuckoo is written in a modular architecture built to be as much customizable it can, to fit all user's needs.

2.4.1 Analysis Packages

As explained in *Analysis Packages*, analysis packages are structured Python scripts that allow you to customize the analysis procedure inside the virtualized Windows environment.

By default Cuckoo provides some default packages you can already use, but you are able to create and use some of your own.

Creating new packages is really easy and just requires minimal knowledge of the Python language.



Getting started

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

```
1  import os
2  import sys
3
4  sys.path.append("\\\\VBOXSVR\\setup\\lib\\")
5
6  from cuckoo.execute import cuckoo_execute
7  from cuckoo.monitor import cuckoo_monitor
8
9  # The package main function "cuckoo_run" should follow a fixed structure in
10 # order for Cuckoo to correctly handle it and its results.
11 def cuckoo_run(target_path):
12     # Every analysis package can retrieve a list of multiple process IDs it
13     # might have generated. All processes added to this list will be added to
14     # the monitored list, and Cuckoo will wait for all of the to complete their
15     # execution before ending the analysis.
16     pids = []
17
18     # The following functions are used to launch a process with the simplified
19     # "cuckoo_execute" function. This function takes as arguments (in specific
20     # order):
21     # - a path to the executable to launch
22     # - arguments to be passed on execution
23     # - a boolean value to specify if the process have to be created in
```

```
24      #   suspended mode or not (it's recommended to set it to True if the
25      #   process is supposed to be injected and monitored).
26      suspended = True
27      (pid, h_thread) = cuckoo_execute(target_path, None, suspended)
28
29      # The function "cuckoo_monitor" invoke the DLL injection and resume the
30      # process if it was suspended. It needs the process id and the main thread
31      # handle returned by "cuckoo_execute" and the same boolean value to tell it
32      # if it needs to resume the process.
33      cuckoo_monitor(pid, h_thread, suspended)
34
35      # Append all the process IDs you want to the list, and return the list.
36      pids.append(pid)
37      return pids
38
39  def cuckoo_check():
40      return True
41
42  def cuckoo_finish():
43      return True
```

Let's walk through the given code.

At line **1** and **2** we import the `os` and `sys` Python modules. At line **4** we append "*\\VBOXSVR\setup\lib*" to Python's modules paths list: this will allow us to invoke Cuckoo's modules directly from the shared folder.

Then we can see that three functions are defined:

- *cuckoo_run()*
- *cuckoo_check()*
- *cuckoo_finish()*

In the given example the package just executes the binary located at `target_path` in suspended mode and instructs Cuckoo to inject the process and start monitoring it.

A slightly more complex example is the PDF analysis package (located at *shares/setup/packages/pdf.py*):

```
1   import os
2   import sys
3
4   sys.path.append("\\\\VBOXSVR\\setup\\lib\\")
5
6   from cuckoo.execute import cuckoo_execute
7   from cuckoo.monitor import cuckoo_monitor
8
9   def cuckoo_run(target_path):
10      pids = []
11
12      # Customize this Path with the correct one on your Windows setup.
13      adobe_reader = "C:\\Program Files\\Adobe\\Reader 9.0\\Reader\\AcroRd32.exe"
14
15      suspended = True
16      (pid, h_thread) = cuckoo_execute(adobe_reader, "\"%s\"" % target_path, suspended)
17      cuckoo_monitor(pid, h_thread, suspended)
18
19      pids.append(pid)
20      return pids
21
22  def cuckoo_check():
```

```
23      return True
24
25  def cuckoo_finish():
26      return True
```

In this example we have the same structure, with the only difference being that instead of executing the file at *target_path*, it executes Adobe Reader with *target_path* as argument. In this way it basically instructs Cuckoo to monitor what Adobe Reader is doing while opening the given PDF file. As you understand, this opens a large spectrum of possibilities on what Cuckoo can be used for.

cuckoo_run()

This function is the starting point of the analysis. In this block you should define every operation that should performed as initialization of the analysis.

This could include the execution of processes, creation of files, injection of processes and whatever you might need to perform.

It should return a list of PIDs that will be used by Cuckoo to monitor their process status: when all monitored processes complete their execution, Cuckoo will terminate the analysis and exit earlier. If none are returned, Cuckoo will assume that there is no process monitored and will just run for the amount of seconds specified by the analysis timeout.

cuckoo_check()

This function is performed regularly every second during the analysis. It can be used to perform custom checks or any other operation needed.

If the `cuckoo_check()` function returns *False*, Cuckoo will assume that the package matched a conditional check and it will terminate the analysis earlier.

cuckoo_finish()

This function is executed when the analysis is completed. It can be used for any post-analysis purpose such as copying files or any other operation you might need to perform before the virtual machine is shut down.

Cuckoo Modules

As you noticed in the packages examples, Cuckoo provides some custom functions that facilitates some complex Windows actions.

These functions are defined in some Python modules that Cuckoo provide by default. You can use any of these modules in your analysis packages.

Following is a list of available modules and the contained functions.

cuckoo.checkprocess

- **Function** `check_process()`:

  Prototype:

  ```
  def check_process(pid)
  ```

Description: check if the specified process is still active and running.

Parameter `pid`: process ID of the process to check.

Return: True if the process is active, otherwise False.

Usage Example:

```
from cuckoo.checkprocess import check_process

if check_process(pid):
    print "Process is active!"
else:
    print "Process is NOT active!"
```

cuckoo.execute

- **Function** `cuckoo_execute()`:

  Prototype:

  ```
  def cuckoo_execute(target_path, args = None, suspend = False)
  ```

  Description: creates a process from the specified file.

  Parameter `target_path`: path to the file to execute.

  Parameter `args`: arguments to pass to the process.

  Parameter `suspend`: set to True if should be created in suspended mode, otherwise set to False.

  Return: returns a list with PID and thread handle.

  Usage Example:

  ```
  from cuckoo.execute import cuckoo_execute

  (pid, h_thread) = cuckoo_execute("C:\\binary.exe")
  ```

cuckoo.inject

- **Function** `cuckoo_inject()`:

  Prototype:

  ```
  def cuckoo_inject(pid, dll_path)
  ```

  Description: injects the process with the specified PID with the DLL located at *dll_path*.

  Parameter `pid`: ID of the process to inject.

  Parameter `dll_path`: path to the DLL to be injected.

  Return: returns True if injection succeeded, otherwise False.

  Usage Example:

  ```
  from cuckoo.inject import cuckoo_inject

  if cuckoo_inject(pid, "C:\\library.dll"):
      print "Process injected successfully!"
  ```

```
else:
    print "Injection failed!"
```

cuckoo.monitor

- **Function** `cuckoo_resumethread()`:

  Prototype:

  ```
  def cuckoo_resumethread(h_thread = -1)
  ```

  Description: resumes a thread from suspended mode.

  Parameter `h_thread`: handle to the thread to be resumed (as returned by `cuckoo_execute()`.

  Return: returns True if resume succeeded, otherwise False.

  Usage Example:

  ```
  from cuckoo.monitor import cuckoo_resumethread

  if cuckoo_resumethread(h_thread):
      print "Process resumed!"
  else:
      print "Process resume failed!"
  ```

- **Function** `cuckoo_monitor()`:

  Prototype:

  ```
  def cuckoo_monitor(pid = -1, h_thread = -1, suspended = False, dll_path = None)
  ```

  Description: instructs Cuckoo to inject and monitor the specified process.

  Parameter `pid`: ID of the process to monitor.

  Parameter `h_thread`: handle to the main thread of the process to monitor (as returned by `cuckoo_execute()`).

  Parameter `suspended`: set to True if the process was created suspended and has to be resumed, otherwise False.

  Parameter `dll_path` (optional): path to the DLL to inject into the process. If none is specified it will use the default one.

  Return: returns True if monitor succeeded, otherwise False.

  Usage Example:

  ```
  from cuckoo.monitor import cuckoo_monitor

  if cuckoo_monitor(pid, h_thread, True):
      print "Process monitoring started successfully!"
  else:
      print "Process monitoring failed!"
  ```

cuckoo.trace

- **Function** `cuckoo_trace()`:

Prototype:

```
def cuckoo_trace(pid = -1)
```

Description: instructs Cuckoo to trace assembly instructions from the specified process.

Parameter `pid`: ID of the process to monitor.

Return: returns True if tracing was successful, otherwise False.

Usage Example:

```
from cuckoo.trace import cuckoo_trace

if cuckoo_trace(pid):
    print "Process traced successfully!"
else:
    print "Process trace failed!"
```

2.4.2 Processing of results

As exaplained in the *Configuration* chapter, once an analysis is completed, Cuckoo invokes a script which can be used to access and manipulate the results produced. It's conceived to be customized by the user to make it do whatever he prefers.

Such script (called "*processor*") is invoked concurrently to Cuckoo, making it completely independent from the sandbox execution, and takes the path to the analysis results as argument.

The default processor looks like following:

```
import sys

from cuckoo.processing.data import CuckooDict
from cuckoo.reporting.reporter import ReportProcessor

def main(analysis_path):
    # Generate reports out of abstracted analysis results.
    ReportProcessor().report(CuckooDict(analysis_path).process())

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print "Not enough args."
        sys.exit(-1)

    main(sys.argv[1])
```

What it does is obtain a dictionary out of the raw results and invoke the generation of the enabled reports as explained in *Configuration*.

In order to simplify some of the processing tasks you might need to perform, Cuckoo provide some ready-to-use functions and classes which are generally located in "*cuckoo/processing/*".

Retrieving details on a file

The first thing you might be interested in, is retrieving some details on the binary you just analyzed. For this purpose there's a dedicated class called `File` provided by the module `cuckoo.processing.file`. It takes the path to a file as argument and invoking `process()` retrieves a dictionary containing some static details. You can actually use this clan on any file you want, perhaps also on dropped files.

Following is an example usage and output:

```
>>> import pprint
>>> from cuckoo.processing.file import File
>>> details = File("analysis/1/malware.exe").process()
>>> pprint.pprint(details)
{'crc32': '76652E7B',
 'md5': '9b2de8b062a5538d2a126ba93835d1e9',
 'name': 'malware.exe',
 'sha1': 'f3b2025f64aaec787b1009223927b78b1677b92a',
 'sha256': '676a818365c573e236245e8182db87ba1bc021c5d8ee7443b9f673f26e7fd7d1',
 'sha512': '807142b3141bddbf5b2c2be78ff755433fca67b3f78ea7c5f7e74001614097a2bf439d90fa6ab415e736
 'size': 194560,
 'ssdeep': '3072:J9GgqeRehDMVYQKSGJhJX11o0wojolTmXJmfEaQHNo8+PZ7ya4aMi4ry0zxLbnJG:J9JqeohDMODSGF
 'type': 'PE32 executable for MS Windows (GUI) Intel 80386 32-bit'}
```

Processing behavioral analysis results

As you read in *Analysis Results*, Cuckoo generates some csv-like raw logs for every process it monitored. These logs contains all the win32 API calls that Cuckoo was able to intercept while tracking the processes. In order to make the information contained there more accessible, you can use the `Analysis` class provided by the module `cuckoo.processing.analysis`.

This class takes the path to the logs files as argument and, by calling its function `process()`, it will return a dictionary containing the behavioral results in a structured format.

Following is an example usage and output:

```
>>> import pprint
>>> from cuckoo.processing.analysis import Analysis
>>> results = Analysis("analysis/1/logs/").process()
>>> pprint.pprint(results)
[{'calls': [{'api': 'LoadLibraryA',
             'arguments': [{'name': 'lpFileName', 'value': 'KERNEL32.DLL'}],
             'repeated': 0,
             'return': '0x7c800000',
             'status': 'SUCCESS',
             'timestamp': '20111219100536.679'},

            [...]

            {'api': 'VirtualAllocEx',
             'arguments': [{'name': 'th32ProcessID', 'value': '764'},
                           {'name': 'szExeFile', 'value': 'binary.exe'},
                           {'name': 'lpAddress', 'value': '0x00000000'},
                           {'name': 'dwSize', 'value': '4826'},
                           {'name': 'flAllocationType',
                            'value': '0x00003000'},
                           {'name': 'flProtect', 'value': '0x00000040'}],
             'repeated': 0,
             'return': '0x00150000',
             'status': 'SUCCESS',
             'timestamp': '20111219100536.679'},
            {'api': 'CreateFileW',
             'arguments': [{'name': 'lpFileName',
                            'value': 'C:\\WINDOWS\\system32\\svchost.exe'},
                           {'name': 'dwDesiredAccess',
                            'value': 'GENERIC_READ'}],
```

```
                'repeated': 1,
                'return': '0x000000b4',
                'status': 'SUCCESS',
                'timestamp': '20111219100546.734'},
               {'api': 'CreateProcessA',
                'arguments': [{'name': 'lpApplicationName',
                               'value': '(null)'},
                              {'name': 'lpCommandLine',
                               'value': 'svchost.exe'}],
                'repeated': 0,
                'return': '1548',
                'status': 'SUCCESS',
                'timestamp': '20111219100546.734'},
               {'api': 'VirtualAllocEx',
                'arguments': [{'name': 'th32ProcessID', 'value': '1548'},
                              {'name': 'szExeFile', 'value': 'svchost.exe'},
                              {'name': 'lpAddress', 'value': '0x00000000'},
                              {'name': 'dwSize', 'value': '0'},
                              {'name': 'flAllocationType',
                               'value': '0x00003000'},
                              {'name': 'flProtect', 'value': '0x00000040'}],
                'repeated': 0,
                'return': '',
                'status': 'FAILURE',
                'timestamp': '20111219100546.734'},
               {'api': 'ExitProcess',
                'arguments': [{'name': 'uExitCode', 'value': '0x00000000'}],
                'repeated': 0,
                'return': '',
                'status': '',
                'timestamp': '20111219100546.744'}],
  'first_seen': '20111219100536.679',
  'process_id': '764',
  'process_name': 'binary.exe'}]
```

Using the normalized data generated by `Analysis` class, you can even generate a tree with the `ProcessTree` class which orders the monitored processes recursively.

Following is an example usage and output:

```
>>> import pprint
>>> from cuckoo.processing.analysis import Analysis, ProcessTree
>>> results = Analysis("analysis/2/logs/").process()
>>> tree = ProcessTree(results).process()
>>> pprint.pprint(tree)
[{'children': [{'children': [], 'name': 'kadef.exe', 'pid': 788},
               {'children': [], 'name': 'cmd.exe', 'pid': 1764}],
  'name': 'malware.exe',
  'pid': 1488}]
```

Processing network traffic

In the exact same way as you can process behavioral results, you can also process network traffic from the PCAP file using the `Pcap` class available from `cuckoo.processing.pcap`.

At current stage it retrieves a dictionary with all the information on DNS and HTTP requests as well as all UDP and TCP packets.

Following is an example usage and output:

```
>>> import pprint
>>> from cuckoo.processing.pcap import Pcap
>>> network = Pcap("analysis/3/dump.pcap").process()
>>> pprint.pprint(network)
{'dns': [{'hostname': 'www.google.com', 'ip': '74.125.127.104'}],
 'http': [{'body': '',
           'data': 'GET / HTTP/1.1\r\nHost: www.google.com\r\nUser-Agent: Mozilla/5.0 (Windows N
           'host': 'www.google.com',
           'method': 'GET',
           'path': '/',
           'port': 80,
           'uri': 'http://www.google.com/',
           'user-agent': 'Mozilla/5.0 (Windows NT 5.1; rv:6.0.2) Gecko/20100101 Firefox/6.0.2',
           'version': '1.1'}],
 'tcp': [{'dport': 80,
          'dst': '74.125.127.104',
          'sport': 1214,
          'src': '10.0.2.15'}],
 'udp': [{'dport': 67,
          'dst': '255.255.255.255',
          'sport': 68,
          'src': '0.0.0.0'}]}
```

Putting all together

If you don't want to bother invoking all the necessary classes but just want a comprehensive (and huge) dictionary containing everything you need, you can simply use the `CuckooDict` class provided by the module `cuckoo.processing.data`, just like the default package do.

Following is an example usage and output:

```
>>> import pprint
>>> from cuckoo.processing.data import CuckooDict
>>> analysis = CuckooDict("analysis/2/").process()
>>> pprint.pprint(analysis)
{'behavior': {'processes': [<results provided by class Analysis>],
              'processtree': [<results provided by class ProcessTree>]},
 'debug': {'log': '<content of analysis.log file>'},
 'dropped': [<results provided by class File on all dropped files>],
 'file': {<results provided by class File on the analyzed file>},
 'info': {'duration': '38846 seconds',
          'started': '2011-12-19 11:05:06',
          'version': 'v0.3'},
 'network': {<results provided by class Pcap>},
 'static': {}}
```

The output has been stripped out of results.

2.4.3 Reporting Results

The *processor* script is responsible for taking analysis results and elaborate them, as explained in the *Processing of results* chapter.

Since version 0.3, Cuckoo Sandbox provides also a reporting engine that can be used to generate consumable reports (as done by the default *processor* script): it takes the analysis results as input and stores the produced reports in the

dedicated folder as explained in the *Analysis Results* chapter.

The reporting engine, called `ReportProcessor`, is designed to load all reporting modules specified in configuration file *reporting.conf* (see *Configuration* chapter) and execute them.

A reporting module is a simple Python script which aggregates, normalizes and correlates analysis data in order to generate a report out of it. Cuckoo comes with several built-in reporting modules described below, but writing your own modules is incredibly simple.

Built-in Reports

Report TXT

This module generates a human-readable report in plain text format.

Report HTML

This module generates a human-readable report in HTML format. These reports are also served by the built-in web server as explained in *Web Interface*.

JSON Dump

This module dumps all Cuckoo's analysis results in JSON format. This is useful when you need to export Cuckoo's data to other tools or services.

Writing your own reporting module

As said, reporting tasks are handled by the `ReportProcessor` class: it loads all reporting modules from *cuckoo/reporting/tasks* folder, checks if they are enabled in the configuration file and then execute them.

If you want to write your own reporting module you have to:

- Create a Python file inside the reporting module folder (*cuckoo/reporting/tasks*), e.g. *foo.py*.
- Append an option inside the reporting configuration file (*reporting.conf*) with the lowercase name of the file and enable it, like following:

  ```
  foo = on
  ```

- Inside your Python script you have to implement the `BaseObserver` interface in a class named "`Report`". When new analysis results are available, Cuckoo calls your `update()` method passing the analysis results as a parameter.

A sample custom reporting module would look like following:

```
1  from cuckoo.reporting.observers import BaseObserver
2
3  class Report(BaseObserver):
4      def __init__(self):
5          # Put here your initialization or leave a pass.
6          pass
7
8      def update(self, results):
9          # Here you get analysis results as parameter.
```

```
10          # Now do your stuff.
11          print "My report!"
```

Whatever operation you might want to run, remember to place it inside the `update()` method or invoke it from there, so that Cuckoo will be able to execute it when needed.

2.5 Final Remarks

2.5.1 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 through IRC on the official #cuckoobox channel.

2.5.2 Contribute

We often get contacted by people willing to contribute. Most of the times they consist of unexperiences users. Other times they never end what they started or promised.

There are a lot of ways to contribute to an open source project, not only developing it but also donating or sponsoring hardware for testing, reporting bug or suggesting new features to improve software, donating money to fund costs like hosting, or simply publishing Cuckoo around the web. Any kind of help is really appreciated and help Cuckoo project.

Cuckoo is a simple but still somewhat complex software, therefore we are very careful on who and how can contribute actively developing the software. This doesn't mean that we don't accept any, but before coming to us please make sure that you are seriously committed into contributing and that you have the required knowledge and experience using Cuckoo and understanding how it works.

The best way to get involved is to *Join the discussion* and start writing code by your own: patches, new modules, new analysis packages. Reflect on what you believe Cuckoo lacks on and do it.

2.5.3 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 on being developed and maintained, 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, anything) you can contact us at `donations at cuckoobox dot org`.

2.5.4 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 :).

Developers

Name	Role	Contact
Claudio "*nex*" Guarnieri	Lead Developer	`nex at cuckoobox dot org`
Dario Fernandes	Developer	`dario at cuckoobox dot org`
Alessandro "*jekil*" Tanasi	Developer	`alessandro at tanasi dot it`

Contributors

- Ryan Sommers (contributed with the concept of the DLL analysis package)

Acknowledgements

- Felix Leder (for mentoring and believing in the project in the first place)
- Kjell Christian Nilsen (for providing valuable feedback and several feature suggestions)
- Mark Schloesser and Angelo dell'Aera (for helping on Python matters :P)
- Georg Wicherski (for supporting the project and giving it a home at first place)
- Carsten Willems (for the precious suggestions provided)
- The Honeynet Project (for supporting the project)
- The Shadowserver Foundation (for supporting the project)
- Everyone using Cuckoo (for giving a sense to all of this)