Demystifiying ELF

Armijn Hemel, MSc - Tjaldur Software Goverance Solutions

What is ELF?

Executable and Linkable Format

- origin in System V Unix
- default format for executables on Linux since mid-1990s
  - but used as a container format by some (example: Android Dex)
- allows both static and dynamic linking

Why analyse ELF files?

two main use cases for ELF analysis:

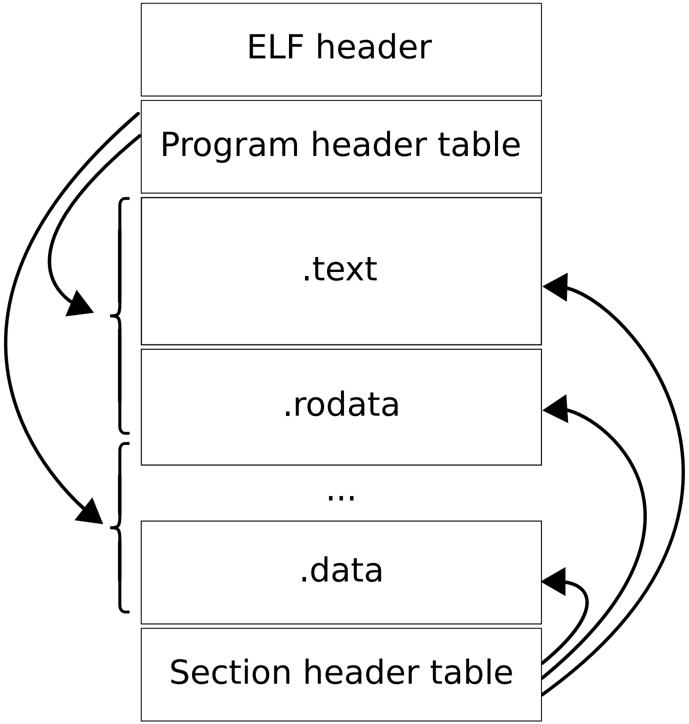
- provenance: where did this binary file come from?
- linking: what is the interaction with other components?

- interestingly, roughly the same information can be used for this!
- a full analysis walkthrough could take weeks. This will be the very quick condensed version.

ELF file format structure

- header, followed by:
  - (optional) program headers
  - segments/sections
  - (optional) section headers
- just an empty header would be a valid ELF file (but not very useful)

ELF in a picture (from Wikipedia)



Program headers vs section headers

ELF provides two ways to access data:

- program headers point to segments
- section headers point to sections

- when executing the operating system (example: Linux) only uses the program headers
- for analysis the sections are much more useful
- fiddling with program headers/section headers is sometimes used as a form of obfuscation and sometimes won't match!

Accessing section information

many tools available for accessing section information:

- readelf (GNU binutils)
- pyelftools
- Kaitai Struct

- in this talk readelf is used
- readelf does a lot of the heavy lifting

readelf: useful switches

-W : wide output

- -h : display header
- -S : display sections
- -s : display symbols
- -a : display everything
- -p : display strings of a specific section

Demo: poking around a few files

let's look at a few files using readelf
  - /bin/ls
  - /bin/vim
- and look at a few things:
  - ELF header
  - section information
  - symbols

Section names

- ELF specification reserves several section names
  - .interp
  - .dynsym
  - .shstrtab
  - .dynamic
  - etcetera
- developers are free to add other sections with non-reserved names

Section types

Sections also have a type:

- NOTE : ELF note sections
- STRTAB : string sections
- VERSYM / VERNEED : symbol versioning information
- PROGBITS : program specific information (not defined by the specification)
- and so on

Interesting NOTE sections

(recent) Fedora stores provenance

- .note.package contains JSON with package information:
- Example:

  {"type":"rpm","name":"vim","version":"9.1.825-1.fc39","architecture":"x86_64","osCpe":"cpe:/o:fedoraproject:fedora:39"}

- this is a standard from systemd:
  - https://systemd.io/ELF_PACKAGE_METADATA/

Interesting PROGBITS sections (1)

Example: Qt

```
$ readelf -WS /usr/lib64/libQt5Core.so.5.15.14 | grep qt
  [19] .qtmimedatabase   PROGBITS        [...]
```

- this section contains a compressed copy of the MIME database. Old versions of Qt use the (GPL licensed) MIME database from freedesktop.org

Interesting PROGBITS sections (2)

Sony ESSTRA is a new project

- **E**nhancing **S**oftware **S**upply Chain **Tra**nsparency
- GCC plugin that stores names of used files as YAML
- YAML stored in an ELF section
- new project, still under development
- https://github.com/sony/esstra/

Interesting PROGBITS sections (3)

Go stores a ton of information in ELF sections

- line table, mapping to source code files at line number
- symbol table
- etc.

Crazy ELF stuff

Android: uses ELF to wrap Dex bytecode

- AppImage version 2 puts its own magic bytes in the ELF header overwriting standard headers
- Go is funky:
  - breaks ASCII requirements for symbols, by using UTF-8 characters
- ELF specifications should sometimes be treated as a “suggestion”

Detecting provenance in more detail

information already available:

- file names (tend to be mostly unique, as most Linux systems are a single namespace)
- several sections (see before)

- but this is not granular enough. Instead use:
  - symbols (functions, variables, etc.)
  - strings

Why use symbols?

symbols are unique enough

- (unscientific) research:
  - download recent Debian
  - look at (defined) functions and variables in **all** ELF files
- result:
  - vast majority of symbols are unique to a single package
  - often unique to a single **file**
  - Linux tends to be a single namespace because ELF linking is based on symbols

Why use strings?

many strings survive compilation and stripping:

- debug strings
- output strings

- platform agnostic
- very specific to programs
  - strings present the Linux kernel is very unlikely to end up in desktop applications, except for things like shared dependencies like compression

Simple method for fingerprinting

1. extract symbols and strings from source code
2. extract symbols and strings from binaries
3. match!

Extracting symbols and strings from source code

no fancy tools or parsers needed!

- for symbols:
  - **ctags**
  - **pygmars**
- for strings:
  - **xgettext**

Extracting symbols from ELF binaries (1)

process output from **readelf**

- parse ELF file:
  - **pyelftools**
  - Kaitai Struct
- **readelf** works well enough for small manual inspection, other methods for programmatically processing large amounts of files
- interesting from the symbol table: **FUNC** and **OBJECT**

Extracting symbols from ELF binaries (2)

```
$ readelf -Ws /bin/bash | egrep -e 'FUNC|OBJECT' | grep -v UND | head -n 4
   234: 00000000000ebd10  1029 FUNC    GLOBAL DEFAULT   16 rl_old_menu_complete
   235: 000000000005d110    25 FUNC    GLOBAL DEFAULT   16 maybe_make_export_env
   236: 00000000000a5100    35 FUNC    GLOBAL DEFAULT   16 initialize_shell_builtins
   237: 00000000000d14a0    39 FUNC    GLOBAL DEFAULT   16 extglob_pattern_p
```

Extracting symbols from ELF binaries (3)

If you are lucky:

- unstripped binaries can contain even more information
  - debug symbols
  - file names

Extracting strings from ELF files

strings (GNU binutils) works well, but:

- for best success limit it to specific **PROGBITS** sections
  - .rodata, .rodata.str1.1, .rodata.str1.4, .rodata.str1.8, etc.
- no guarantee for success, as strings could instead be stored as a trie!
- **strings** will not (by default) catch “wide” strings
- still: this is good enough!

Fingerprinting caveats

Fingerprinting doesn't always work or catch all data:

- there has to be enough unique data to work with!
- statically linked ELF files, with no unique strings, are a challenge
- should be used as a starting point for further research, not a definitive answer

Tool support for fingerprinting

Binary Analysis Next Generation (BANG):

- main analysis program extracts data from binaries
- helper scripts to:
  - extract data from source code
  - generate YARA scripts from data extracted from source code and binaries
- soon: integration with AboutCode's “purl2sym” service

ELF dynamic linking analysis

- knowing how ELF files interact is important:
  - GPL & LGPL compliance
  - “derivative works”
- a tool like **ldd** can help, but:
  - uses the dynamic linker configuration of the host system
  - doesn't allow search
- so, let's look at a better method

ELF dynamic linker

- kernel loads ELF file and looks for the **interpreter** (dynamic linker):

```
$ readelf -Wa /bin/ls | grep interpreter
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

- dynamic linker loads configuration (typically **/etc/ld.so.conf** and friends) to find library search path
- ELF file is queried for declared shared libraries that are needed, recursively
- symbols that are needed are searched in the shared libraries and resolved, recursively

ELF dependencies

$ readelf -Wa /bin/ls | grep NEEDED

0x0000000000000001 (NEEDED) Shared library: [**libselinux.so.1**]

0x0000000000000001 (NEEDED) Shared library: [**libcap.so.2**]

0x0000000000000001 (NEEDED) Shared library: [**libc.so.6**]

Finding ELF symbols

```
$ readelf -Ws /bin/ls | egrep -e "FUNC|OBJECT" | grep UND | head -n 4
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __ctype_toupper_loc@GLIBC_2.3 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND getenv@GLIBC_2.2.5 (3)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND cap_to_text
     4: 0000000000000000     0 OBJECT  GLOBAL DEFAULT  UND __progname@GLIBC_2.2.5 (3)
```

ELF symbol versioning

- some ELF libraries provide versioning:
  - attempt at API
  - can be used for stricter fingerprinting and linking analysis
- not universally implemented, but some important libraries use it:
  - glibc
  - Qt
  - ALSA
  - PAM

ELF linking graph

- result is ELF linking graph that could be used for:
  - fine grained querying of symbols
  - visual representation of link dependencies
  - possibly detect cruft
- tool support:
  - callgraph (next talk!)
  - BANG (in progress)
  - elfcall

Resolving symbols, recursively

1. extract symbols that need to be resolved from a binary
2. find libraries that have been defined as a dependency
3. for each library:
   a. search for any unresolved symbols from step 1 to see if the library defines this symbol and record as “resolved”
   b. go to step 1

More information + acknowledgments

https://formats.kaitai.io/elf/

- https://refspecs.linuxfoundation.org/elf/
- https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- ELF picture by Santiago Urueña Pascual released under CC-BY-SA 3.0