Walkthrough: intro

In this binary, we are going to take a look at the basic toolkit that we use to disassemble and analyze binaries. This binary isn't meant to be difficult but rather as an introduction to assembly.

First Checks

When you download the binary, the first thing I recommend is to run it. Because you downloaded the binary from the internet, execution will be turned off by default. To fix this:

```
$  chmod +x intro
$  ./intro
```

We are provided the following output:

```
$  ./intro
Enter the flag here: flag
Nope, try again!
```

It seems that we need to just enter the flag and it will tell us if we get it right.

Second, we check the security measures of the file. Use `checksec` for this. If your program says `checksec: not found`, let me know and I think I know the problem. `checksec intro` prints the following:

```
[*] '/home/joybuzzer/Documents/vunrotc/live/00-introduction/intro/intro'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Let's break this down:

- `Arch: amd64-64-little` - This means that the binary is compiled for a 64-bit architecture. This is important because it means that we will be using 64-bit registers and instructions.
- `RELRO: Full RELRO` - This means that the Global Offset Table (GOT) is read-only. This is important because it means that we cannot overwrite the GOT to redirect execution to a different function. *More on this later.*
- `Stack: No canary found` - This means that the stack is not protected by a Canary. This is important because it means we can overwrite stack data without the program knowing.
- `NX: NX enabled` - This means that the stack is not executable. This is important because it means we cannot put outside instructions into the program and force the program to execute them.

- `PIE: PIE enabled` - This means that the binary is compiled with Position Independent Executable (PIE) enabled. This is important because it means that the binary will be loaded into a random location in memory. *More on this later.*

What does all this mean?

- We are dealing with a 64-bit binary.
- If we wanted to write to a specific place in memory, we can't because the stack & heap will be in different places every execution.

Let's start disassembling this binary.

Disassembly

Running this binary with `gdb` will let us see the assembly behind the program. To do this, we run `gdb intro`. From here, the best thing to check are the available functions (`info functions`):

```
gef➤  info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001080  __cxa_finalize@plt
0x0000000000001090  puts@plt
0x00000000000010a0  printf@plt
0x00000000000010b0  fgets@plt
0x00000000000010c0  strcmp@plt
0x00000000000010d0  malloc@plt
0x00000000000010e0  _start
0x0000000000001110  deregister_tm_clones
0x0000000000001140  register_tm_clones
0x0000000000001180  __do_global_dtors_aux
0x00000000000011c0  frame_dummy
0x00000000000011c9  main
0x0000000000001258  _fini
```

Most of these functions are standard library functions. The ones suffixed `@plt` are external library functions. The only function that we are interested in is `main`. Let's take a look at the assembly for `main` (`disas main`):

```
gef➤  disas main
Dump of assembler code for function main:
   0x00000000000011c9 <+0>: endbr64
   0x00000000000011cd <+4>: push   rbp
   0x00000000000011ce <+5>: mov    rbp,rsp
   0x00000000000011d1 <+8>: sub    rsp,0x20
   0x00000000000011d5 <+12>:    mov    DWORD PTR [rbp-0x14],edi
   0x00000000000011d8 <+15>:    mov    QWORD PTR [rbp-0x20],rsi
   0x00000000000011dc <+19>:    mov    edi,0x20
   0x00000000000011e1 <+24>:    call   0x10d0 <malloc@plt>
```

```
   0x00000000000011e6 <+29>:    mov    QWORD PTR [rbp-0x8],rax
   0x00000000000011ea <+33>:    lea    rax,[rip+0xe13]        # 0x2004
   0x00000000000011f1 <+40>:    mov    rdi,rax
   0x00000000000011f4 <+43>:    mov    eax,0x0
   0x00000000000011f9 <+48>:    call   0x10a0 <printf@plt>
   0x00000000000011fe <+53>:    mov    rdx,QWORD PTR [rip+0x2e0b]        #
0x4010 <stdin@GLIBC_2.2.5>
   0x0000000000001205 <+60>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000001209 <+64>:    mov    esi,0x20
   0x000000000000120e <+69>:    mov    rdi,rax
   0x0000000000001211 <+72>:    call   0x10b0 <fgets@plt>
   0x0000000000001216 <+77>:    mov    rax,QWORD PTR [rbp-0x8]
   0x000000000000121a <+81>:    lea    rdx,[rip+0xdf9]        # 0x201a
   0x0000000000001221 <+88>:    mov    rsi,rdx
   0x0000000000001224 <+91>:    mov    rdi,rax
   0x0000000000001227 <+94>:    call   0x10c0 <strcmp@plt>
   0x000000000000122c <+99>:    test   eax,eax
   0x000000000000122e <+101>:   jne    0x1241 <main+120>
   0x0000000000001230 <+103>:   lea    rax,[rip+0xdfd]        # 0x2034
   0x0000000000001237 <+110>:   mov    rdi,rax
   0x000000000000123a <+113>:   call   0x1090 <puts@plt>
   0x000000000000123f <+118>:   jmp    0x1250 <main+135>
   0x0000000000001241 <+120>:   lea    rax,[rip+0xe03]        # 0x204b
   0x0000000000001248 <+127>:   mov    rdi,rax
   0x000000000000124b <+130>:   call   0x1090 <puts@plt>
   0x0000000000001250 <+135>:   mov    eax,0x0
   0x0000000000001255 <+140>:   leave
   0x0000000000001256 <+141>:   ret
End of assembler dump.
```

Now this is a lot. The way I recommend to do this is to go from external function call to external function call, understanding what gets passed to them. Let's do this one at a time.

Hold on, One more thing

We need to understand how 64-bit functions pass parameters. These are done **via registers**. The first 6 parameters are passed in the following registers:

- `rdi` (sometimes as `edi`, the lower 4-bytes of `rdi`)
- `rsi` (sometimes as `esi`, the lower 4-bytes of `rsi`)
- `rdx` (sometimes as `edx`, the lower 4-bytes of `rdx`)
- `rcx` (sometimes as `ecx`, the lower 4-bytes of `rcx`)
- `r8` (sometimes as `r8d`, the lower 4-bytes of `r8`)
- `r9` (sometimes as `r9d`, the lower 4-bytes of `r9`)

The difference between `rdi` and `edi` is confusing. `rdi` and `edi` are connected to the same storage location, meainng that changing `edi` affects `rdi`, and vice versa. `edi` is just the bottom half of `rdi`, as we see below.

```
rdi = [ _ _ _ _ _ _ _ _ ]
        edi = [ _ _ _ _ ]
```

Assembly often passes parameters, especially parameters with small values, using the lower four bytes (i.e. `edi`). Assembly does this because *it's faster*, but is no different than passing all 8 bytes. The same goes for `rsi`, `rdx`, `rcx`, `r8`, and `r9`.

Now let's continue.

`malloc`

`malloc()` in C is how we allocate memory. It makes space on the heap for us to use. If you want to see the parameters for `malloc()`, we look at the `man` pages. This is where the function header is defined. `man malloc` shows the following:

```
SYNOPSIS
       #include <stdlib.h>

       void *malloc(size_t size);

DESCRIPTION
        The malloc() function allocates size bytes and returns a pointer to
        the allocated memory.  The memory is not initialized.  If size is
        0, then malloc() returns either NULL, or a unique pointer value
        that can later be successfully passed to free().
```

This tells us that `malloc()` takes a parameter being the number of bytes that gets passed to it. Since we know that `rdi` is the register that **always** holds the first parameter, we check above the function call for `rdi` being stored.

```
   0x00000000000011dc <+19>:    mov    edi,0x20
   0x00000000000011e1 <+24>:    call   0x10d0 <malloc@plt>
   0x00000000000011e6 <+29>:    mov    QWORD PTR [rbp-0x8],rax
```

Here we see that `edi` (aka `rdi`) is being loaded with `0x20` (`32`). This means that the equivalent function here is `malloc(0x20)`, which allocates `32` bytes on the heap.

The `str` variable resides in memory at address `rpb-0x8`. Immediately after the `malloc` returns, register `rax` contains the `malloc` return value, and that value is stored in the memory at address `rpb-0x8` (i.e. it is stored in `str`).

`printf`

Let's check the arguments for `printf`.

```
SYNOPSIS
       printf FORMAT [ARGUMENT]...
       printf OPTION

DESCRIPTION
       Print ARGUMENT(s) according to FORMAT, or execute according to
OPTION:
```

The `man` pages aren't super helpful here, but `printf` just prints text to the screen. `rdi` will have the string in memory that we want to print.

```
   0x00000000000011ea <+33>:    lea    rax,[rip+0xe13]        # 0x2004
   0x00000000000011f1 <+40>:    mov    rdi,rax
   0x00000000000011f4 <+43>:    mov    eax,0x0
   0x00000000000011f9 <+48>:    call   0x10a0 <printf@plt>
```

The first command is the most confusing. `lea` stands for **Load Effective Address**, which takes the address of the memory location and stores it in the register. This is the same as `mov rax, 0x2004`. The next command moves the address of the string into `rdi`, and the last command calls `printf`. *Why does assembly use `lea`?* `lea` is faster than `mov` because it doesn't have to access memory to get the value. It just gets the address and stores it in the register.

After this, it moves this address into `rdi`. If we check what's at this address, we see the following:

```
gef➤  x/s 0x2004
0x2004: "Enter the flag here: "
```

This makes sense with what we saw in the program output.

`fgets`

Let's check the `man` pages to see what `fgets` does.

```
SYNOPSIS
       char *fgets(char *s, int size, FILE *stream);

DESCRIPTION
       fgets()  reads  in  at  most one less than size characters from
stream and stores
       them into the buffer pointed to by s.  Reading stops after an EOF or
a  newline.
       If  a  newline  is  read,  it is stored into the buffer.  A
terminating null byte
       ('\0') is stored after the last character in the buffer.
```

This tells that `fgets()` is an input function. Since `fgets()` takes three arguments, we need to find what `rdi`, `rsi`, and `rdx` are to understand what's going into the function.

```
   0x00000000000011fe <+53>:    mov    rdx,QWORD PTR [rip+0x2e0b]        # 
0x4010 <stdin@GLIBC_2.2.5>
   0x0000000000001205 <+60>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000001209 <+64>:    mov    esi,0x20
   0x000000000000120e <+69>:    mov    rdi,rax
   0x0000000000001211 <+72>:    call   0x10b0 <fgets@plt>
```

Let's find the parameters.

- We see that `rdi` is loaded with `rax`, which is loaded with the address at `rbp-0x8`. From before, we know that this is where the `malloc` happened.
- `rsi` is loaded with `0x20`, meaning we are writing up to `0x20` bytes.
- `rdx` is loaded with the address of `rip+0x2e0b` (which GDB tells us is at `0x4010`). GDB informs us that this is `stdin`, which is the stream that the data comes from.

From this, we know that the C code looks like this:

```
buffer = malloc(0x20);
fgets(buffer, 0x20, stdin);
```

What does this mean? From earlier, we know that when `malloc` is called, data is written on the **heap**. This means that we're writing data to the heap, *rather than the stack*.

> Why is this important? This means that we can't perform many of the stack overflow techniques we are going to learn. There are a series of heap overflow techniques, but they are relatively out of the scope of the screener.

`strcmp`

`strcmp`, or *string compare*, is how C compares strings. The `man` pages shows us that it takes 2 arguments -- the strings to compare.

Let's go find `rdi` and `rsi`.

```
   0x0000000000001216 <+77>:    mov    rax,QWORD PTR [rbp-0x8]
   0x000000000000121a <+81>:    lea    rdx,[rip+0xdf9]        # 0x201a
   0x0000000000001221 <+88>:    mov    rsi,rdx
   0x0000000000001224 <+91>:    mov    rdi,rax
   0x0000000000001227 <+94>:    call   0x10c0 <strcmp@plt>
```

From this, we notice two things:

- `rdi` is loaded with the address of `rbp-0x8`, which is where the `malloc` happened. This means that `rdi` is the first string.
- `rsi` is loaded with the address of `rip+0xdf9`, which is `0x201a`. This is not something that we loaded. We notice that it is based on the instruction pointer (because of PIE), which tells us that it is something hardcoded. *More on this later*, but for now let's check what's there.

```
gef➤  x/s 0x201a
0x201a: "flag{welcome_to_runtime}\n"
```

As we expected, this is our flag. The following puts statements aren't really relevant to us, but we can guess by the code there that it prints a "yes" or "no" based on the return value of `strcmp`.

Conclusions

Based on this program, we see that the flag was hardcoded. This is not going to be the case in 99% of the binary exploitation problems. There is going to be some reverse engineering problems where the flag is obfuscated and your challenge is to figure out how it is converted, but not for binary exploitation problems.

For your reference, here is the source code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define STR_SIZE 0x20

int main(int argc, char* argv[])
{
    char* line = malloc(STR_SIZE * sizeof(char));
    printf("Enter the flag here: ");
    fgets(line, STR_SIZE, stdin);

    if (strcmp(line, "flag{welcome_to_runtime}\n") == 0)
        printf("That's the right flag!\n");
    else
        printf("Nope, try again!\n");

    return 0;
}
```

PS: You could have run `strings intro | grep flag` to find the flag. `strings` returns the hardcoded strings in binaries, so it's not a bad thing to check as a first step.