Pwn task

Author: Karol Baryła

There are 2 main bugs in the code that can be used together to exploit it.

- There is no bounds checking when accessing `stack_token` and `stack_value` - so we can easily cause buffer overflow with the right expression, e.g. one that begins with `'(' * 32`.
- When parsing `)` character, there is no check if `stack_token[sp-1] == TOKEN_VALUE` - which makes it possible to read memory we shouldn't be able to, by doing something like `(+)` - parsing of `+` character won't trigger write to `stack_value[sp]`, but parsing `)` will read this field, and copy it to previous one - while at the same time setting `stack_token[sp-2]` to `TOKEN_VAL`.

Combining those 2 bugs, we can e.g. create payload that reads canary from stack: `((((((((((((((((((((((((((((((+))))))))))))))))))))))))))))))`. Output values are always 64-bit floating point values, but we can get back original value using Python's struct module:

```
>>> struct.pack('d', -2.39820918180181417655e-39)
b'\x00\xb1\xbbB;\x1d\xea\xb7'>>> hex(struct.unpack('Q', b'\x00\xb1\xbbB;\x1d\xea\xb7')[0])
'0xb7ea1d3b42bbb100'
```

The only problem with payload like presented earlier, is that reading some value from stack also copies it to every field between between start of `stack_value` and original location. The way to fix it, is to write back the right values back after reading what we want. We can do it by pushing read value few slots down, going back up to the broken values, and writing them using proper combination of `)`'s, `+`'s and values.

Example: First, let's read value immediately after buffer and save it to variable `a`: `a=((((((((((((((((((((((((((((((+))))))))))))))))))))))))))))))`. Then, to read next value from stack, while preserving previous one, we can use payload like this: `b=((((((((((((((((((((((((((((((+)))))))+0*(0+a))))))))))))))))))))))))`. How it works: we push new value 8 slots down using 8x`)`. Then, we go 6 slots up, using `+0*(0+`, then write `a`'s value to proper slot - the one below original location of `b`. If we need to fix more entries, it is very similiar: `c=((((((((((((((((((((((((((((((+))))))))))+0*((0+b)+a))))))))))))))))))))))))`. After each written entry we go 2 slots down, and then one up, to write next entry.

We have everything we need, let's hack the program. The battle plan is to read some values from the stack in order to get libc's and binary's addresses. When we know addresses, we can create simple

ROP to execute `system('/bin/sh')`. This ROP will consist of 3 elements - pop rdi, then address of '/bin/sh' string in libc, and then address of `system()`. After a few hours of debugging, I got a working script. LINK TO SCIRIPT

And the flag from server: `BSK{s3Nd-r3cV-f1X-5enD-h4X}`

Now let's move on to "harder" version. The only difference is that it quits after executing expression that doesn't assign to variable. This effectively prohibits reading any contents of memory. This is a problem for our script, because we now have no way of leaking addresses back to us, so no way of calculating addresses needed for ROP.

The solution is pretty simple - the binary is a calculator, let's make it calculate everything for us. This would be impossible in general case - this calculator performs operations on doubles, we want to perform operations on representations of those doubles, treated as unsigned 64 bit ints. Turns out - that's not a problem here, thank's to how doubles are represented in memory. 12 high bits are sign bit and exponent - those will always be 0 in our case, because we operate on addresses, and in Linux those will always have 16 high bits cleared (or set to `1`'s, but that's not something we need to worry about here - I think it's only possible in kernel). So, if the sign bit and exponent are 0's, addition of 2 doubles is equal to addition of their representations (if the addition itself doesn't change exponent). `a * 2^-1023 + b * 2^-1023 = (a+b) * 2^-1023`. Well, that's all we need, let's modify exploit. LINK TO SCIRIPT

And the flag from server: `BSK{0n35hoT-w1Th-d3n0rM4l-f1o4Tz}`

Bonus: "beautiful" GDB script that was very helpful while debugging my exploits

---

Revision #6
Created Mon, Dec 28, 2020 6:56 PM by Lorak_
Updated Mon, Dec 28, 2020 8:03 PM by Lorak_