Cuckoo Sandbox

- open source automated malware analysis

This document is submitted as the white paper for the Cuckoo Sandbox workshop at Blackhat US 2013.

It is best viewed at its online location.

Table of Contents

Introduction

The current landscape of automated dynamic analysis of malicious files includes a few vendors with commercial tools and several open source projects and independent efforts.

Cuckoo Sandbox is one of the open-source projects that has gained popularity in the recent years. It is widely used by academic and independent researchers as well as small to large companies and enterprises. The last version counted over 10000 downloads over the course of several months. Even in IT environments with dedicated CERTs and large budgets for commercial tools the project is very popular due to active community contributions and flexible customization possibilities.

Specifically easy-to-use APIs as well as user interfaces are needed for today's security analysts as the whole community can not scale with the threats observed every day. Automation and customized tool chains make it possible to become more efficient every day through an iterative process.

This observation defines the main design goal of Cuckoo. We try to achieve flexibility and customization possibilities through modular design and simplicity. Every stage of the analysis process can be modified and integrated into existing infrastructure.

The success of Cuckoo Sandbox as an open source project shows with over 10000 downloads of the latest release. Even though there is still some effort required for setting up a running system, the extensive documentation guides new users along.

http://docs.cuckoosandbox.org/en/latest/

This document will give an overview of Cuckoo Sandbox and describe the individual components and technologies involved. For any further guidance the documentation, community portal and mailing lists can be considered.

Malware Analysis with a Sandbox

Generally Sandbox systems are used to both process a large number of malware samples and to gain starting points for an analysis case. Before using debugging techniques and static reverse engineering it can be useful to collect some corner pieces from a sandbox report.

When used to process a high amount of malware it is important that the analysis can be completely automated. This means there should be simple ways to submit files to the sandbox, define options and features for the analysis run and extract the results one is interested in afterwards.

In other environments like smaller IT teams it is important that a sandbox is easy to use and provides a high-level summary of the malicious activity conducted during the analysis run. This makes malware analysis very accessible and allows everyone to conduct the first stages in an

investigation. In case it yields interesting results, the case can then be handed off to an analysis team or external partner.

With low amounts of human resources the automation and integration capabilities of a sandbox solution are very important. If a single person is responsible for handling malware occurrences, then the processes need to be largely automated to suit the user's needs and allow for efficient workflows. Thus flexible design and customization features accessible to anyone are a very appreciated feature in any tool and especially security software.

Open Source community efforts

We believe in open source software and leveraging the creativity of the community to build better and more useful software. Thus Cuckoo Sandbox has always focused on allowing people to customize it and build their own modules for any stage of the analysis process. This makes it possible to write custom config extraction plugins, write results to already existing custom database setups and analyze new file types if necessary. Also the detection of specific malicious behavior through so-called signatures is largely a community effort that anyone can contribute to.

Cuckoo has a special community repository for collecting any modules that are contributed and thus anyone setting up his instance can cherry-pick a combination of signatures, reporting modules, etc that suits their needs.

https://github.com/cuckoobox/community

We believe that there's high value and potential in the malware research community to be more transparent and cooperative and the community repository intends to be an initiative to support that.

Additionally we provide malwr.com - a free ready-to-use version of Cuckoo running on sponsored hardware on the internet. This way any user who does not need an offline customized setup can directly upload malware samples without disclosing them to third parties.

This instance has a special web interface with user logins to add commenting and tagging of certain sample reports.

https://malwr.com/

The interface used on this setup will be released as a part of Cuckoo and thus open source software as well. This is currently still work in progress and will happen shortly before or during the presentation at Blackhat USA 2013.

Related work - other projects and commercial solutions

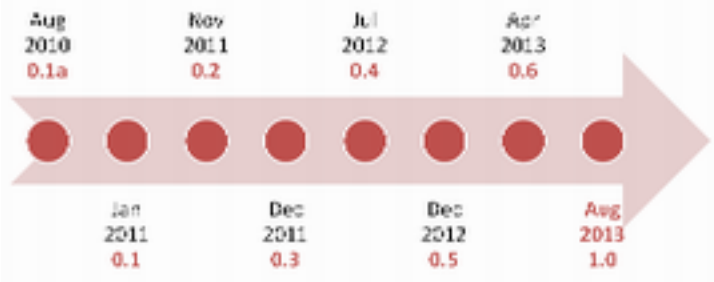
There are a few products available that largely automate malicious document analysis and throw a fair amount of developer resources at any integration requests and feature implementation. Commercial tools have have a variety of features and approaches that might be hard to obtain in similar quality for open source projects with smaller teams. On the other hand the cost of a commercial tool often is so high that it immediately rules it out as an option for small to medium sized businesses.

One major aspect in the comparison are the open internals of Cuckoo Sandbox and flexible modular design. It sometimes means that even sophisticated APIs of commercial tools might be more difficult to use than plain patching of the actual source code.

For this exact reason we have received feedback of large companies that invest time and money into commercial tools - but at the same time still rely on Cuckoo Sandbox for parts of their analysis as it is easy to integrate and customize.

Cuckoo Sandbox Implementation Overview

Cuckoo Sandbox was started in 2010 as a Google Summer of Code project. Throughout the years it has undergone several re-designs and major improvements. By now the software consists of around 50000 lines of code written in Python and C by 4 core developers and roughly 25 contributors over time.



An analysis process in Cuckoo Sandbox starts after submitting a malicious file to it. The submitted information can contain both the location of the sample, as well as the designated analysis package and options such as the timeout for running the file. The data is stored in an internal database and as soon as a virtual machine is available will be scheduled for processing.

Before uploading the analyzer to the virtual machine together with the malicious file, the VM is restored to a snapshot to start from a clean state. Inside the machine the only necessary part is called the “Cuckoo Agent” - which fundamentally is a XMLRPC server that receives a compressed package and executes the analyzer contained in it.

These phases of malware analysis can be customized almost completely. Machinery modules define how the virtualization solution is managed and VMs are started, restored and stopped.

There already exists support for the libvirt library – but other software could be added by writing a new module.

Analysis packages define how a certain type of file is executed after being uploaded into the VM. A PE (.exe) file can be started directly while a Word document needs to be loaded into an office tool. The reason for Cuckoo supporting URL analysis is merely the existence of a “Internet Explorer” package start starts IE on the specified URL and instruments the created process.

Another module type in Cuckoo is the *Auxiliary* class. These modules are run concurrently to the analysis process and do not directly influence the instrumentation or logging. An example use case for this is the Human module. It emulates some user behavior like mouse movement and mouse clicks in order to defeat anti-analysis tricks based on the fact that a sandbox VM is not operated with a mouse.

```
68 class Human(Auxiliary, Thread):
69     """Human after all"""
70
71     def __init__(self):
72         Thread.__init__(self)
73         self.do_run = True
74
75     def stop(self):
76         self.do_run = False
77
78     def run(self):
79         while self.do_run:
80             move_mouse()
81             click_mouse()
82             USER32.EnumWindows(EnumWindowsProc(foreach_window), 0)
83             KERNEL32.Sleep(1000)
```

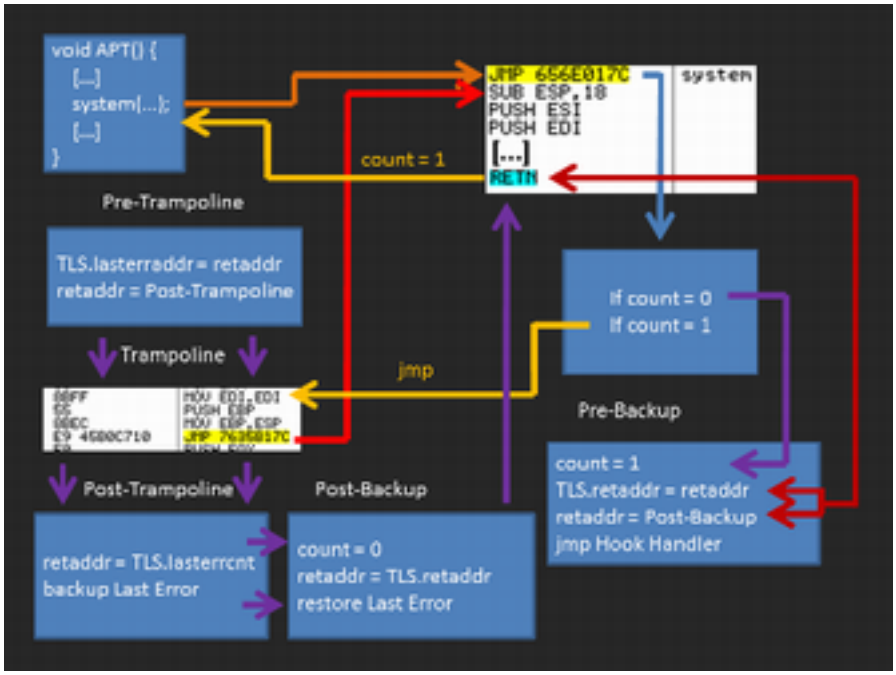
The actual instrumentation of the running processes is currently done by injecting a DLL that hooks certain Windows API functions and logs their parameters when called. This DLL is described in the next section.

The analyzer core: Cuckoomon

The actual instrumentation component within Cuckoo Sandbox is a dynamic library that logs actions conducted by the process to the main sandbox running on the host machine. This is achieved via classic userspace inline hooking, albeit with a few interesting aspects in the custom hooking engine. Cuckoomon is able to randomize the instructions written to the target function preamble in order to make detection more difficult.

Every hooked function is diverted into a series of trampolines that check the current logging state. For example we don't want to create duplicate log entries in case a hooked function executes another hooked function. We already capture the first call's parameters and thus don't need to log the second call at all.

Cuckoomon also diverts the control flow on function exit in order to do some cleanup work such as setting the correct “LastError” value. To achieve this we keep track of a stack of return addresses to jump back to – similar to the operating system itself. The following graphic shows an outline of the control flow for hooked functions. The exact explanation is out of the scope of this paper and thus will be skipped at this point.



The DLL logs the results to the sandbox process on the host. A dedicated “result-server” talks to Cuckoomon using a custom “netlog” protocol that was introduced in version 0.5. This real-time

logging makes sure that no results are lost even in the case of a VM crash. Also we don't have to collect results in a time-consuming way but can forcefully stop the VM after the timeouts are hit or all instrumented processes have exited.

Post-processing analysis results

After the execution of the target file has finished, the processing modules kick in. These gather information from the collected results – such as monitored processes and their properties, behavior summaries and the actual logged API calls.

The output of this step is then passed into all available signature modules to detect suspicious or malicious behavior. The matched signatures can aid the analyst in deciding whether the file is actually malicious and also quickly see several capabilities like browser information stealing or anti-debugging activity.

The following image shows a signature class that checks if certain mutexes were created during analysis that are specific for the SpyEye malware family. A more complex example of this can be found in the appendix.

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

class SpyEyeMutexes(Signature):
    name = "banker_spyeye_mutexes"
    description = "Creates known SpyEye mutexes"
    severity = 3
    categories = ["banker"]
    families = ["spyeye"]
    authors = ["nex"]
    minimum = "0.5"

    def run(self):
        indicators = [
            "2XeRY3a_PtW.*",
            "SPYNET",
            "__CLEANSWEEP__",
            "__CLEANSWEEP_UNINSTALL__",
            "__CLEANSWEEP_RELOADCFG__"
        ]

        for indicator in indicators:
            if self.check_mutex(pattern=indicator, regex=True):
                return True

        return False
```

Finally the reporting modules in Cuckoo take care of storing all collected information into databases and generating different types of output files – such as a rendered HTML file and a raw JSON report.

```
import os
import json
import codecs

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

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

    def run(self, results):
        """Writes report.
        @param results: Cuckoo results dict.
        @raise CuckooReportError: if fails to write report.
        """
        try:
            report = codecs.open(os.path.join(self.reports_path, "report.json"), "w", "utf-8")
            json.dump(results, report, sort_keys=False, indent=4)
            report.close()
        except (UnicodeError, TypeError, IOError) as e:
            raise CuckooReportError("Failed to generate JSON report: %s" % e)
```

Other users have successfully implemented reporting modules for storing into their existing cloud storage or other database systems. Another use case could be sending notifications or uploading partial results depending on certain signatures being matched during processing (see appendix - PoisonIvy Upload Module).

A best-practice setup

The main setup step needed before being able to analyze files is installing and preparing a target virtual machine. On internal instances we run both Windows XP and Windows 7 VMs. Once installed they are prepared with Python and the Cuckoo agent, as well as any software the user deems necessary.

If one wishes to also run PDF files through the sandbox, the PDF reader (Adobe) needs to be installed. The same goes for Microsoft Office and any other tools. Once this step is done, the system is ready to run and analyze files. However there are recommended additional aspects to consider like internet access for the VM and submission possibilities.

If the VM is created with full internet connectivity, then any malicious activity against remote hosts actually happens during analysis runs. Especially sending spam mails and denial of service attacks should be limited when running the malware through the sandbox. Thus we recommend at least firewalling the VM to limit connection attempts per time and redirect certain protocols such as SMTP to a fake service instead of going out to the public internet.

Another approach would be to completely separate the analysis target from the internet and employ a combination of fake DNS responses and fake services to analyze network behavior while at the same time containing the sample in a local only setup. We recommend this setup for processing large amounts of malware and only allowing certain samples to access online resources based on individual decisions or when interesting behavior is observed.

These environment choices can be made according to the user's needs and can vary from one deployment to the next. An interesting combination could also be to let samples contact online resources by default, but provide fake ones in case those can not be reached. This allows for a high coverage of possible behavior in the sample.

The necessary scripts and tools for putting together the environment are provided both by Cuckoo developers and community contributors. Fake services can be deployed by software such as “Inetsim” or “honeyd”. Links to these projects as well as a few small related scripts are linked in the appendix for reference.

Summary - Outlook

The Cuckoo Sandbox project intends to make dynamic analysis of malicious documents available and easy to use for the whole community. We see ourselves not as competition of commercial tools but rather as an orthogonal solution that fits into heterogenous environments that need flexible integration and largely rely on community driven improvements. Especially focused on people with lower budgets we hope to benefit the community as a whole.

The core team and several contributors are trying hard to improve the software even more and further increase flexibility, third party tool integration and stability. A recent blog post talked about the completion of the Magnifiscent7 sponsorship program for Cuckoo and some of the

new features that are introduced or upcoming. Rapid7 has enabled several improvements within Cuckoo by sponsoring and allowing core developers to devote some work time on the project.

We hope to improve and grow the project in the future - please visit the website and subscribe to the mailing list in order to help in this effort and provide feedback or contribute code.

Cheers,

the Cuckoo developers

Appendix

This appendix contains any screenshots or code listings added for reference.

PoisonIvy Upload Module

A reporting module that contacts a certain web service when poisonivy related analysis results are found.

```python
import requests

from lib.cuckoo.common.abstracts import Report

class PoisonReport(Report):

    def run(self, results):
        if not "poisonivy" in results or not results["poisonivy"]["domain"]:
            # No PoisonIvy detected.
            return

        requests.post("http://192.168.1.10/report/poisonivy", data=results["poisonivy"])
```

Prinimalka Signature

A signature module that looks for captured behavior typical for the Prinimalka malware family.

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

class Prinimalka(Signature):
    name = "banker_prinimalka"
    description = "Detected Prinimalka banking trojan"
    severity = 3
    categories = ["banker"]
    families = ["prinimalka"]
    authors = ["nex"]
    minimum = "0.5.1"

    def run(self):
        server = ""
        path = ""

        for process in self.results["behavior"]["processes"]:
            for call in process["calls"]:
                if call["api"] != "RegSetValueExA":
                    continue

                correct = False
                for argument in call["arguments"]:
                    if not server:
                        if argument["name"] == "ValueName" and argument["value"] == "nah_opt_server1":
                            correct = True

                        if correct:
                            if argument["name"] == "Buffer":
                                server = argument["value"].rstrip("\\x00")
                    else:
                        break

                if server:
                    break

            if server:
                self.description += " (C&C: {0})".format(server)
                return True

        return False
```

Environment setup links for reference

- http://www.inetsim.org/
- http://www.honeyd.org/
- https://github.com/rep/misc/commit/92257bb479f5b16e92c4d474153834938fe77b9f