PyBox — A Python Sandbox

Markus Engelberth Jan Göbel Christian Schönbein
Universität Mannheim

Felix C. Freiling*
Friedrich-Alexander-Universität Erlangen-Nürnberg

Abstract: The application of dynamic malware analysis in order to automate the monitoring of malware behavior has become increasingly important. For this purpose, so-called sandboxes are used. They provide the functionality to execute malware in a secure, controlled environment and observe its activities during runtime. While a variety of sandbox software, such as the GFI Sandbox (formerly CWSandbox) or the Joe Sandbox, is available, most solutions are closed-source. We present the design, implementation and evaluation of PyBox, a flexible and open-source sandbox written in Python. The application of a Python based analysis environment offers the opportunity of performing malware analyses on various operating systems as Python is available for almost every existing platform.

1 Introduction

The growing amount, variety, and complexity of malicious software (malware) poses major challenges for today’s IT security specialists. It is often necessary to analyze different types of malware in order to understand and assess the resulting threats. However, since manual reverse engineering is time consuming, dynamic malware analysis has become a standard analysis approach in practice. In a dynamic analysis, malware is executed in a controlled environment and all actions during runtime such as changes in the registry or access to the network are recorded. A log file provides the analyst with a first and often sufficient overview over the basic functionality of the malware.

1.1 Related Work

Several sandbox solutions have been developed for Windows systems in the past. CWSandbox [WHF07] is a widely used sandbox tool which is now commercially available under the name “GFI Sandbox” [Sof]. While it is possible to use the sandbox over a web interface as part of a research project [Fri], the sandbox software itself is closed source. The basic idea of CWSandbox is to “hook” specific Windows API calls, i.e., to divert

*Contact author. Address: Am Wolfsmantel 46, 91058 Erlangen, Germany.
control flow into own code in user mode before executing the original API call.

A tool similar to CW Sandbox is Joe Sandbox [joel1]. The primary difference to CW- Sandbox lies in the way how malware is observed. While CW Sandbox applies user-mode hooking, the behavior engine of Joe Sandbox is implemented as a Windows driver and therefore runs in kernel mode. So in addition to being able to monitor malware running in kernel mode, it is also much more difficult for malware to detect the analysis environment. Still, Joe Sandbox is also a commercial closed-source system.

While not being commercial, Anubis (formerly known as TTAnalyze) [BMKK06] is a sandbox system for research. However, Anubis is not open-source and can only be accessed through a web frontend\(^1\).

Leder and Plohmann [LP10] developed a user-mode sandbox which is open-source [PL]. The idea of their system is to inject an entire Python interpreter into the process to be monitored. The monitoring functionality is then provided through external Python scripts. In contrast to CW Sandbox, this provides more flexibility and a higher degree of configurability. The major drawback of this approach is the severe performance overhead that is introduced.

Another open-source sandbox solution is Cuckoo [Cla10], which is developed by the Honeynet Project. Cuckoo is a complete and therefore also rather complex framework for malware analysis including control of virtual machines to execute malicious software on.

### 1.2 Contribution

In this paper, we describe the design and implementation of an open-source sandbox called PyBox (Python Sandbox). PyBox combines ideas from existing sandbox systems to create a unique and expandable tool for automated malware analysis:

- PyBox implements user-mode API hooking to monitor the behavior of executed software, i.e., it injects a dynamic-link library (DLL) into created processes in order to intercept API calls and the according arguments.

- PyBox allows execution control (resume, abort) during runtime.

- PyBox itself uses a programming interface based on Python for improved flexibility.

The main novelty of PyBox compared to existing sandboxes is the fact that it is open source as well as its flexibility. While comparable closed source software may work well in business environments, PyBox as an open source product targets to be used in research as a building block for new malware analysis tools in the future. This way, researchers can extend PyBox’s functionality and build customized solutions for their special analysis targets or work together to create a more powerful and portable analysis tool. The required flexibility is achieved by using Python as programming language for the analysis tool. Due to its easy to understand syntax Python is widely used and available for almost every

\(^1\)See [http://anubis.iseclab.org/](http://anubis.iseclab.org/).
platform. Therefore, the interface can be used to analyze targets on various systems by providing the respective hook libraries. Python also offers the required low-level support to interact with operating systems as well as integration with the used virtualization software. All this renders Python a perfect choice for our analysis tool. PyBox can be downloaded from the PyBox homepage [pyb].

After giving a short background on the techniques used by PyBox in Section 2, we briefly describe the design requirements of PyBox in Section 3. We give an overview of the implementation in Section 4 and present a brief functional evaluation in Section 5. Section 6 concludes the paper.

2 Background

This section provides some general background on techniques used by PyBox.

2.1 Inline API Hooking

Hooking is a concept used to gain control of a program’s execution flow without changing and recompiling its source code. This is achieved by intercepting function calls and redirecting them to infiltrated customized code.

PyBox implements so-called inline API hooking [HB08, Eng07] to monitor software behavior. The complete process is displayed in Figure 1. Inline hooks directly overwrite the function’s code bytes in memory. In particular, only the first few instructions are replaced with a five byte jump instruction to the actual hook function. The replaced instructions are stored in the trampoline function, which is used as a new entry point to the original API call. Within the hook function the logging of information or modification of arguments can take place before the original API function is executed.

2.2 DLL Injection

In order to make use of hooked API functions within the process space of the software that we want to analyze, we first need to inject code into the target process. The mechanism we apply for this purpose is called DLL injection. We instrument the API functions CreateRemoteThread and LoadLibrary that enable us to create a new thread in the target process and load a DLL file which in turn installs the hook functions.
3 Design of PyBox

The PyBox analysis environment consists of three major parts: a virtual machine, the analysis tool PyBox.py, and the hook library pbMonitor.dll. Each is described in more detail in the following paragraphs. A schematic overview of the different parts of PyBox and their interaction is displayed in Figure 2.

**Virtual Machine.** Using a virtual machine as a basis for malware analysis guarantees a secure and controlled environment in which the malware can be executed and the original system state can be restored afterwards. Inside the virtual machine we can observe system activity of a certain target software during runtime.

**Analysis Tool.** The analysis tool, called PyBox.py, acts as the hook server. The hook server is responsible for setup adjustments according to the settings defined in the configuration files, target process creation, and hook library injection. During the execution of malicious software, it also receives and processes the log data from the hooked API functions and in the end generates a final behavior report.

**Hook Library.** The hook library pbMonitor.dll implements the actual hooking and monitoring functionality. It is responsible for installing the specified hooks, monitoring the system calls, and creating log entries, which are then sent to the hook server. Therefore, the hook library and the hook server have to interact with each other very closely by means of *inter-process communication* (IPC). This way information exchange between the two processes is straightforward.

The hook library pbMonitor.dll is the only component implemented in *Visual C++*. 

Abbildung 1: Inline hooking [Eng07, p. 33]
In this case we have chosen C++ as programming language because Python cannot create DLL files and we have to make much use of various API functions provided by Windows. This requires the use of specific C data structures and is therefore more comfortable to program in C++.

### 4 Implementation Excerpts

Describing the complete implementation of PyBox would be out of the scope of this paper. We therefore focus on a few details only. For more information see Schönbein [Sch11].

#### 4.1 Callbacks and Trampolines

The callback function allows us to execute our own customized code within the address space of the target process. Hence, we implement the monitoring functionality here and send the resulting data to the PyBox analysis tool.

```c
return_type calling_convention callback_function(arg1, ..., argn)
{
    return_type status;
    info = obtain_information(arg1, ..., argn);
    if (prevent_execution == false)
    {
        status = trampoline_function(arg1, ..., argn);
    }
}```
else
{
    status = customized_return_value;
}
create_log_entry(info);
return status;

Listing 1: Callback function structure

The basic structure of such a callback function is depicted in Listing 1. The variables named `prevent_execution` and `customized_return_value` represent settings that have been specified by the user beforehand. The value of `prevent_execution` determines whether or not the trampoline function is called in order to execute the original API function. In case it is not called, a custom value is returned. Finally, the function `create_log_entry` notifies the analysis environment about the API function that was executed and the arguments that were used.

In order to describe the usage of callback functions in more detail, we present the implementation of a sample callback function in the following. The function `NtCreateFile` is provided by the native API. It is usually called in order to create a new file or to open an existing one. The interface requires eleven arguments that all have to be passed for a function call. The most important ones are described in the following:

- **FileHandle** is a pointer to a file handle corresponding to the created or opened file after the function has been executed.
- **DesiredAccess** defines the requested access rights concerning the file.
- **ObjectAttributes** is a pointer to a structure containing information about the requested file such as the file name and its path.
- **AllocationSize** is an optional pointer to the size of the initial allocation in case a file is created or overwritten. This pointer can also be NULL implying that no allocation size is specified.
- **FileAttributes** contains flags specifying the file’s attributes. Such attributes can for example mark a file as read-only or hidden.

The corresponding callback function is shown in Listing 2.

16 NTSTATUS NtAPI NtCreateFile_callback(
17   .out PHANDLE FileHandle ,
18   .in ACCESS_MASK DesiredAccess ,
19   .in OBJECT_ATTRIBUTES ObjectAttributes ,
20   .out PIO_STATUS_BLOCK IoStatusBlock ,
21   .in_opt LARGE_INTEGER AllocationSize ,
22   .in ULONG FileAttributes ,
23   .in ULONG ShareAccess ,
24   .in ULONG CreateDisposition ,
25   .in ULONG CreateOptions ,
26   .in_opt PVOID EaBuffer
)
27     \_ in \ ULong EaLength
28 }  
29 {
30 \ NTSTATUS status ;
31 ...  
32   
33 \ if ( hook.settings [0].preventExecution == FALSE )
34 {  
35     // Call trampoline function
36     status = NtCreateFile.trampoline ( FileHandle , DesiredAccess ,
37           ObjectAttributes , IoStatusBlock , AllocationSize , FileAttributes ,
38           ShareAccess , CreateDisposition , CreateOptions , EaBuffer , EaLength );
39         executed = 1 ;
40 }  
41 \ else
42 {  
43     // Get customized return value
44     status = ( NTSTATUS ) hook.settings [0].returnValue ;
45 }  
46  
47 // Create log entry and return
48 SOCKET_ADDRESS_INFO sai = \{ 0 \};
49 createLog ( 0 , object , L "" , "" , executed , DesiredAccess , FileAttributes ,
50           ShareAccess , CreateDisposition , CreateOptions , sai , status );
51 \ return status ;
52  

Listing 2: The NtCreateFile callback function

As soon as all required information is gathered, it is determined whether the original API is to call as well, or if a predefined value should be returned. This check is implemented in line 57. Finally, the callback function returns the value of status (line 72) to the object that has called the API function.

An important piece of information that is passed to the createLog function in line 71, is the file path of the target file associated with the API call. All information concerning the target file of a NtCreateFile call can be found in the argument ObjectAttributes. This structure contains the two fields providing the information we are looking for: RootDirectory and ObjectName. In order to obtain the complete path to a target file, we simply have to combine these two parts. The resulting string is stored in the variable object and finally used to create the associated log entry.

4.2 Detection Prevention

More recent malware samples implement methods to inspect the system environment they are executed in, to detect analysis software. Since we do not want malware to behave differently than on a usual personal computer, we need to avoid easy detection of the PyBox analysis environment.

Possible techniques applied by malware in order to examine the system are to list all running processes or to scan the file system. Often, certain API functions are applied in order to implement the required detection techniques. In PyBox, we consider two different methods. The first method is to use the API function CreateToolhelp32Snapshot in
order to create a snapshot containing a list of all running processes and to parse this list using the API functions `Process32First` and `Process32Next`. The second method considered is to use the API functions `FindFirstFile` and `FindNextFile` in order to scan the filesystem.

We use the hooking functionality to intercept these API functions and successfully hide files and processes of PyBox. If an object returned by one of the above mentioned API functions belongs to the analysis framework, the corresponding trampoline function will be called again until it returns an object which does not belong to the analysis framework or until there is no more object left in the list to be queried. In this way, PyBox is basically invisible to other processes of the analysis system, in particular to the examined target process, too.

### 4.3 PyBox Analysis Tool

The PyBox analysis tool is the interface between the hook library and the analyst. It enables the following tasks: configuration management, process creation, library injection, data processing, and report generation.

![Abbildung 3: PyBox analysis tool layout](image)

In order to fulfil these tasks, PyBox includes various packages, modules and configuration files, as illustrated in Figure 3.

In the upper part of Figure 3, the five packages belonging to the PyBox analysis tool are depicted. The package `setup` is responsible for reading the provided configuration files and for extracting their information. Furthermore, it converts this information into objects that can be used by the analysis tool. The package `process` allows to execute process-related operations such as the creation of the target process. Additionally, it provides required process-specific information. The DLL injection method is implemented in the package `injection`. The final report generation operations are made available by the package `report`. More precisely, it offers the functionality to process the data received from the monitored process and turn them into a XML-based report. The communication and inter-
action with the target process is implemented in the package ipc.

The lower part of Figure 3 displays the two configuration files named pybox.cfg and hooks.cfg. The file pybox.cfg defines settings that are required by the analysis framework to work. In particular it contains the path to the hook library, which is injected into the target process. In contrast, the file hooks.cfg influences the execution of the hook library inside the target process. More specifically, it defines which hooks to install and, thus, which API functions to monitor.

During the start of the execution of PyBox.py, all hooks are read and mapped to an array of C data structures that are present both in the PyBox main application and the hook library. Each hook has a specific identification number by which it can be identified. Once the array is created out of the configuration file, the hook library can obtain the array and install all specified hooks. More details about logging and the generation of the XML report can be found in Schönbein [Sch11].

5 Evaluation

In order to test the functionality of PyBox, we have tested the analysis environment examining different malware samples. Due to space restrictions, in this section, we present only one example. More examples can be found in Schönbein [Sch11].

The malware sample analyzed here can be categorized as back door, trojan horse, or bot. Its common name is “Backdoor.Knocker”. Information about this threat is provided by VirusTotal [Vir], Prevx [Pre], and ThreatExpert [Thr]. ThreatExpert refers to it as Backdoor.Knocker. Its purpose is to frequently send information such as the IP address or open ports of the infected system to its home server.

During the analysis run of this sample it did not come to an end. Therefore, the analysis process terminated the target process after the specified time-out interval of two minutes.

227 \texttt{<NtCreateFile CreateDisposition ="0x00000005" CreateOptions ="0x00000064" DesiredAccess ="0x40110080" Executed ="YES" FileAttributes ="0x00000020" Object ="\texttt{C:|WINDOWS|system32\|cssrss.exe}" ReturnValue ="0x00000000" ShareAccess ="0x00000000" Timestamp ="3115"/>}

228 \texttt{<NtWriteFile Executed ="YES" Length ="20928" Object ="\texttt{C:|WINDOWS|system32\|cssrss.exe}" ReturnValue ="0x00000000" Timestamp ="3116"/>}

Listing 3: Extract from the file management section of the malware sample’s analysis report

The first interesting aspect of the behavior report is in line 227 of Listing 3. The file section documents that the target process has created a file in the Windows system folder and written data to it. The file has been named cssrss.exe. These entries also indicate malicious behavior as the file name is very similar to the file name csrss.exe. The latter is the executable of the Client Server Run-Time Subsystem (CSRSS), which is an important part of the Windows operating system. It seems that the analyzed object is named similar to a system process to hide its presence.
The behavior report's registry section also contains many entries, which have to be considered. The process queries various information about system and network services as well as about the operating system such as the system's version. One particularly noticeable part of the registry section is depicted in Listing 4. The report reveals that the target process uses registry API functions to change the value EnableFirewall (line 1212) and furthermore to add its executable file to the list of authorized applications (line 1214). These entries are clear evidence that the application tries to disable security mechanisms and thus performs malicious behavior unwanted by the system's user.

6 Conclusion and Future Work

Given the available commercial dynamic analysis tools on the market, PyBox is the first publicly available open-source tool which written in a flexible and platform independent programming language. This allows PyBox to be easily improved in many ways.

In PyBox we currently only hook native API functions to monitor the behavior of malware samples. But different Windows versions often use very different native API functions. Therefore, the hook library has to be extended in order to provide compatibility with various Windows versions.

An approach for extending the functionality of PyBox is to add further hooks to the analysis framework in order to monitor more types of activity. Furthermore, the analysis tool's report generation can be extended by mapping provided numeric values such as file creation flags to strings that describe the meaning of the activated flags in order to simplify the interpretation of the report. A third approach is to implement a hook library for other operating systems such as Android and thus provide the opportunity to analyze malware samples designed to attack mobile devices. In order to provide more flexibility and exten-
dibility for malware analyses we could incorporate the functionality of a C compiler. We could use a special Python script and create the hook library’s code out of existing code fragments according to the configured settings specified by the analyst. Thus, we would implement a modular solution that automatically compiles a separate, customized hook library that can be injected into the remote process to be monitored. In doing so, we would have a more flexible solution, which also considers the performance criterion.

Literatur


