Beatrix: A Malicious Code Analysis Framework

Christian Wressnegger
christian@wressnegger.info

Abstract

Especially in recent years malicious codes (malware) and the techniques used to bypass modern detection systems evolved at a tearing speed. We believe that in research a more advanced way of cooperation on malware detection is needed to bring forward security in general. Up to now there is no system which assists researchers in doing so. We are working on bridging this gap and present *Beatrix* as an analysis framework. It allows researchers to focus on the development of new techniques instead of bothering about visualization, testing environments or the publication and distribution of the final prototypes. Especially simplifying the latter results in increased availability of state-of-the-art developments others may build on top of. For this purpose, we provide a software framework which introduces a plug-in infrastructure, that breaks down the the detection process into disjoint sub-tasks. Hence, it is not only possible to utilize other prototypes but even single components of it without forcing a special license or demand open source implementations. The benefit for each individual developer arises from the reduced implementation effort for a complete prototype, since the framework already takes over significant parts for binary analyses.

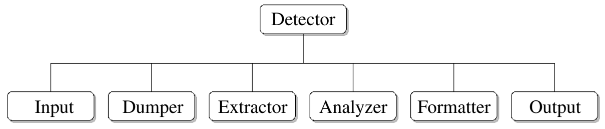

Figure 1: Framework structure

Structure

The Beatrix Framework splits up the task of malicious code detection into the six categories shown in Figure 1. The internal prossing scheme is as follows: we iterate over the set of Inputs and pass each piece of input one by one to the Dumper. The results of the Dumper are post-processed by a number of Extractors or left as they are if no Extractor is specified. Based on these results one or more Analyzers perform the actual classification, which again is optionally post-processed by a set of Formatters. The final results are handled by at least one Output module.

Inputs

The initial step of the detection process is to retrieve some kind of input, i.e. the piece of binary data that should be analyzed. Although, reading data from a given resource or a file is an elementary task, it still enables a broad range of interesting and reusable applications. First of all, reading in a number of benign and malicious files is used for most evaluations and, of course, is possible at this stage of the framework as well. One may extend this, for instance, by retrieving the samples using shellcode and/or exploit generators. Alternatively it is possible to take a more practical line and replace this by a scanner which passes the files that just performed specific system-level actions (network access, file access, etc.) to the framework. Reading data from network is another interesting scenario for testing methods in the field of network intrusion detection systems.

Dumpers

The raw byte stream provided by the input layer is processed and interpreted by a Dumper. In general this type of modules provides verbose information about the inspected suspect. Hence, this is usually the point to operate against obfuscation and stealth techniques, anti-debugging and anti-virtualization, etc.
Dumpers are able to navigate through the provided stream and therefore, it controls which bytes are provided next **by** the framework. To do so a Dumper module only specifies the number of bytes which it processed

and the offset, where it wishes to start the next iteration. This allows the implementation of overlapping analysis of data or to perform a rollback in case the processing of the Dumper failed for some reason. An example would be the extraction of execution chains and the thereby necessary disassembly step. If a byte sequence is detected that does not represent a valid instruction, the starting offset might have been wrong. In such a case the Dumper tags the first byte as invalid and the next time the framework calls the Dumper with the offset increased by one. That way the logic and implementation effort for navigating and handling the streams are supplied by the framework, but the supervision completely remains with the module.

Extractors

Extractors are intended to filter the findings of the Dumper in order to provide meta properties and occurrences. For this purpose they may draw first conclusions and apply heuristics such as hit thresholds, clusters of simultaneously or sequenced occurred events, etc. Assuming, for instance, the Extractor receives a list of all system calls invoked by the currently inspected suspect. The aggregation of a sequence of `GetProcAddress` and `LoadLibrary` calls to a meta property may act as an indicator for Windows shellcodes.
A special property of this category is that it is possible to order its modules, i.e. if an Extractor e_1 is executed before another Extractor e_2, then e_2 is able to access the results of e_1. Thus, it is possible to develop more abstract meta objects out of existing once over and over again. At the end, all the results of the available Extractors and the Dumper are passed to each Analyzer.

Analyzers

As the name implies, the actual analysis and classification is done using these modules. Analyzers operate on the pool of data provided by the Extractors and the Dumper in order to trigger classification events whenever one occurs. Such an event consists of at least the input specification (details on the underlying resource, position within the stream, etc.) and time/ date of the classification. The details about the actual classification is up to the particular implementation of the Analyzer.

Formatters

Formatters are post-processors for Analyzers, or, to put it another way, they are mediators between Analyzer and Output modules. This category is intended to increase the interoperability between the Beatrix Framework and other tools, e.g. for processing the classifications. Many systems use a common standard to provide that interoperability. Although we appreciate this movement, we use Formatters to transform the internal representation to arbitrary formats instead of relying on a single format. As a result, external tools which might not be flexible enough to parse different classification formats and did not make an effort to implement that standard, still can be used. However, due to this flexibility we still have the possibility to provide events in various standardized formats as a Formatter module.
In case no proper module is specified, only the original classification is passed to the Output layer.

Outputs

At the end a detector has to provide the final results. Next to the most obvious scenarios, as for instance, writing to log files or the standard output there are more advanced variations which are often tedious to implement. For instance, the interaction with other applications, as motivated in the previous section, might be done by interprocess communication or network. Furthermore, this type of module is the only one which is designed to implement a full featured graphical user interface (GUI).
Output modules only define the way data are handled. Due to the Formatter modules, they are completely decoupled from the actual output representation. Therefore, it is not necessary to modify an Output module implementation in order to change the actual output.

Tooling

When talking about *Beatrix* (*Project*) one differentiates between two major components: the *Beatrix Detector* and the *Beatrix IDE*. Both are based on the *Beatrix Framework* which in turn makes use of *Beatrix Modules*. The **Beatrix Detector** is a stand-alone application which performs the actual detection process based on a configuration file and its program arguments. This configuration file specifies the individual plug-ins (so-called Beatrix Modules) to use and contains their required parameters. Therefore, the Beatrix Detector represents the “operational implementation” of the Beatrix Framework.
The set of tools for using this infrastructure is collectively called the **Beatrix IDE**. It is intended to assist the developer in composing an individual detector (the mention configuration) from scratch. Furthermore, the IDE is able to start these detector configurations in a debugging mode (independent of the Beatrix Detector but sharing the same code base – the Beatrix Framework). This debugger allows one to inspect the data which are currently processed within each plug-in used (no matter if third-party or own developments) and navigate through their execution.

Discussion

The framework at its current state of development is written in the Java programming language. People are often very critical concerning Java when doing implementations in fields of application where it usually matters to have a low-level point of view and to keep an eye on execution performance. However, we argued that *Beatrix* is designed for prototyping and to try to get the bottom of detection performance. Concerning this, Java seems to be the optimal choice due to its capability to integrate native libraries or almost arbitrary scripting languages.
Nevertheless, we clearly underestimated the value of the general request to slip execution performance in prototyping as well. Due to this, we are rebuilding the framework to fit this need, but simultaneously preserve the concepts and ideas of flexibility we introduced. See the next section for details.

Future and Current Work

We implemented a number of modules which cover the most usual cases of application for input and output handling: reading from single and lists of files, processing of network streams, writing log files, listing classification events in a simple GUI, etc. In the course of that we worked on the integration of the MetaSploit Framework [4] as a supplier for malware samples.
As a basic showcase we drew on the most basic malware detection setup, namely a signature matcher. This showcase was mainly developed to demonstrate the existing modules for in- and output handling. This evolved from the initial implementation via using different third party signature databases through to the attempt to provide the technologies used in the Clam AV [1] engine as module for the framework. Another module we worked on is designed to utilize the Linux ptrace debugging interface [6] for the extraction of execution chains.
Furthermore, we are permanently evaluating the possibility of integrating prototypes based on Beatrix in other systems and applications, as for instance, the Snort IDS [5]. At the moment especially in the mentioned example of an intrusion detection system this proved to be contradicting regarding execution performance. Directly related to this is our recent attempt to provide a low-level implementation of the framework to meet the community's demands as mentioned in the previous section.
Therefore, we are forcing a C/C++ implementation of the basic framework, i.e. we reverse our initial approach of utilizing native libraries from within Java to utilizing Java (among others) using a native build of *Beatrix*. We expect a significant performance improvement for computational intensive tasks and therefore, more acceptance in the research community.

Related Work

High quality software products usually introduce some kind of abstraction of their internal structures to keep maintenance and future development easy. However, they are mostly not intended to be extended externally. Others have the capability to process plug-ins, as *Beatrix* does, but only for one specific task. Those plug-ins allow (third-party) developers to extend the product without changing the actual product itself. We see an increasing amount of such system in maleware/ security research as well, as for instance Snort, the Anubis Project [3], etc.
However, *Beatrix* enables to fully specify its main behaviour (i.e. the complete detection process) by plug-ins, though. Therefore, we take a similar line as, for instance, the MetaSploit Framework. It allows users to put together a shellcode, such that it fits their needs. One is able to select the payload to use, the characters (byte values) that are not permitted or the encoder to obfuscate the resulting code. All these steps for generating a shellcode are provided as (Ruby) modules and can be added or removed by third-party developers without modifying the framework. Hence, MetaSploit makes available and specifies the infrastructure and outsources substantial tasks to the framework. This exactly is the paradigm we are following as well.

Availability

The complete *Beatrix Project* is free software under the terms of the GNU Public License version 2 [2] with a so-called classpath exception. That exception allows explicitly the usage of the framework in and with other software without forcing a specific license of the final product. The applications of the project, their source codes and more detailed documentation for users and developers are available from the corresponding project website.

`http://beatrix.sf.net`

References

[1] ClamAV Team. ClamAV. URL: http://www.clamav.com.

[2] Free Software Foundation. GNU GENERAL PUBLIC LICENSE version 2. URL: http://www.gnu.org/licenses/gpl-2.0.txt, June 1991.

[3] International Secure Systems Lab, et al. Anubis: Analyzing Unknown Binaries. URL: http://anubis.iseclab.org/, 2009.

[4] Metasploit LLC. Metasploit™ Framework. URL: http://www.metasploit.com.

[5] Sourcefire, Inc. SNORT. URL: http://www.snort.org.

[6] The Linux *man-pages* project. ptrace - process trace. URL: http://www.kernel.org/doc/man-pages/online/pages/man2/ptrace.2.html.