Executable and Linkable Format 101. Part 2: Symbols

By Ignacio Sanmillan

Published: 2018-02-07 · Archived: 2026-04-05 19:35:24 UTC

In our previous post, we focused on understanding the relationship between sections and segments, which serve as the foundation for understanding the *ELF* file format. However, we will soon discover that we have ignored some degree of detail for the sake of simplicity.

In the next couple of posts, we will focus on explaining the details behind *Symbols* and *Relocations,* which play a fundamental role in understanding the *ELF* file format in greater depth.

Defining Symbols

In the development process of a program, we often use names to refer toa objects in our code, such as functions or variables that are used all throughout the program. This information is what is known as the program's symbolic information. Symbolic references in machine code get translated into offsets and addresses on the compilation process.

However, compilers are not limited to machine code generation; they also export symbolic information from source code. Symbolic information is exported in order to improve the interpretation of the generated machine code. The *ELF* file format has a means to hold this symbolic information and provides an interface for local and external *ELF* objects to access it.

A wide range of tools interact with symbols to enforce their functionality. One example of these tools are *Linkers*. *Linkers* usually interact with a *Symbol Table* in order to match/reference/modify a given symbol inside an *ELF* object at linktime. Without *Symbols*, *Linkers* would not work, since they won't know which relocations must take place, and therefore would not have a mechanism to match a given symbolic reference to its corresponding value.

Another example of tools that heavily depend on symbolic information are *Debuggers*. Without *Symbols,* the process of *Debugging* can become considerably harder or even worthless, especially for projects with massive code bases such as the *Linux Kernel*. Debugging without symbols implies that one has no ability to identify functions or variables by name among other impediments. This would considerably reduce the features afforded by debuggers.

Interestingly, Malware also interacts with symbol tables as an anti-analysis mechanism. Symbol Tables can appear to be completely removed from the binary as an attempt to harden the analysis process, or in other cases their String Table entries may be swapped so that symbols will be miss-interpreted. (We'll cover malware anti-analysis techniques in coming posts.)

ELF Symbol structure

In the ELF file format, each *Symbol* is represented as an instance of an *Elfxx_Sym* structure inside a given *Symbol Table*.

```
struct Elf64_Sym{
    Elf64_Word      st_name;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf64_Half      st_shndx;
    Elf64_Addr      st_value;
    Elf64_Xword     st_size;
};
```

The Elements of this structure include:

- st_name: index in string table of symbol name. If this field is not initialized, then the symbol doesn't have a name
- st_info: contains symbol bind and type attributes. Binding attributes determine the linkage visibility and behavior when a given symbol is referenced by an external object. The most common symbol binds are the following:
    - STB_LOCAL: Symbol is not visible outside the *ELF* object containing the symbol definition.
    - STB_GLOBAL: Symbol is visible to all object files.
    - STB_WEAK: Representing global symbols, but their definition can be overridden.

The most common symbol types are the following:

-
    - STT_NOTYPE: symbol type is not specified
    - STT_OBJECT: symbol is a data object (variable).
    - STT_FUNC: symbol is a code object (function).
    - STT_SECTION: symbol is a section.

In order to retrieve these fields, a set of bitmasks are used. These bitmasks include:

-
    - ELF64_ST_BIND(info) ((info) >> 4)
    - ELF64_ST_TYPE(info) ((info) & 0xf)
- st_other: information about symbol visibility. Symbol visibility defines how a given symbol may be accessed once the symbol has become part of an executable or shared object. Some common symbol visibilities are the following:
    - STV_DEFAULT: for default visibility symbols, its attribute is specified by the symbol's binding type.
    - STV_PROTECTED: symbol is visible by other objects, but cannot be preempted.
    - STV_HIDDEN: symbol is not visible to other objects.

    - STV_INTERNAL: symbol visibility is reserved.

The major difference between symbol visibility and symbol binding is that the former is enforced on the host object of a given symbol and applies to any external object referencing it. Symbol binding, on the other hand, is specified by an external object referencing the symbol and is enforced at link-time. In other words, object binding depends on the object referencing the symbol, while symbol visibility depends on how the object containing the symbol was compiled.

The only relevant contents of this field are its three last significant bits. Other bit fields do not contain specific meaning. In order to retrieve visibility information, the following macro is used:

- - ELF64_ST_VISIBILITY(o) ((o) & 0x3)

- st_shndx: Every symbol entry within the *Symbol Table* is associated with a section. This value specifies the section index within the *Section Header Table* associated with the given symbol. Common values for section type field include:
    - SHT_UNDEF: section is not present in current object. This value is typically set in symbols that have been imported from external objects.
    - SHT_PROGBITS: section is defined in current object.
    - SHT_SYMTAB, SHT_DYNSYM: Symbol Table (*.symtab, .dynsym*).
    - SHT_STRTAB, SHT_DYNSTR: String Table (*.strtab, .dynstr*).
    - SHT_REL: Relocation Table without explicit addends (*.rel.dyn, .rel.plt*).
    - SHT_RELA: Relocation Table with explicit addends (*.rela.dyn, .rela.plt*).
    - SHT_HASH: Hash Table for dynamic symbol resolution (*.gnu.hash*)
    - SHT_DYNAMIC: section holding *Dynamic Linking* information (*.dynamic*)
    - SHT_NOBITS: section takes no space in disk (*.bss*).

- st_value: This field specifies the symbol value for a given *Symbol Table entry*. The interpretation of this field can vary depending on the object type.
    - For *ET_REL* files *st_value* holds a section offset. The section in which this offset resides is specified on its st_shndx field.
    - For *ET_EXEC* / *ET_DYN* files, *st_value* holds a virtual address. If this field contains a value of 0 and the symbol's section pointed by *st_shndx* has a *sh_type* field of type *SHT_UNDEF*, the symbol is an imported relocation, and its value will be resolved at runtime by the *RTLD* (*ld.so*).

- st_size: Field containing symbol's size.

Symbol and String Tables

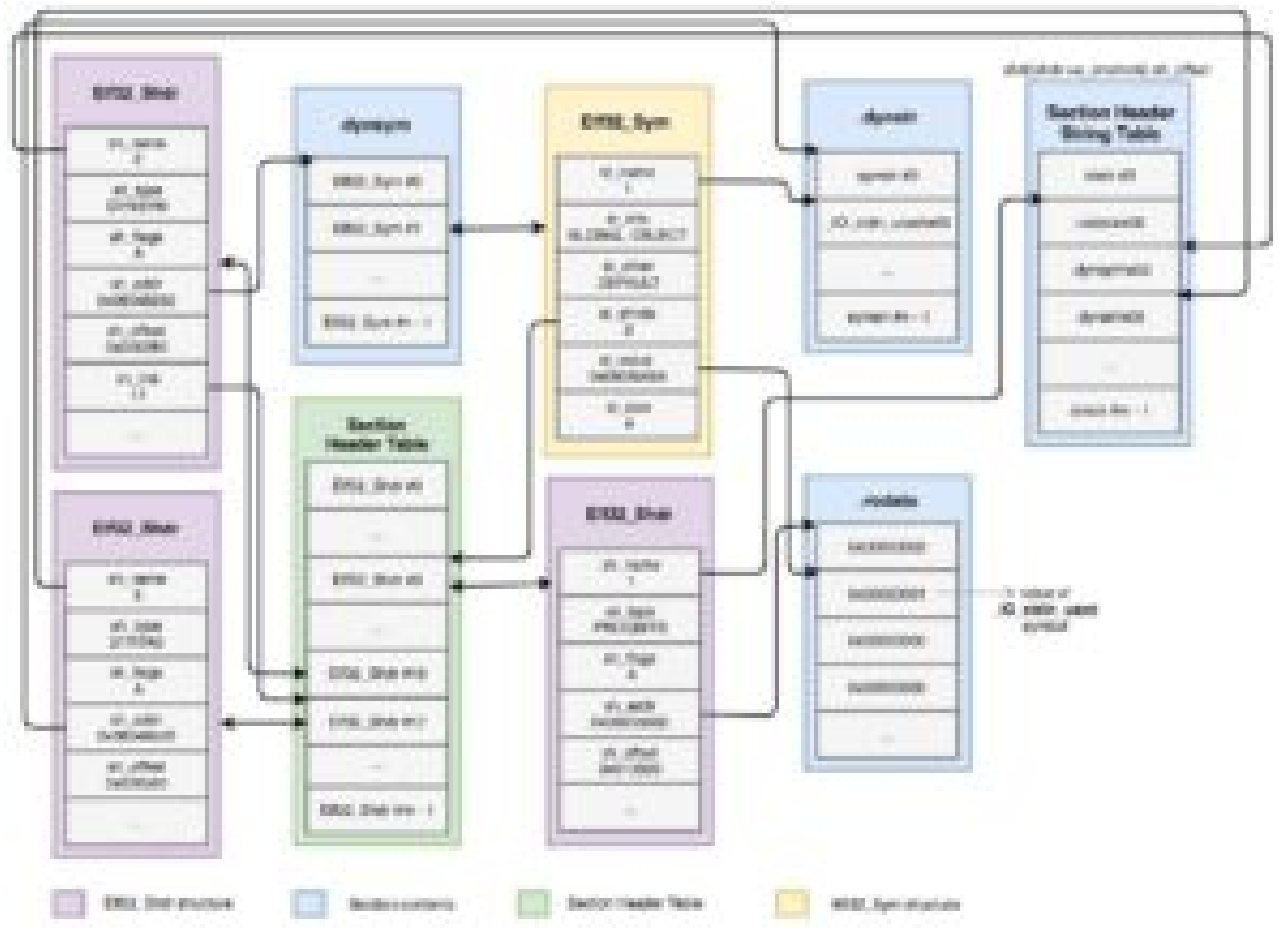
A single *ELF* object may contain a maximum of two *Symbol Tables*. These are *.symtab* and *.dynsym*. The difference between these two instances involves the number and type of symbols they contain. We refer *.symtab* as the binary's global *Symbol Table*, containing all symbol references in the current object. The section *.strtab* is the *String Table* of *.symtab Symbol Table*. String Tables store null-terminated strings used to reference objects from a different section. Each String Table contains the exact number of entries as its corresponding Symbol Table. This

entails that each string entry at a given index in *.strtab* corresponds to an *Elfxx_Sym* entry at the same index in *.symtab*.

On the other hand, we have *.dynsym Symbol Table*. This *Symbol Table* only holds symbols needed for *Dynamic Linking*. (We'll cover *Dynamic Linking* extensively in future posts.) As an overview, when developing an application, sometimes we'd want to use symbols that don't reside within the context of our program but are instead defined in external objects such as libraries (*Shared Objects*). *Dynamic Linking* is the process where the linker tries to dynamically bind those external symbols at runtime in order for them to be referenced safely within the context of a program.

If a given binary has been stripped (.symtab/.strtab have been removed) and this same binary has been compiled so that a subset of its symbols will be dynamically linked, this subset of symbols can be recovered by parsing *.dynsym* table located at *DT_SYMTAB* entry within *PT_DYNAMIC* segment. For dynamically linked executables, *.dynsym* will not be removed even if target binary has been stripped, since it is needed at runtime by the *RTLD* in the process of *Dynamic Linking*.

As with *.symtab*, *.dynsym* has its own string table called *.dynstr*. All the relationships previously covered between *.symtab* and *.strtab* also apply between *.dynsym* and *.dynstr*. The following diagram illustrates the concepts of Symbols and String tables that we've just discussed:



In this diagram, we see an example representation of various data structures from a binary containing the symbol *_IO_stdin_used*. This symbol is represented as the second *Elf32_Sym* instance in *.dynsym Symbol Table*. Note that

.dynsym is a section represented as an *Elf32_Shdr* structure, and the index of its *String Table* within the *Section Header Table* can be retrieved by its *Elf32_Shdr's sh_link* field.

Moreover, This symbol resides within the *.rodata* section as shown by its *Elf32_Sym st_shndx* field, which denotes the fourth *Elf32_Shdr* instance within the *Section Header Table*. This *Elf32_Shdr* instance's *sh_name* field points to the second null terminated string within the Section Header String Table, which is the '*.rodata*' string.

Furthermore, we can obtain *_IO_stdin_used*'s attributes based on its *Elf32_Sym* instance. We can see that it is of type *OBJECT*, and its binding is of type *GLOBAL*. Therefore, we can assume this symbol is a Global variable. Note that the *st_value* field may lead to misconceptions since it contains a virtual address denoting the symbol's location within its correspondent section, but not the actual value of the Symbol.

Summary

We've covered the intricacies of Symbol handling within the *ELF* file format. We now have an understanding of what symbols are and their different uses. We have also covered the different types of Symbol Tables and how they correlate to different String Tables. In the next post, we will dive into ELF Relocations and the Dynamic Linking process.
