Karta Documentation

Release 1.2.0

Eyal Itkin

Apr 20, 2020

User Guide:

CHAPTER 1

Prerequisites

Karta makes extensive use of 2 python packages:

- elementals
- sark

Using the `setup.py` script, one can install all of these prerequisites, and be ready to go: `./setup.py install`

CHAPTER 2

Installing the Plugin

Nothing should be done :) There is no need to copy the directory to some IDA plugin folder. Instead, once your binary is loaded to IDA, simply start the desired plugin by loading it using the `File->Script File...` menu. That's it.

CHAPTER 3

Thumbs Up

3.1 Introduction

Karta is highly sensitive to the quality of the function analysis that was done by IDA. Therefore, we developed **Thumbs Up**. This mini-plugin should be used as a pre-process phase to automatically achieve to main goals:

1. Drastic improvement of the disassembler's analysis
2. For ARM binaries - clear seperation between ARM and THUMB code regions

More information about the script and it's Machine-Learning-based analysis, can be found in this detailed blog post: Thumbs Up - Using Machine Learning to improve IDA's Analysis.

Important Note Thumbs Up performs a series of major changes to the binary on which it was invoked. We highly recommend that you **backup** your original binary **before** executing the script. Better safe than sorry.

3.2 Firmware Files

Although the plugin was mainly designed for improving the analysis of firmware files, there are still some precondition steps that are required before executing the script.

1. Make sure that the different code segments are clearly defined in IDA
2. **Code** segments (executable and not writable) will be treated differently than **Data** segments (non executable)

The list of code segments and data segments will be outputed to the screen (and log) at the start of the script. Once the segments are properly configured, simply load the script file named `thumbs_up/thumbs_up_firmware.py` and wait for the magic to happen.

The script's performance heavily depends on IDA's analysis, as well as on the different phases it has to perform. On ARM binaries you should expect a much longer execution time than on other binaries, as it also needs to adjust the ARM/THUMB code transitions.

3.3 ELF Files

Executing the script on ELF files is easier, as the ELF header already defines all the information we need for the code segments. For ELF binaries one should load the script file named `thumbs_up/thumbs_up_ELF.py` and wait for the magic to happen.

CHAPTER 4

Open source fingerprinting

4.1 Identifier Plugin

The `karta_identifier.py` script identifies the existence of supported open source projects inside the given binary, and aims to fingerprint the exact version of each located library. Once your binary was loaded to IDA, simply load the script `karta_identifier.py`, and it will output the results to the output window and to an output file. Here is an example output after running the script on an HP OfficeJet firmware:

```
Karta Identifier - printer_firmware.bin:
========================================

Identified Open Sources:
------------------------
libpng: 1.2.29
zlib: 1.2.3
OpenSSL: 1.0.1j
gSOAP: 2.7
mDNSResponder: unknown

Identified Closed Sources:
--------------------------
Treck: unknown

Missing Open Sources:
---------------------
OpenSSH: Was not found
net-snmp: Was not found
libxml2: Was not found
libtiff: Was not found
MAC-Telnet: Was not found

Final Note - Karta
------------------
```

(continues on next page)

(continued from previous page)

```
If you encountered any bug, or wanted to add a new extension / feature, don't␣
↪hesitate to contact us on GitHub:
https://github.com/CheckPointSW/Karta
```

As can be seen, the output includes 3 parts:

1. List of identified open source libraries, with their version if identified or "unknown" if failed to identify it
2. List of identified closed source libraries
3. List of missing open source libraries, so that you will know what libraries are supported by the identifier at the moment

4.2 Manual Identification

Sometimes we would like to feed **Karta** with some knowledge we already acquired about the matched open source. When **Karta** locates a library, but fails to identify it's exact version, we can manually tell it the version so the matcher could match it. For example, in the above example we could manually configure the version for the "mDNSResponder" library which we located, but failed to identify.

User defined library versions can be declared by running the `karta_manual_identifier.py` in the command line, using the following arguments:

```
C:\Users\user\Documents\Karta\src>python karta_manual_identifier.py --help
usage: karta_manual_identifier.py [-h] [-D] bin

Enables the user to manually identify the versions of located but unknown
libraries, later to be used by Karta's Matcher.

positional arguments:
  bin          path to the disassembler's database for the wanted binary

optional arguments:
  -h, --help   show this help message and exit
  -D, --debug  set logging level to logging.DEBUG
```

The script will store the configurations in a `*_knowledge.json` file near the disassembler's database file.

Note: After we manually identify the version of a previously located but unknown library, future calls to the identifier plugin will use our supplied version automatically.

CHAPTER 5

Matching supported libraries

5.1 Prerequisites

Identifier

It is always recommended to start with the identifier script, so you would know if you already have pre-compiled configurations for all the libraries you need. In case it is needed, a guide for compiling a new configuration can be found in the next section.

Function Analysis - Thumbs Up

Karta is highly sensitive to the quality of the function analysis that was done by IDA. It is important to make sure that the matcher plugin is invoked only *after* the binary is well analyzed. For example: even if there is an un-reffed code snippet, make sure that IDA marked it as a function if it is an un-reffed function. It is highly recommended to use **Thumbs Up** for automatic improvement of IDA's analysis.

5.2 Manual Anchors

Sometimes we would like to feed **Karta** with some knowledge we already acquired about the matched open source. In this case we can define "manual anchors", and **Karta** will use them as part of the initial anchors list. User defined anchors can be declared by running the `karta_manual_anchor.py` in the command line, using the following arguments:

```
C:\Users\user\Documents\Karta\src>python karta_manual_anchor.py --help
usage: karta_manual_anchor.py [-h] [-D] [-W] bin lib-name lib-version configs

Enables the user to manually defined matches, acting as manual anchors, later
to be used by Karta's Matcher.

positional arguments:
  bin            path to the disassembler's database for the wanted binary
  lib-name       name (case sensitive) of the relevant open source library
```

(continues on next page)

(continued from previous page)

```
  lib-version    version string (case sensitive) as used by the identifier
  configs        path to the *.json "configs" directory

optional arguments:
  -h, --help     show this help message and exit
  -D, --debug    set logging level to logging.DEBUG
  -W, --windows  signals that the binary was compiled for Windows
```

The script will store the configurations in a `*_knowledge.json` file near the disassembler's database file.

5.3 Matcher Plugin - Start

Assuming you are all set and ready to go, and that your binary is already open in IDA, load the `karta_matcher.py` script and set up the needed configurations:

- Full path for Karta's configuration directory - the `configs` dir with all of the `*.json` files
- In case of a binary that was compiled for Windows, set up the checkbox (not required for firmware binaries)

Once again, the output will be shown in IDA's output window, and will also be stored to a file. Every matched open source library will open 2 windows:

1. Window with the match results from the library
2. Window with the proposed match results for external (usually libc) functions, used by the open source library

5.4 Matcher Plugin - Output

The matched library functions include the reason for the matching. As some matching rules are much more accurate than others, they are colored in dark-green, while the others are marked in green. You can now select a subset of matches, right click, and export the selected matches to be names in IDA. Or, you can simply right click and import all of the matches directly to IDA.

The matching process is relatively fast (less than a minute for a small-medium open source), however no user interaction is needed after each library is matched, so you can also run it at night and check all of the results in the morning.

CHAPTER 6

Compiling a configuration file

6.1 Compiling the Open Source

Adding support for a new version for an already supported library, requires only to compile a new (`*.json`) configuration file for it. As **Karta** is a source code assisted plugin, this process requires you to compile the open source library according to the guidelines of the open source project, together with the specific guidelines that can be found in the `compilations` directory.

Important: Karta will need two compiled parts for building the configuration

1. A static library - .a in Linux, and .lib in Windows
2. A folder containing all of the *.o (in Linux) or *.obj (in Windows) files

Note: Some libraries, such as OpenSSL, are split to several static libraries. In this case you should make sure you found all of the parts for each such static library

6.2 Running the script

Now that we have all of the parts, we should run `karta_analyze_src.py` in the command line, using the following arguments:

```
C:\Users\user\Documents\Karta\src>python karta_analyze_src.py --help
usage: karta_analyze_src.py [-h] [-D] [-N] [-W]
                            lib-name lib-version dir archive [dir archive ...]

Compiles a *.json configuration file for a specific version of an open source
library, later to be used by Karta's Matcher.

positional arguments:
  lib-name          name (case sensitive) of the open source library
  lib-version       version string (case sensitive) as used by the identifier
  dir archive       directory with the compiled *.o / *.obj files + path to
```

(continues on next page)

(continued from previous page)

```
                    the matching *.a / *.lib file (if didn't use "--no-
                    archive")

optional arguments:
  -h, --help          show this help message and exit
  -D, --debug         set logging level to logging.DEBUG
  -N, --no-archive    extract data from all *.o / *.obj files in the directory
  -W, --windows       signals that the binary was compiled for Windows
```

Note: The script must be executed from Karta's src directory.

1. Name of the open source library (case sensitive)
2. Version of the library (as will be identified by the identifier script)
3. Path to the directory that contains the compiled (*.o / *.obj) files
4. Path to the compiled static library file (if "–no-archive" wasn't used) In case there are multiple static libraries, simply extend the list of "dir archive" (with archive) or list of "dir" (without archive), depending on the "–no-archive" flag)

The script will ask you for the path to your disassembler (IDA), and will suggest a default path. Enter the path to your disassembler, press ENTER, and a progress bar will show you the progress of the script.

6.3 Storing the config file

In the end, a new `*.json` file will be generated (using the library name + version), and it should be stored together with the rest of the configuration files in the `configs` directory.

CHAPTER 7

Compilation Guidelines

7.1 Basic Invariant

Karta's main compilation assumption is that the source compilation can't modify (inline / split to parts) a function if the wanted binary hadn't done the exact modification to this function. This means that:

1. A function can be modified (inlined) in the binary even if we didn't inline it in our "source" compilation
2. If a function was modified in our "source" compilation, it must be modified in the same way in our wanted binary

Since we want to maintain this basic invariant, we usually want to compile our open source library with flags for:

- No inlining
- No compiler optimizations

7.2 Windows Compilation

It seems that when compiling a binary using `nmake` or `visual studio`, the Window's compilation adds some linker optimizations. As we couldn't imitate these linker optimizations when compiling with `gcc`, **Karta** can (and should) support 2 different configurations for the same version of a specific library:

1. Basic (unix) configuration - Used for Linux, Mac, of various firmware files
2. Windows configuration

7.3 Bitness - 32 vs 64

After various testing rounds, it seems that a configuration for 32 bits can also achieve great matching results for 64 bit binaries. Therefor there is no need to maintain two different configurations files, one for each bitness mode. When compiling a configuration file, the rule of thumb should be:

- Basic (unix) based configurations should be compiled for 32 bits (-m32) - firmware binaries are usually 32 bits
- Windows configurations should be compiled for 64 bits

7.4 Updating the compilation notes

After a successful compilation was made, a new "compilation tips" file should be created and stored under the `compilations` folder. The file's name should be `<library name>.txt` and it should have a similar structure as the already existing files.

7.5 Adding a python identifier for your library

As most of the open source projects have unique string identifiers that hold their exact version, all of the currently supported fingerprinting plugins are based on a basic string search.

searchLib(): Scans all of the strings in the binary (using the `self._all_strings` singleton), in search for a key string (holding the version) or a unique library string that is stored locally near a clear version string.

identifyVersions(): Will be called only after `searchLib` had identified the existence of the library in the binary. This function is responsible for parsing the exact library version, usually using the `self._version_string` that was found by `searchLib`.

CHAPTER 8

Adding support for a new open source

Support consists of two parts:

1. A fingerprint *.py file with the logic required for locating the library
2. An initial configuration file for a chosen version

Compiling the configuration is done exactly as described in the prior section. However, you should make sure to document the flags you changed in the project's Makefile, by storing your guidelines in the `compilations` folder under a new *.txt file named after your open source library.

Adding a new *.py file for the identification script is rather simple. The needed steps are:

1. Copy some existing file from the `libs` folder (`libpng.py` for instance) to a new *.py file with the name of your library, and place it too under the `libs` folder
2. Update the `__init__.py` file with your library, and place your new import line at the **end** of the list
3. Update the name of the class
4. Update the `NAME` variable, with an exact string name (case sensitive)
5. Update the logic of `searchLib()` method - currently based on a basic string search
6. If needed, update the logic of the `identifyVersions()` method

CHAPTER 9

Karta

9.1 Motivation

The main motivation for developing **Karta** was the need to identify open sources in large firmware files. My previous experience with other available tools (at the time) was that they have a memory blowup when dealing with large binaries, meaning that sometimes they will completely crash and give no results :(

If we could work with a subset of functions, that will be polynomial to M (number of functions in the open source) and not in N (number of functions in the binary) we could escape the limitations that arise when M << N. And this was the main idea.

9.2 Key Idea - Linker Locality

Matching two functions (src and dest) is usually done after converting them into some "canonical" representation. We aim to narrow our search space, and to convert only a minimal set of binary functions into their canonical representation. And here comes the linker to our rescue:

- The compiler usually compiles each source file (.c / .cpp) into a single binary file (.o or .obj depending on the compiler)
- The linker then attaches them all together into a single blob
- This blob will be inserted to the firmware **as is**

conclusion #1: The compiled open source will be contained in a single contiguous blob inside the firmware / executable.

conclusion #2: Once we find a single representative of that blob (a.k.a **anchor**), we can speculate about the lower and upper bound of this blob in the binary, according to the number of functions we know that should be in the blob

9.3 Matching Steps

Using these conclusions, **Karta** matches each open source using the following steps:

1. Fingerprint: Identify the existence of the open source, and the version that is being used
2. Search for **anchor** functions: functions with unique and rare artifacts (strings or consts)
3. Draw basic file boundaries: a map for each located file, and overall scope for the entire open source
4. Use **file hints**: search and match functions that contain a string with their source file name
5. Locate **agents**: functions with file-unique artifacts (minor **anchors**)
6. Regular score-based matching:
    - Scoring similarities
    - Control Flow Graph (CFG) analysis
    - **Note:** gives special attention for geographic location

9.4 Geographic Location

Compilers tend (when they are nice) to preserve the order of the functions in the compiled binary. For example, if "foo()" was defined after "bar()" in the same source file, the compiled "foo" will usually be found right after the compiled "bar". This means that our matching and scoring logic will pay special attention to geographic characteristics:

1. Possible matching candidates must reside in the same file as our source function
2. Adaptively boost the score of neighbours (according to seen matching history)
3. Use neighbours to "discover" new matching candidates
4. Static functions shouldn't be referenced by functions from other files / outside of our open source

9.5 Modularity

Using these basic concepts, **Karta** was designed to be modular, to allow other matching libraries to use the basic file mapping logic.

CHAPTER 10

Scoring Tips

10.1 Brief

During the work on **Karta** I learned quite a few lessons about the nature of scoring algorithms for binary matching. This section will include a list of the tips I found useful, hoping they could help other researchers / developers as well.

10.2 Tips

1. **Anchor** functions can easily generate many matches later on.
2. Finding **anchor** functions should be done without any dependency on the way we later on match additional functions. **Anchor** functions are too important to be missed by optimizations.
3. The compiler *can* sometimes mess around with the order of the functions inside a single compiled binary file. However, it tends to keep the existing order as-is.
4. Don't give (non-constant) positive scoring to artifacts when there is a reasonable scenario in which low meaningfully different functions receive a "match" score only because of this artifact. For example: number of instructions, frame size, etc.
5. Don't jump to make score-based decisions. Round up all of the possible matching candidates, and only pick the promising ones - those who receive enough score points and are way ahead of their competitors.
6. Functions can be be complicated, store a full call order (path per ref, all paths per call), otherwise the call order will trigger a False Positive (a.k.a. **FP**).
7. Try to adaptively learn the characteristics of the matched binary through the eyes of matched couples. For example: does the compiler maintained function locality (matching neighbours)? what is the ratio between the instructions in the binary and the source?
8. Adaptive scoring changes after every match, we can't assume that a change in score implies we should double check / match our candidates.
9. Give bonus score for "exact matching" feature: all (>1) consts matched, all (>1) strings matched, num calls (>1) matched, ...

10. Small functions contain limited scoring artifacts. Double their score so they would have the chance to reach the scoring threshold.
11. Code blocks score is tightly coupled with instruction score, and their sum should be scored accordingly (they shouldn't be handled separately).
12. We can't assume we know the file order in advance, we will have to deduce it on the flight.
13. Using information from the single compiled files, we can see what functions are exported. Non-exported (static) functions can NOT be referenced by the integrating project (or even other library files when there is no inlining in the binary), and we can rely on this fact when we filter our candidates.
14. Large leftovers can lead to false flagging of an external function as an internal one. This mainly means we are prone to errors when two libraries are adjacent and use one another. It also means that several parts of the same library *must* be handled together (as was done in OpenSSL).
15. Scoring based on calls is good, however if we know that these calls are to the wrong functions (using knowledge from previous matches) we should update our score.
16. On Windows there are linker optimizations, and they really mess-up the call graphs and the assumptions about locality / static functions.
17. Basic support for linker optimizations (by detecting collision groups) can drastically improve the matching results.

CHAPTER 11

Disassembler

11.1 IDA

On it's initial version **Karta** was developed as an IDA plugin. However, the disassembler is mainly used for extracting artifacts from functions during the creation of the canonical representation. During this phase, we mainly use sark.

11.2 Supporting Other Disassemblers

Since **Karta** was developed to be modular, and because one of our researchers (Itay) mainly uses radare2, we added the ability to support other disassemblers.

The `src\disassembler\disas_api.py` file defines the interface needed by **Karta**, and can be split to 3 main parts (as can be seen inside the folder `src\disassembler\IDA`):

1. Basic API - finding the name of a function, getting a segment list, etc.
2. Cmd API - functionality for activating the disassembler from the command line.
3. Analysis API - core logic needed for creating the canonical representation of a function.

While the first 2 parts can be easily implemented as empty adapters without any logic, the 3rd part is a bit more complex. We recommend developers to read the code from `src\disassembler\IDA\ida_analysis_api.py` as an example implementation, when trying to implement the same functionality in the added disassembler.

CHAPTER 12

File Map Logic

The file map, and the logic that is implemented on top of it, is the key concept of **Karta**. While different implementations can use different scoring algorithms, and use different matching tactics, we believe that the file map can be used in other binary matching tools as well.

Having that in mind, we built our matching engine out of 2 main parts:

1. File Layer - describes the file map, and the ability to define a file in a given scope, mark functions inside as matched, etc.
2. Matching Engine - Basic matching engine that initializes the file map using anchors, including the logic needed in order to find those anchors.

As one can see, these 2 parts can be used by other matching tools, and so we've put them inside the `src\core` folder. The additional logic that **Karta** adds on top of these layers, such as file based matching tactics (searching for neighbours), or matching steps, is implemented in other classes that inherits from these basic class (and carry the same name).

We hope that other matching tools could integrate our file map logic, and hopefully will profit from it as well.

```
/$$   /$$                          /$$
| $$  /$$/                         | $$
| $$ /$$/   /$$$$$$   /$$$$$$  /$$$$$$    /$$$$$$
| $$$$$/   |____  $$ /$$__  $$|_  $$_/   |____  $$
| $$  $$    /$$$$$$$| $$  \__/  | $$      /$$$$$$$
| $$\  $$  /$$__  $$| $$        | $$ /$$ /$$__  $$
| $$ \  $$|  $$$$$$$| $$        |  $$$$/|  $$$$$$$
|__/  \__/ \_______/|__/         \____/   \_______/
```

CHAPTER 13

Brief

"Karta" (Russian for "Map") is an IDA Python plugin that identifies and matches open-sourced libraries in a given binary. The plugin uses a unique technique that enables it to support huge binaries (>200,000 functions), with almost no impact over the overall performance.

The matching algorithm is location-driven. This means that it's main focus is to locate the different compiled files, and match each of the file's functions based on their original order within the file. This way, the matching depends on K (number of functions in the open source) instead of N (size of the binary), gaining a significant performance boost as usually N >> K.

We believe that there are 3 main use cases for this IDA plugin:

1. Identifying a list of used open sources (and their versions) when searching for a useful 1-Day
2. Matching the symbols of supported open sources to help reverse engineer a malware
3. Matching the symbols of supported open sources to help reverse engineer a binary / firmware when searching for 0-Days in proprietary code

13.1 Identifier

Karta's identifier is a smaller plugin that identifies the existence, and fingerprints the versions, of the existing (supported) open source libraries within the binary. No more need to reverse engineer the same open-source library again-and-again, simply run the identifier plugin and get a detailed list of the used open sources. Karta currently supports more than 10 open source libraries, including:

- OpenSSL
- Libpng
- Libjpeg
- NetSNMP
- zlib
- etc.

13.2 Matcher

After identifying the used open sources, one can compile a `.json` configuration file for a specific library (libpng version 1.2.9 for instance). Once compiled, Karta will automatically attempt to match the functions (symbols) of the open source in the loaded binary. In addition, in case your open source used external functions (memcpy, fread, or zlib_inflate), Karta will also attempt to match those external functions as well.

13.3 Credits

This project was developed by me (see contact details below) with help and support from my research group at Check Point (Check Point Research).

13.4 Links

- https://github.com/CheckPointSW/Karta
- https://research.checkpoint.com/karta-matching-open-sources-in-binaries/
- https://research.checkpoint.com/thumbs-up-using-machine-learning-to-improve-idas-analysis

13.5 Contact

- @EyalItkin
- eyalit at checkpoint dot com