ELF's Story Part1: How is an ELF create

aleeamini.com/elfs-story-part1/

ELF’s Story Part1: How is an ELF create

1- Introduction

Hello guys I’m back with a new series of blog posts.
Last year I focused on Linux binaries. I read a lot about ELF files, How they compile, and how load in memory and run. So I decided to write my experiences in some blog posts and named it ELF’s story. But this is not about ELFs in Lord of the Rings

This story is for those who know about Linux and C/C++ programming and Want to know how a code converts to a binary file and how works on a computer.

2- ELF’s Story

Now I want to tell you the story of our ELF. At the beginning, I wanna tell you how an ELF was born.
An ELF file is born in a UNIX Operation System, by a combination of a compiler and some codes together.
Let’s dive into the codes.

2-1 How is an ELF created?

When you write code in a native language, like C/C++ (I won’t talk about interpreter base languages like java/C# or python, because their functionality is very different with native languages and they never convert to an ELF), when you want to run it, you should compile it.

The compiler converts your code into a binary file, depending on your OS.
In this part, I want to tell more details about the compiling process.
OK, we have a simple C code like this:

```
/* code sample 1-1
myheader.h
*/

#define PI 3.14

int func_add(int a, int b)
{
  return a+b;
}
```

```
/*code sample 1-2
main.c
*/

#include <stdio.h>
#include “myheader.h”

int main()
{
  int a=0,b=0,sum=0;
  scanf(“%d”,&a);
  scanf(“%d”,&b);
  sum = func_add(a,b);
  printf(“the PI is: %f”, PI);
  printf(“the result is: %f”, sum*PI);
  return 0;
}
```

Now we want to compile it.
This code has some important sections: the header files. Like included files.
Also, we have a #define MACRO that is used to define a PI number and two functions.
If we compile this code with a compiler like GCC or any other compiler, in the end, we have a compiled binary file that can get executed in OS. But what happens in the process of compilation?

Compilation is the process of translating human-readable source codes like C/C++, into machine codes.

Compiling C code involves four phases. Other languages compilation is very similar to C but may have some additional phases.

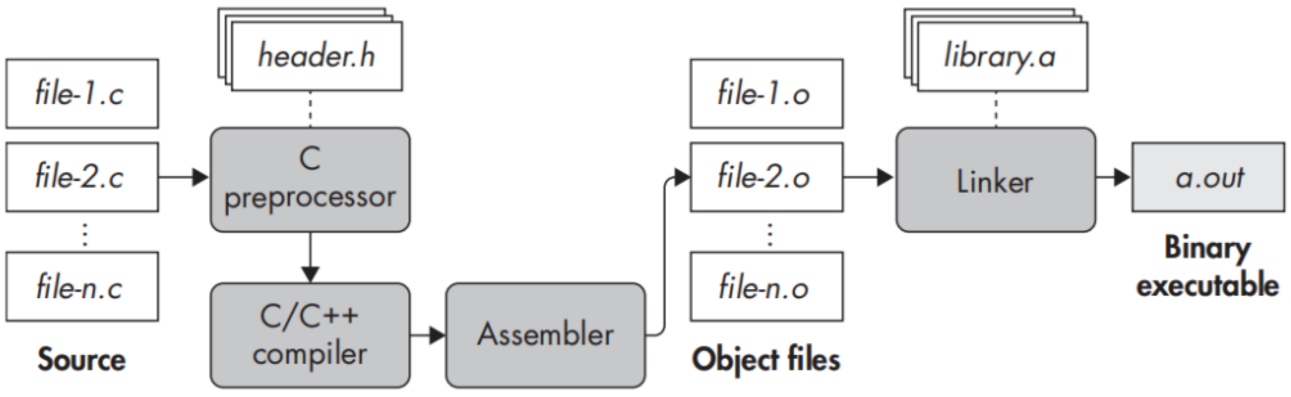


Figure 2-1: Compile Process

In the above figure, we can see the process of compilation.
In a real project, we may have multiple source files(.c/.cpp) and also multi-header files(.h/.hpp). but how the compiler finds out how to compile these files and generate an executable file at the end for us.

Let’s see the phases of the compilation process.

2-1-1 Pre-processing

In this phase, the compiler collects all .c/.cpp or sources files and merges them with their includes and places defined values where they had used. for example in our sample, we have a main.c and in it, we included two header files, stdio.h and myheader.h also I have a defined value PI number. So the compiler at the preprocessing phase reads the main.c file from top to down and replaces every includes, and definition, with their real sources. So after the preprocessing phase, we should have a file like this:

```
/*code sample 1-2
After preprocessing
*/

Functions and variables in stdio.h

#define PI 3.14

int func_add(int a, int b)
{
  return a+b;
}

int main()
{
  int a=0,b=0,sum=0;
  scanf("%d",&a);
  scanf("%d",&b);
  sum = func_add(a,b);
  printf("the PI is: %f", 3.14);
  printf("the result is: %f", sum * 3.14);
  return 0;
}
```

After preprocessing, all contents of header files are merged into the main source file. For example, all contents of stdio.h and myheader.h are merged into main.c file. Additionally, the PI-defined macro is placed in locations where it is used (as highlighted in the 3.14 value).

To see the result in a real sample, GCC has a switch that can be used to stop the compiler after the preprocessing phase. The -E switch tells the GCC to stop after preprocessing, and the -P switch causes the compiler to omit debugging information, resulting in cleaner output.

```
$ gcc -E -P main.c
/*
I omit some of functions of stdio.h
*/
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
int func_add(int a, int b)
{
 return a+b;
}
int main()
{
  int a=0,b=0,sum=0;
  scanf("%d",&a);
  scanf("%d",&b);
  sum = func_add(a,b);
  printf("the 3.14 is: %f", 3.14);
  printf("the result is: %f", sum*3.14);
  return 0;
}
```

Alright, so as we can observe, the results are consistent with what I had previously informed you. Moving on, let's consider a hypothetical scenario where the compiler generates a new file for each .c/.cpp file and combines all the included files, sources, and macros.
This approach enables the compiler to efficiently manage the code for compilation and error-handling purposes.
As of now, we have a preprocessed code. Let's now proceed to discuss the subsequent phase.

2-1-2 Compilation

In this phase of the compile process, the compiler should convert the preprocessed code to a low-level language. This low-level language is assembly.
If you are familiar with optimization(-O option in GCC), I should tell you that the optimizations run at this phase.
The compiler converts all C/CPP codes to Assembly language text code. So at the end of this phase, we have an assembly code of preprocessed codes.
But a question here is why the compiler doesn't generate machine code directly.

The answer is that we are talking about a compiler, not a C compiler. In a C compiler yes this is possible to convert code to machine code. But we are talking about some languages (List, Go, C++,…) that everyone has a special structure and the compiler should parse this structure and then optimize it, and then convert it to machine language.
For example, in the C++ language, we have virtual functions.

These functions will be handled with a Vtable in assembly code. So the compiler should first convert the code to assembly and then create this vtable by parsing the whole C++ code. So converting code directly to machine code is time-consuming for the compiler. The compiler converts code to assembly to parse it easily and can do optimization on it.
OK, now how we can see the result?

Like the previous phase, the gcc hasan **-S** option that tells gcc to stop the compile process after the compilation phase. The result of this command is in a file with the same name as your source code with .s extension (here main.s). we can tell gcc to show the assembly code in Intel or AT&T syntax (-masm)

```
$ gcc -S -masm=intel main.c
$ cat main.s
        .text
        .globl  func_add
        .type   func_add, @function
func_add:
.LFB0:
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        mov     DWORD PTR -4[rbp], edi
        mov     DWORD PTR -8[rbp], esi
        mov     edx, DWORD PTR -4[rbp]
        mov     eax, DWORD PTR -8[rbp]
        add     eax, edx
        pop     rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   func_add, .-func_add
        .section        .rodata
.LC0:
        .string "%d"
 1 .LC2:
        .string "the PI is: %f"
.LC3:
        .string "the result is: %f"
        .text
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        mov     rbp, rsp
        .cfi_def_cfa_register 6
        sub     rsp, 16
        mov     DWORD PTR -8[rbp], 0
        mov     DWORD PTR -12[rbp], 0
        mov     DWORD PTR -4[rbp], 0
        lea     rax, -8[rbp]
        mov     rsi, rax
        lea     rdi, .LC0[rip]
        mov     eax, 0
        call    __isoc99_scanf@PLT
        lea     rax, -12[rbp]
```

```
    mov     rsi, rax
    lea     rdi, .LC0[rip]
    mov     eax, 0
    call    __isoc99_scanf@PLT
    mov     edx, DWORD PTR -12[rbp]
    mov     eax, DWORD PTR -8[rbp]
    mov     esi, edx
    mov     edi, eax
    call    func_add
    mov     DWORD PTR -4[rbp], eax
    mov     rax, QWORD PTR .LC1[rip]
    movq    xmm0, rax
2   lea     rdi, .LC2[rip]
    mov     eax, 1
    call    printf@PLT
    pxor    xmm1, xmm1
    cvtsi2sd        xmm1, DWORD PTR -4[rbp]
    movsd   xmm0, QWORD PTR .LC1[rip]
    mulsd   xmm1, xmm0
    movq    rax, xmm1
    movq    xmm0, rax
    lea     rdi, .LC3[rip]
    mov     eax, 1
    call    printf@PLT
    mov     eax, 0
    leave
```

We now have the assembly code, which follows a particular format. Strings are labeled as LC0, LC2, and so on.
But why does the compiler label strings like this?
For instance, in line 1, we see the string “the PI is %f”, which is then used in line 2 to pass to the printf function.
This is for linking purposes, which I will explain in later parts.
The compiler now has an assembly code that it can optimize and resolve any other necessary issues.

2-1-3 Assemble

In this phase, the compiler should convert the assembly code to the machine code. In the previous phase, the assembly code was generated, and now the compiler can convert the final version of it to machine code. These machine codes are a series of 0 and 1.

After this phase, we have a file named, **object** file. An object file is a compiled file, But it needs another phase to become an executable file. So in this phase, we have an object file.

Typically, each object file corresponds to one assembly file(compiled file), and each assembly file corresponds to one source file. So if we have three *.C/*.CPP files, we have three object files for them.

Now we can tell gcc to stop after the assembly phase and create an object file with the **-c** option.

```
$ gcc -c main.c
$ file main.o
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

OK, now we have an object file main.o. the compiler by default sets the name of the object file as the source file name.
When we examine the object file with the file command, we see a result like the above output.
But there are some hints that I should talk about them:

ELF 64-Bit: tell us that the target file is an ELF file that has a 64-bit architecture.

LSB: Least Significant Byte meaning that numbers are ordered in memory with their least significant byte first.

Relocatable: this keyword tells us that this file is an object file, not an executable because it can relocate and change shortly. I told you before that every object file corresponds to a source file and we can have multiple object files that everyone corresponds to a source file. So at the end compiler should join these object files together and make the final executable file. But in this phase, the compiler doesn’t know about other objects, and every object file is created separately from other objects, so the compiler doesn’t know about the addresses of functions and global variables. So it leaves those places that are used from these unknown addresses blank or fills with dummy values until in the last phase all object files are present, fills them with their real values. I’ll talk about relocating in the next parts.

Not stripped: this keyword tells us that this file is not stripped. It means that this file has string tables and debugging info. I’ll talk about this in the next part (ELF structure).

Now let’s see what is in an object file. Due to the file being a binary file and not a text file, we should use a hex-viewer tool. We can use xxd in Linux.

```
$ xxd main.o
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 0000 0000 0000 0000 f004 0000 0000 0000  ................
00000030: 0000 0000 4000 0000 0000 4000 0d00 0c00  ....@.....@.....
00000040: 5548 89e5 897d fc89 75f8 8b55 fc8b 45f8  UH...}..u..U..E.
00000050: 01d0 5dc3 5548 89e5 4883 ec10 c745 f800  ..].UH..H....E..
00000060: 0000 00c7 45f4 0000 0000 c745 fc00 0000  ....E......E....
00000070: 0048 8d45 f848 89c6 488d 3d00 0000 00b8  .H.E.H..H.=.....
00000080: 0000 0000 e800 0000 0048 8d45 f448 89c6  .........H.E.H..
```

As you see the content of the file is some binary data.
If you want to find out what are these bytes and dummy data, should wait until the next part. I'll discuss about ELF structure.
But for now, we can disassemble this object file and see the machine codes. We can use the objdump tool. objdump is a linear disassembler that can convert machine codes to assembly language.
For those who don't know about disassembly, I should say that when we write code in a high-level language like C, the compiler converts this code to machine code. but if we want to read these machine codes for reverse engineering proposes, it gets hard for us to read some hex values.
But There are some tools named, Disassembler that can convert machine codes to a low-level language like assembly. Objdump is one of them.
So we can use Objdump to see what is in this object file.

```
$ objdump -s -M intel main.o
main.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <func_add>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
/* I omit some lines */
  10:   01 d0                   add    eax,edx
  12:   5d                      pop    rbp
  13:   c3                      ret

0000000000000014 <main>:
  14:   55                      push   rbp
  15:   48 89 e5                mov    rbp,rsp
  18:   48 83 ec 10             sub    rsp,0x10
  1c:   c7 45 f8 00 00 00 00    mov    DWORD PTR [rbp-0x8],0x0
  23:   c7 45 f4 00 00 00 00    mov    DWORD PTR [rbp-0xc],0x0
  2a:   c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4],0x0
  31:   48 8d 45 f8             lea    rax,[rbp-0x8]
  35:   48 89 c6                mov    rsi,rax
  38:   48 8d 3d 00 00 00 00    lea    rdi,[rip+0x0]        # 3f <main+0x2b>
  3f:   b8 00 00 00 00          mov    eax,0x0
  44: e8 00 00 00 00            call   49 <main+0x35>
/* I omit some lines */
  a1:   f2 0f 59 c8             mulsd  xmm1,xmm0
  a5:   66 48 0f 7e c8          movq   rax,xmm1
  aa:   66 48 0f 6e c0          movq   xmm0,rax
  af:   48 8d 3d 00 00 00 00    lea    rdi,[rip+0x0]        # b6 <main+0xa2>
  b6:   b8 01 00 00 00          mov    eax,0x1
  bb:   e8 00 00 00 00          call   c0 <main+0xac>
  c0:   b8 00 00 00 00          mov    eax,0x0
  c5:   c9                      leave
  c6:   c3                      ret
```

As you see the results are disassembled codes.

If you remember, I talked about relocation in the previous section and told you that the compiler doesn't know about addresses.
Now if you look at this assembly code, for example at line 44, see that we have a call operator whose argument is a number 49.
call operator is an Intel x86 code to jump to a function (If you back and see the source code see that the first function that we used is "scanf"). So the call operator needs the target function address.
But here we see just a 49 number.
Surely this is not the real address of the "scanf" function because the compiler doesn't know the real addresses.
The compiler just fills these addresses with **dummy** values like 49, until the next phase which is the **linking** phase, and then replaces them with real values. so you can find out why an object file is relocatable and isn't executable.

The compiler job is down here and at the next phase, the compiler just calls another tool named **Linker**.

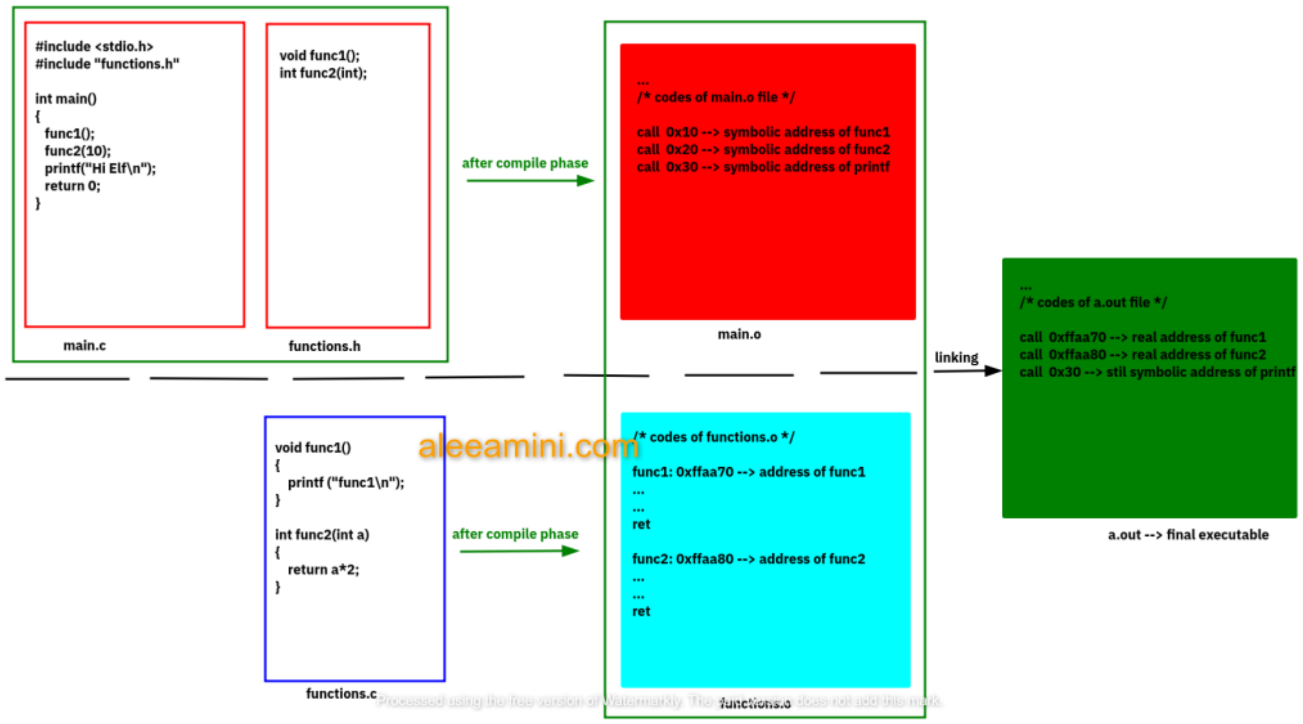
OK, let's see the last phase.

2-1-4 Linking

The last phase is linking. At the linking phase, the compiler calls the Linker. In Linux, we have a tool "ld" that does this job.
In this phase, the linker links all objects together and makes the final executable file.
Based on which type of executable file (executable or shared object) you want, the linker decides how to link codes.
In the previous phase, we saw that addresses in the object files were not resolved and the compiler filled them with some symbolic values and left them for the linker.
But in this phase, the linker knows about all object files and can replace the symbolic addresses with their real values.
However, some addresses still are unknown for the linker.
For example, in our sample code, we used from "printf" and "scanf" functions. Also, we used another function that is in our code, "func_add".

Now in the linking phase the linker searches in all created object files, to locate "func_add" and in our case, finds it in the main object file and replaces the real address with that dummy address.
But what about "printf" and "scanf" functions that are external functions?
As you know the "scanf" and "printf" are functions that exist in the C runtime library or "stdlib(GLIBC)" in Linux OS and the "stdlib" is a shared library that lives in Linux as a library file.
So here, we are using a library that is handled by OS, and the linker at the link time doesn't know about this library, addresses, where is it, and so on.

Now the linker decides to leave addresses of these functions with some symbolic values and continue its job. This type of linking will resolve at run time and it will done with the dynamic linker. I'll talk about types of linking in the next parts when you learn ELF structure.
In this picture, you see the process of compiling and linking.



#include <stdio.h>
#include "functions.h"
int main()
{
func1();
func2(10);
printf("Hi Elf\n");
return 0;
}
main.c
void func1();
int func2(int);
functions.h
after compile phase
...
/* codes of main.o file */
call 0x10 --> symbolic address of func1
call 0x20 --> symbolic address of func2
call 0x30 --> symbolic address of printf
main.o
linking
...
/* codes of a.out file */
call 0xffaa70 --> real address of func1
call 0xffaa80 --> real address of func2
call 0x30 --> stil symbolic address of printf
a.out --> final executable
void func1()
{
printf ("func1\n");
}
int func2(int a)
{
return a*2;
}
functions.c
after compile phase
/* codes of functions.o */
func1: 0xffaa70 --> address of func1
...
...
ret
func2: 0xffaa80 --> address of func2
...
...
ret
functions.o

Figure 2-2: ELF Compile Process

As you see the main.c file just knows that there are some functions. But it doesn't know where there exactly.
After compile phase in the main.o we see that the compiler addresses func1,func2, and printf with symbolic addresses because it doesn't know about them.
Also, on the other hand, the compiler compiles separately functions.c and make functions.o file.
Now in the functions.o the func1 and func2 are stabled at a particular address and now the compiler calls the linker.
Linker comes in and replaces symbolic addresses with real addresses but not for the printf function because we do not have any stdlib.o in this picture. So it should resolve at run time.

Let's see the linking phase

```
$ gcc main.c -o main.out
$ file main.out
main.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=93ac661cd1d1ca30d34a834b2295dbc54acc29af, for GNU/Linux 3.2.0, not
stripped
```

The -o option tells the name of the executable file.
The final executable file is created and now we see the type of file.
As you see we have an executable file, not relocatable.

Dynamically linked: The file has some symbols like functions that should resolve dynamically at runtime. Like printf function.

Interpreter: this tells us the name and path of the dynamic linker. Dynamic linker is that file that executes this file and resolves addresses of those unresolved functions.

Relocation
In the last section of this part, I want to talk about relocation.
The question is how the linker fills addresses of the unresolved functions.

OK, let’s go back to the previous phase, and compile. In this phase we have an object file Let us see again the disassembled code of our object file main.o:

2-2 Relocation

As you see in the highlighted line, at index 0x6b we have a call. This call is for the third function in our sample. This call is to the “func_add” function. At the leftmost column, we have 5 bytes of hex values. These bytes are the opcode of CALL operation and its operand in Intel x86 architecture.

```
$ objdump -s -M intel main.o
main.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000014 <main>:
  14:   55                      push   rbp
  15:   48 89 e5                mov    rbp,rsp
  /* omited some codes */
  38:   48 8d 3d 00 00 00 00    lea    rdi,[rip+0x0]        # 3f <main+0x2b>
  3f:   b8 00 00 00 00          mov    eax,0x0
  44:   e8 00 00 00 00          call   49 <main+0x35>
  49:   48 8d 45 f4             lea    rax,[rbp-0xc]
  4d:   48 89 c6                mov    rsi,rax
  50:   48 8d 3d 00 00 00 00    lea    rdi,[rip+0x0]        # 57 <main+0x43>
  57:   b8 00 00 00 00          mov    eax,0x0
  5c:   e8 00 00 00 00          call   61 <main+0x4d>
  61:   8b 55 f4                mov    edx,DWORD PTR [rbp-0xc]
  64:   8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
  67:   89 d6                   mov    esi,edx
  69:   89 c7                   mov    edi,eax
  6b:   e8 00 00 00 00          call   70 <main+0x5c>
  70:   89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  73:   48 8b 05 00 00 00 00    mov    rax,QWORD PTR [rip+0x0]        # 7a
<main+0x66>
  7a:   66 48 0f 6e c0          movq   xmm0,rax
  7f:   48 8d 3d 00 00 00 00    lea    rdi,[rip+0x0]        # 86 <main+0x72>
  86:   b8 01 00 00 00          mov    eax,0x1
  8b:   e8 00 00 00 00          call   90 <main+0x7c>
  /* omited some codes */
  c5:   c9                      leave
  c6:   c3                      ret
```

E8 is the opcode of CALL and the rest four bytes after it, are the address of the target which is the start address of a function. For example, if I want to call a function at address 0xff775522, in machine code we have this: E8 22 55 77 ff

The E8 is the opcode of CALL and the rest bytes are addresses of function in little-endian format.

But here we see that the address of the “func_add” function is four bytes of zeros. As said before, in this phase(Compile phase) the compiler doesn’t know about addresses. So leave them for Linking time.
Now let’s take a look at a table that exists in ELF files, the Relocation table.
Before I start talking about this table, I should tell you that I’ll describe all of these tables and the structure of the ELF file in the next part.
In an ELF file, we have many tables that contain information for some purpose.

One of them is the “Relocation” table or “.reloc”. this table in an object file, contains relocation information that helps the linker to resolve the addresses.

For a clear example let’s see the .reloc table of main.o file:

```
$ readelf --relocs main.o
Relocation section '.rela.text' at offset 0x350 contains 11 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000003b  000500000002 R_X86_64_PC32     0000000000000000 .rodata - 4
000000000045  000c00000004 R_X86_64_PLT32    0000000000000000 __isoc99_scanf - 4
000000000053  000500000002 R_X86_64_PC32     0000000000000000 .rodata - 4
00000000005d  000c00000004 R_X86_64_PLT32    0000000000000000 __isoc99_scanf - 4
00000000006c  000900000004 R_X86_64_PLT32    0000000000000000 func_add - 4

000000000076  000500000002 R_X86_64_PC32     0000000000000000 .rodata + 24
000000000082  000500000002 R_X86_64_PC32     0000000000000000 .rodata - 1
00000000008c  000d00000004 R_X86_64_PLT32    0000000000000000 printf - 4
00000000009d  000500000002 R_X86_64_PC32     0000000000000000 .rodata + 24
0000000000b2  000500000002 R_X86_64_PC32     0000000000000000 .rodata + d
0000000000bc  000d00000004 R_X86_64_PLT32    0000000000000000 printf - 4
```

“**readelf**” is a tool that can parse an ELF file and dump its contents. I tell it to show the Relocation table of the main.o object file.

As you see there are some columns. But the most important for us is the leftmost column, “Offset” and the rightmost, “Sym. Name+Addend”.

Imagine the process of linking started and the linker starts the linking phase. The linker opens all object files (here we just have one object file, main.o) and reads their Relocation tables. It starts reading function names from top to down finds the real address of them and replaces them with symbolic values.

For example, in our case, we want to find out how the linker resolves the address of “func_add”.

In the relocation table, in the fifth row, we see the entry “func_add”. linker lookup at all object files that were created at the compile phase to find the “func_add” function. Imagine this function is at 0xff8855aa address at file main.o. So now the linker knows the address of “func_add”. After that linker should find the locations that are used from this function.

But how? The key is here:
In the leftmost column “offset”, we see a 6-Byte value for the “func_add” entry it is 0x6c. what is it? It is the offset in the code section that the linker should overwrite with the real address. If you back and take a look at that call to “func_add”, you will see that the offset of that call is **0x6b** :

```
  6b:   e8 00 00 00 00          call   70 <main+0x5c>
```

So 0x6b is the offset of this CALL and is the offset of 0xE8, but in the relocation table, we see the value 0x6c. it's equal to the offset of the instruction that needs to be fixed, plus 1 (0x6b+1=0x6c). So we can find out that the value of offset in the relocation table points to the next byte after the opcode. Now the linker replaces these zeros bytes with the real address of "func_add":

```
  6b:   e8 aa 55 88 ff          call   1145 <main+0x5c>
```

Now after linking let's look at the same line in the final executable file:

```
$ objdump -S -M intel main.out
11a6:   8b 55 f4                mov    edx,DWORD PTR [rbp-0xc]
11a9:   8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
11ac:   89 d6                   mov    esi,edx
11ae:   89 c7                   mov    edi,eax
11b0:   e8 90 ff ff ff          call   1145 <func_add>
11b5:   89 45 fc                mov    DWORD PTR [rbp-0x4],eax
11b8:   48 8b 05 71 0e 00 00    mov    rax,QWORD PTR [rip+0xe71]
11bf:   66 48 0f 6e c0          movq   xmm0,rax
11c4:   48 8d 3d 40 0e 00 00    lea    rdi,[rip+0xe40]
11cb:   b8 01 00 00 00          mov    eax,0x1
11d0:   e8 5b fe ff ff          call   1030 <printf@plt>
11d5:   66 0f ef c9             pxor   xmm1,xmm1
11d9:   f2 0f 2a 4d fc          cvtsi2sd xmm1,DWORD PTR [rbp-0x4]
11de:   f2 0f 10 05 4a 0e 00    movsd  xmm0,QWORD PTR [rip+0xe4a]
11e5:   00
11e6:   f2 0f 59 c8             mulsd  xmm1,xmm0
```

As you see the address of "func_add" is resolved in the final executable and we haven't those zero bytes.
So we find out how the linker resolves the addresses in object files.

But remember that we have many types of relocation and in this section, I just talked about one of them.
We get end of the first part of ELF's Story.
In this part, we saw the process of compilation and linking.
I'll write other parts in a few days.
Good Luck
[Refrences]:
https://eli.thegreenplace.net
the Practical Binary Analysis Book

Filed under: Binary analysis,ELF's Story - @ 06/08/2023 23:27