Using the Binary Analysis Tool - part 2

Subjects

In this course you will learn:

- to determine the type of a file
- to find dependencies of dynamically linked binary files (ELF format)
- to extract human readable strings from binary files
- to extract function names from dynamically linked ELF files
- how these methods are implemented in the Binary Analysis Tool

Examining individual files

Possible types of analysis that could be performed on an individual file:

- file type
- dynamically linked libraries
- extracting human readable strings
- match strings extracted from binaries with source code
- extracting function names from ELF executables
- match function names extracted from binaries with source code

Interesting files for compliance analysis

There are various types of files that are interesting in for license compliance:

- executables (ELF, bFLT, Java, Windows executables)
- libraries
- Linux kernel images

Other files, such as audio, video or graphics files, can be interesting too for trademarks, or licensing (Creative Commons), but are outside of the scope of this course.

Determining file types

You can make a good first guess with the `file` command, which uses `libmagic`.

`libmagic` uses the so called "magic database" which contains descriptions for many sorts of files. It is usually found in `/usr/share/magic` on Linux systems.

By looking at a significant number of bytes and it can make an educated guess which type a file has, since many files have unique identifiers or characteristics.

Excercise: determine the type of files

Use the `file` command to determine the type of various files on your system.

Drawbacks of using `file` and `libmagic`

There are some drawbacks to using commands like `file` or tools that use `libmagic`:

- the descriptions in the magic database can and will change over time and sometimes contains errors.
- the magic database is not complete.
- some file types don't have a fixed magic type.
- not the entire file is considered, but just a number of bytes (up to a few hundred), so false positives can happen, especially if extra data has been appended to the file (this happens frequently with firmware images).

File type verification in BAT

In BAT data from `libmagic` is used for reporting, but not for searching and unpacking. It is used sometimes as a verification step.

Instead, a more *conservative* approach is used:

- BAT uses standard tools to verify, even though this sometimes costs more time. For example `gifinfo` is used to verify GIF files. Compared to `file` these tools often actually do take the whole file into account.
- BAT errs on the safe side: if not sure (or sure enough) that a file is of a certain type then it is treated as an unknown file type.

Inspecting dynamically linked ELF files

In case the file is an ELF executable (executable program) or shared object (library for dynamic linking) it is important to find the run time dependencies (also called *shared libraries*) because linking creates a *derivative work* which is very significant for fullfilling licensing conditions.

ELF dynamic linking 101

ELF executables can be linked in two ways:

- static linking: all dependencies (C library, other libraries) are included into the final executable.
- dynamic linking: dependencies are recorded at build time, but only resolved when the program is run (run time)

For license requirements it is very significant to see what a program links to (either dynamically or statically). If a program is dynamically linked you need to inspect the file to find the dependencies.

Using `readelf` to display shared libraries

The best tool to display shared libraries is to use the `readelf`, which is part of GNU binutils.

The command:

`readelf -d | grep NEEDED`

will display the shared libraries that are used, for example (output formatted using `cut`):

```
$ readelf -d /bin/ls | grep NEEDED | cut -f 4- -d " "
            Shared library: [libselinux.so.1]
            Shared library: [librt.so.1]
            Shared library: [libcap.so.2]
            Shared library: [libacl.so.1]
            Shared library: [libc.so.6]
```

Why not use `ldd`?

One reason to not use `ldd` is that `ldd` uses the dynamic linker on the *local* system to find dependencies.

It might seem that an executable or library is linked with a certain library, while in fact it is not and just a dependency on the *local* system.

Also, it will not work for executables compiled for different architectures.

Excercise: find the dynamically linked libraries of a file

1. Look at several dynamically linked executables on your system with `readelf`. Also look at them with `ldd` and try to spot differences.
2. repeat, but with files from the test firmware(s)

Extracting human readable strings from a file

With the `strings` program it is possible to extract sequences of human readable text (in the ASCII range) from a file (by default sequences of length 4 or greater):

```
strings /path/to/file
```

These sequences are (possibly) string constants that are not stripped by the compiler:

- output messages
- error messages
- debug messages
- etcetera

If extracted strings can be matched to source code they can provide a fingerprint of what source code was used.

Excercise: extracting strings from a file using `strings`

Examine a few executables on your system by running `strings` on it:

```
strings /path/to/file
```

It is useful to play with the size parameter (`-n`) and see how output will differ: smaller sequences will give more output, but also have significantly more noise. Example to extract sequences of size 6:

```
strings -n 6 /path/to/file
```

Matching strings from binaries with source code

After extracting strings you need to find out where they come from. Good sources are:

- search engines
- database containing strings from source code

BAT has mechanisms available to use databases and there are companies that offer pregenerated databases.

Unique matches

If a string can only be found in one package (counting different versions, or rebranded packages, as one package) it is a *unique match*.

If there are many unique matches, it is a good sign that a certain piece of software was used in the binary.

However: you should never base a conclusion on just one unique match.

Non-unique matches

There are several reasons why strings might appear in multiple programs:

- massive reuse of (open source) code and copying of code between programs ("cloning"): good examples are `zlib` and `libjpeg`
- protocols and standards (POSIX, RFCs, W3C, C, etcetera)
- coding styles (GNU, others)

String extraction in the Binary Analysis Tool

In the Binary Analysis Tool only strings with length of 5 or longer are considered.

Two ways to compare strings are implemented:

1. checks with a few hardcoded strings (that are most likely unique for some packages)
2. advanced check using database backend (needs database)

The first method is very quick, the second method is much more accurate, but requires a lot of data. Generating a database is covered in a later course and scripts to generate databases are freely available.

Extracting function names from a dynamically linked ELF file

Another way to fingerprint a binary is to extract *function names* from the binary and compare them to function names extracted from source code.

Because function names need to be resolved during runtime ("dynamic linking") they need to be declared so the dynamic linker knows which function names to find where. Therefore all function names are found in a table with function names.

Local function names (that have associated code in the binary) have an address associated with them, while remote functions are declared "undefined". It is very easy to extract local functions.

Excercise: extract function names from a dynamically linked ELF file

Extract function names from several dynamically linked ELF files on your system using the following command:

```
readelf --dyn-syms /path/to/file | grep FUNC | grep -v UND | tr -s " " | cut -f 9 -d " "
```

Matching function names from binaries with source code

After extracting function names you need to find out where they come from, just like with strings. Good sources are:

- search engines
- database containing function names extracted from source code

BAT has mechanisms available to use databases and there are companies that offer pregenerated databases.

Function name extraction in the Binary Analysis Tool

Function names are extracted from source code using `ctags` and stored in a database.

During analysis of the binary the right ELF section is extracted using `readelf`, similar to shown before. Function names that come from C++ are *demangled* first, using `c++filt` to restore function names.

Then function names are matched with the database. Currently only unique function names are used in reporting.

Circumventing detection techniques

The techniques mentioned above are quite easy to beat. String comparisons can be beaten by changing strings. Function name comparisons can be beaten by changing function names.

Trying to foil detection is not hard, but it is costly:

- extra costs replacing function names and strings (and maintaining compatibility)
- maintenance costs

plus these are not the only possible detection methods that could be used in the future.

From a compliance enforcement point of view the idea of these techniques is to make circumvention more expensive than complying with the licenses in the first place.

Conclusion

In this course you have learned about:

- determining the type of a file
- determining names of dynamically linked libraries
- extracting human readable strings from binary files
- extracting function names from dynamically linked ELF executables
- how the above are implemented in the Binary Analysis Tool

In the next course we will dig into how the Binary Analysis Tool can be configured and how results can be read.