January 29, 2019

Better unpacking binary files using contextual information

Armijn Hemel

Better unpacking binary files using contextual information

Abstract

To unpack firmware files, disk images, raw flash dumps, file systems or other archives various tools are available, that examine the contents of the file, find offsets of archives, compressed files, media files and so on, carve these from a larger file, decompress the carved files, and make the unpacked data available for recursive unpacking.

Currently available tools treat all found files of a certain type the same (all PNG files are treated the same, all ZIP files are treated the same and so on), without taking the context in which they were found into account, which actually could matter depending on the situation.

This document describes possible approaches to this problem, where contextual information from unpacking is made available to allow for more accurate unpacking and labeling of files.

Keywords

firmware, reverse engineering, binary analysis, carving

Background

For fields such as forensics (finding presence of files or finding evidence), security analysis (firmware analysis), or reverse engineering it is important to be able to unpack archives and firmwares to be able to analyze the contents of these archives and firmware files.

With firmware files becoming larger and larger (300,000 files in mobile phone firmware images is not an exception) it becomes desirable to be able to focus on the most interesting files. What is interesting differs per use case.

Certain carving tools, such as Binary Analysis Tool [1][2][4] and its successor Binary Analysis Next Generation[3] already label each file that was found: graphics files are labeled as 'graphics', audio files as 'audio', ZIP archives as 'compressed' and 'zip', ASCII files are labeled as 'text', and so on (see drawing 1).

Files can have multiple labels and labeling allows a user to filter and focus on the files found interesting in a very fine grained way.

However, not every file of a certain type is as interesting as every other file of that type, but it depends on where in the archive it is found. This might even apply to files that are identical: a file might be very interesting when found in a certain context, but completely irrelevant when found somewhere else.

Consider the following situation: a firmware update file for a router running Linux contains a special version of Linux and associated programs that are only used during update of the firmware, but not

during normal operation. When analyzing the software that is used on the router during normal operations the software used during the update is irrelevant and should be ignored. To make this possible the labeled should be tagged as such, so it can quickly be discarded during analysis.

Method

Drawing 1 graphically shows how a straightforward unpacker (such as current tools[1][2][3][4]) would recursively unpack files:

1. get a file from a scanning queue (has been put in there by another process)
2. start searching from offset 0 for identifiers
3. for the first identifier found try to unpack the file. If not continue with the next identifier.
4. add the results of unpacking to the queue and continue with the next identifier.
5. end if no more identifiers were found (example: end of the file)

In this process for each file labels are kept that describe the file (depending on success, whether or not the entire file was unpacked, and so on).

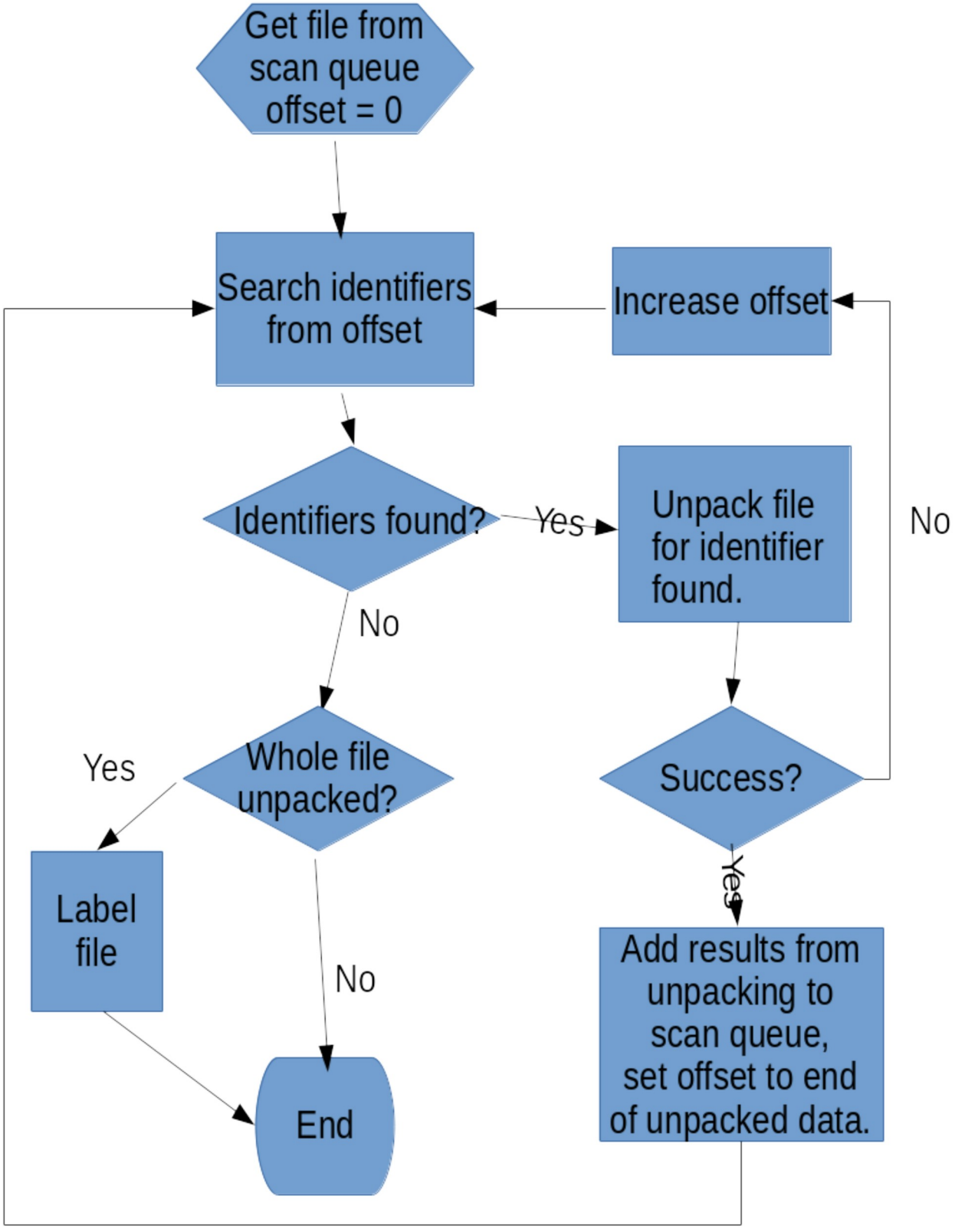
The new method differs in how the labeling works. In the scanning queue each file gets some extra metadata, such as:

- file from which the file was unpacked (the “parent file”)
- any labels from the parent file

Based on the labels from the parent file different decisions for labeling can be made.

Examples:

1. In Android files there are typically many files with resources, the so called “Chrome PAK” files[5]. These files contain text, images and other metadata. The text files can be mistaken for base64 encoded data, even though this is extremely unlikely. By knowing that the files were unpacked from a Chrome PAK file the unpacking method for base64 files can be skipped, reducing false positives.
2. Many embedded Linux systems use the opkg package manager[6][7]. Inside the files used by this package manager there are several control files for metadata. Since it is based on a format also used by Debian GNU/Linux there are some files that are very similar or identical to the files used by Debian GNU/Linux. By knowing that the file comes from opkg these files can be labeled correctly.



Drawing 1: Unpacking of a file

References

[1] Finding Software License Violations Through Binary Code Clone Detection, IP.com Disclosure Number: IPCOM000214472D, https://priorart.ip.com/IPCOM/000214472

[2] Hemel, Armijn & Trygve Kalleberg, Karl & Vermaas, Rob & Dolstra, Eelco. (2011). Finding software license violations through binary code clone detection. Proceedings - International Conference on Software Engineering. 63-72. 10.1145/1985441.1985453.

[3] Binary Analysis Next Generation, https://github.com/armijnhemel/binaryanalysis-ng

[4] Introducing the Binary Analysis Tool, http://web.archive.org/web/20190128182710/https://events.static.linuxfound.org/sites/events/files/als13_hemel_bat.pdf

[5] Description of Chrome PAK files (version 4), http://dev.chromium.org/developers/design-documents/linuxresourcesandlocalizedstrings

[6] https://git.yoctoproject.org/cgit/cgit.cgi/opkg/about/

[7] https://en.wikipedia.org/wiki/Opkg