ELF Format

В Linux, BSD и других Unix-like системах ELF формат используется для исполняемых файлов, shared libraries, объектных файлов, coredump'ов и даже для загрузочного образа ядра. Все это делает ELF-формат важным для изучения теми, кто хочет лучше понимать reverse engineering, binary hacking и исполнение программ. Вообще, разобраться с бинарными форматами ,такими как ELF, не так уж и просто и вообще может показаться сухим и нудным занятием, но если сочетать его изучение с практикой ,то все окажется иначе.

В этом посте я постараюсь описать следующие темы:

- Типы файлов ELF
- Заголовки программы
- Секция заголовков
- Символы
- Релокации
- Динамическая линковка
- Написание ELF-парсера

Типы файлов ELF

- ET_NONE: неизвестный тип. Говорит о том, что тип файла неизвестен или не был еще определен.
- ET_REL: настраиваемый файл. По сути, просто

объектный файл ,который не слинкован. Обычно такие файлы являются Position independent code (PIC), которые еще не были слинкованы в исполняемый файл. После компиляции кода можно часто увидеть .o файлы.

- ET_EXEC: исполняемый файл. исполняемый ELF является начальной точкой в том как стартует процесс.
- ET_DYN: shared objects. Означает, что файл помечен как динамически подключаемый ,также известный как shared libraries. Эти библиотеки линкуются в образ процесса во время его работы.
- ET_CORE: ядро типа ELF, которое помечает core-file. Core file - дамп процесса во время падения программы или когда был получен SIGSEGV сигнал(ошибка сегментации). Gdb умеет читать подобные файлы, которые можно "вылечить" дебаггингом и понять что вызвало падение.

Теперь посмотрим на ELF файл с помощью readelf -h, где мы увидим init хедер файла. Данный хедер показывает тип ELF файла, адрес входа ,где начинается исполнение ,и архитектуру, а также предоставляет смещения на другие типы хедеров. Большинство из них будет освещено позднее и большая часть из низ станет понятна после после того, как мы рассмотрим значение хедера секций и хедера программы.

Если посмотреть в man на 5ю страницу по формату ELF, то мы увидим структуру ELF init хедера:

```
#define EI_NIDENT 16
typedef struct {
  unsigned char e_ident[EI_NIDENT];
```

```
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff; // init offset
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint32_t      e_ehsize;
    uint32_t      e_phentsize;
    uint32_t      e_phnum;
    uint32_t      e_shentsize;
    uint32_t      e_shnum;
    uint32_t      e_shstrndx;
} ElfN_Ehdr;
```

Javascript

ELF-file хедеры

Хедеры необходимы для правильного описания сегментов внутри бинарника и правильной загрузки программы. Сегменты интерпретируются ядром во время загрузки и описывают размещение исполняемого файла на диске и то ,как он должен быть оттранслирован в памяти. Доступ к хедеру может быть получен по смещению `e_phoff`, которое может быть найдено в initial ELF хедере, в структуре `ElfN_Ehdr`.

Всего выделяют 5 типов популярных хедеров, которые мы здесь обсудим. Программные хедеры описывают сегменты исполняемого файла(включая shared libraries) и типы этих сегментов(что за тип данных или кода). Прежде всего давайте взглянем на структуру `Elf32_Phdr` ,которая является входным хедером в таблице program хедера 32битного ELF executable.

```
typedef struct {
    uint32_t p_type; (segment type)
    Elf32_Off p_offset; (segment offset)
    Elf32_Addr p_vaddr; (segment virtual address)
```

```
    Elf32_Addr p_paddr; (segment physical address)
    uint32_t p_filesz; (size of segment in memory)
    uint32_t memsz; (size of segment in memory)
    uint32_t p_flags; (segment flags, I.E
execute|read|read)
    uint32_t p_align; (segment align in memory)
} Elf32_Phdr;
```

Go

PT_LOAD

Исполняемый файл будет всегда иметь хотя бы один сегмент типа `PT_LOAD`. Этот тип хедера описывает загружаемый сегмент, что означает ,что сегмент будет загружен или отображен в памяти.

Для примера, ELF executable с динамической линковкой будет содержать два сегмента `PT_LOAD`:

- text segment
- data segment

Эти два сегмента будут отображены в памяти и выравненны значением находящимся в `p_align`.

---

text segment, также известный как сегмент кода, будет обычно иметь такие права как `PF_X | PF_R` (READ+EXECUTE).

data segment обычно имеет права сегмента выставленные в `PF_W | PF_R` (WRITE+READ).

Зараженный файл полиморфным вирусом возможно изменил эти права таким образом, что text segment стал редактируемым, добавив `PF_W` флаг в сегмент флагов внутри хедера файла (`p_flags`).

PT_DYNAMIC - Phdr для динамического сегмента

Динамический сегмент присущ исполняемым файлам ,которые динамически слинкованы и содержат информацию для динамического линковщика, примерно следующего характера:

- Список shared libraries, которые будут слинкованы во время исполнения
- Адрес Глобальной таблицы смещений (Global offset table [GOT])
- Информация о записях перемещений

Вот полный список тегов:

Tag name	Description
DT_HASH	Address of symbol hash table
DT_STRTAB	Address of string table
DT_SYMTAB	Address of symbol table
DT_RELA	Address of Rela relocs table
DT_RELASZ	Size in bytes of Rela table
DT_RELAENT	Size in bytes of a Rela table entry
DT_STRSZ	Size in bytes of string table
DT_STRSZ	Size in bytes of string table
DT_STRSZ	Size in bytes of string table

Tag name	Description
DT_SYMENT	Size in bytes of a symbol table entry
DT_INIT	Address of the initialization function
DT_FINI	Address of the termination function
DT_SONAME	String table offset to name of shared object
DT_RPATH	String table offset to library search path
DT_SYMBOLIC	Alert linker to search this shared object before the executable for symbols
DT_REL	Address of Rel relocs table
DT_RELSZ	Size in bytes of Rel table
DT_RELENT	Size in bytes of a Rel table entry
DT_PLTREL	Type of reloc the PLT refers (Rela or Rel)
DT_DEBUG	Undefined use for debugging
DT_TEXTREL	Absence of this indicates that no relocs should apply to a nonwritable segment
DT_JMPREL	Address of reloc entries solely for the PLT
DT_BIND_NOW	Instructs the dynamic linker to process all relocs before transferring control to the executable
DT_RUNPATH	String table offset to library search path

Динамический сегмент содержит серию структур, которые хранят релевантную информацию о динамической линковке. `d_tag` контролирует интерпретацию `d_un`.

```
typedef struct {
    Elf32_Sword   d_tag;
        union  {
    Elf32_Word    d_val;
    Elf32_Addr    d_ptr;
    } d_un;
} Elf32_Dyn;
extern Elf32_Dyn _DYNAMIC[];
```

Go

О динамической линковке мы поговорим позже.

PT_NOTE

Сегмент типа `PT_NOTE` может содержать вспомогательную информацию специфичную для конкретного вендора.

Иногда вендору или производителю системы необходимо пометить объектный файл специальной информацией, которую остальные программы проверят на совместимость и т.д.
Интересный факт: учитывая тот факт, что этот сегмент используется только для спецификаций ОС и обычно не должен быть исполняемым, то этот сегмент становится привлекательным местом для вирусной инфекции...

PT_INTERP

Этот маленький сегмент содержит только место и размер строки\0 ,которая описывает где находится интерпретатор программы; для примера:`/lib/linux-ld.so.2` - обычно место, где хранится динамический линковщик, который также является программным интерпретатором.

PT_PHDR

Этот сегмент содержит локацию и размер самой таблицы заголовков. Phdr таблица содержит и описывает все сегменты файла.

Чтобы узнать подробнее о других типах Phdr, обратитесь к ELF(5) man'у. Мы покрыли основные и наиболее встречающиеся сегменты.

Можно использовать readelf -l <filename> ,чтобы посмотреть Phdr таблицу:

```
Entry point 0x8049a30
There are 9 program headers, starting at offset 52
Program Headers:
Type          Offset   VirtAddr   PhysAddr  FileSiz
MemSiz  Flg Align
PHDR          0x000034 0x08048034 0x08048034 0x00120
0x00120 R E  0x4
```

```
INTERP         0x000154 0x08048154 0x08048154 0x00013
0x00013 R      0x1
[Requesting program interpreter: /lib/ld-linux.so.2]
LOAD           0x000000 0x08048000 0x08048000 0x1622c
0x1622c R E    0x1000
LOAD           0x016ef8 0x0805fef8 0x0805fef8 0x003c8
0x00fe8 RW     0x1000
DYNAMIC        0x016f0c 0x0805ff0c 0x0805ff0c 0x000e0
0x000e0 RW     0x4
NOTE           0x000168 0x08048168 0x08048168 0x00044
0x00044 R      0x4
GNU_EH_FRAME 0x016104 0x0805e104 0x0805e104 0x0002c
0x0002c R      0x4
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000
0x00000 RW     0x4
GNU_RELRO      0x016ef8 0x0805fef8 0x0805fef8 0x00108
0x00108 R      0x1
```

Javascript

Мы видим точку входа исполняемого файла, а также разные сегменты ,в том числе и те, что мы обсудили выше. Обратите внимание на флаги и