Ring 2: EncryptSvc2 [1]

I heard that EncryptSvc had a bug so I've updated it a bit.
I think there is no problem now. Isn't it?

`encsvc2.chall.cddc2020.nshc.sg 9005`

MD5("EncryptSvc2"): `64be4ff56953f0cdf832b91c3b8a56c4`

History

```
cddc_qualifiers_2019-writeups $ nc encryptsvc.cddc19q.ctf.sg 54321

-------------------------------------------
Text Encryption Service
-------------------------------------------

[Service Menu]
        1) Show example
        2) Encrypt message
        3) Decrypt message
        4) Show public key
        5) Quit

Your selection : 2

        You have selected [2]



[*] Input : wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
Input buffer overflow
[Service Menu]
        1) Show example
        2) Encrypt message
        3) Decrypt message
        4) Show public key
        5) Quit

Your selection : 3

        You have selected [3]

Failed to create RSAcddc_qualifiers_2019-writeups $
```

A long, long time ago,

Team <my-team> absolutely failed `encryptsvc`. Expected, considering we couldn't make heads-or-tails of a Buffer Overflow back then.

It isn't much too important to dwell on this past, but `encryptsvc2` shares too many similarities with its predecessor to simply ignore it. We'll do a cross-comparison between the two to figure out how to go about the challenge.

First off, the basic premise of *`encsvc`*, summarized:

- buffer overflow with opt [2],
- so as to replace the server's RSA key with your own,
- allowing the flag to be encrypted (opt [1]) with a known RSA key,
- which is then decrypted client-side, on the attacker's machine, with pycryptodome.

Barely any of that is relevant for `encryptsvc2`. Let's understand why.

`diff`

In no particular order, here are the important changes[1] from 2019 to 2020:

```
__int64 rsa_pub_enc_nopadding(__int64 from, unsigned int flen, __int64 a3,
__int64 to){
  return RSA_public_encrypt(flen, from, to, pub_key_struct, RSA_NO_PADDING);
}
```

Change: `RSA_NO_PADDING` was `RSA_PKCS1_PADDING`.

Importance: Exploit would not work with padding.

```
_QWORD modify_exponent(int new_exponent){ //note that new_exponent = 7
  _QWORD *pointer_to_EXPONENT; // rax
  pointer_to_EXPONENT = **(_QWORD ***)(pub_key_struct + 40LL); //
pub_key_struct->e
  *pointer_to_EXPONENT = new_exponent;
  return pointer_to_EXPONENT;
}
```

Change: This function didn't even exist in the original

Importance: Changes the exponent, `e`, of the server's RSA key. Basis of exploit.

```
for ( i = 0; i <= 1; ++i ){ //for-loop terminates after executing twice
    puts("[Service menu]\n\t1) Show example \n\t2) Encrypt message\n\t3) Decrypt
message\n\t4) Show publickey\n\t5) Quit");
    printf("\nselect : ");
    __isoc99_scanf(" %c", &opt);
    opt -= 48;
    printf("\n\tYou select [%d]\n\n", (unsigned int)opt);
    fgetc(stdin);
    if ( opt >= 0 && opt <= opt_max )
      break;
    sprintf(&s, "[-] Please select 1-5 (%02x)\n", (unsigned int)opt);
    printf("%s", &s);
  }
  return (unsigned int)opt; // can probably select any number
```

Change: No change at all!

Wait, what? It's important to know how to activate `modify_exponent()`, so I've shoved it here.

```
case 7u:       // you can write 7 twice to get here
        modify_exponent(v8); //v8 = 7
        goto LABEL_20;
```

Change: `case 7` didn't exist previously.

Importance: Allows the new function `modify_exponent()` to run.

That sums up the dirty work we need for reconnaissance. Now what?

Mathematics

Noting the changes in the binary, we can test[2] out what `case 7` does on the encryption key:

```
~$ ./EncryptSvc2
...
select : 1
...
[+] Encrypted Text :
iXtZGVB7i5ow9zPv8ghCGlYG/XMh2Dpqqkzp23d+H+kdTURUdHglP5cktRNy5Mn8
rv7YXGzmvNGtXPqr8a9TpYgZ4E3q78ViyUl73yOGjDDNz+4ubF6ntt9AZmDLtShX
jzYno4sNolUt1/sowc41yTE3LMAZ1kxf/HCF4Sp7Yw8=
...
select : 7
...
select : 7
...
select : 1
...
[+] Encrypted Text :
oyfiv/M8sakxFvjgEEAiueUn9sFGdJqoOGs6Ft451d7YJRrKceCrrW6luPtPDeqd
9ZgE+KzvXRPVrO2TAaGXj0aYauSaKZXjovhkNkTd8WhxsCDoRIWGcj8kKKuqt94k
Hf6w+SRvgyQAoZCniyvm1sXcyExJO/yUengmZ1EDNcM=
```

Or more succinctly represented in `pwn`:

```python
from pwn import *
from base64 import b64decode as bd
r = process('./EncryptSvc2')#remote('encsvc2.chall.cddc2020.nshc.sg', 9005)
def sendopt(o: int): r.sendlineafter(': ', str(o))
sendopt(1)
r.recvuntil('Encrypted Text : \n')
orig_c = int.from_bytes(bd(r.recvuntil('=\n')), 'big')
sendopt(7)
sendopt(7)
sendopt(1)
r.recvuntil('Encrypted Text : \n')
diff_c = int.from_bytes(bd(r.recvuntil('=\n')), 'big')
print(orig_c,'!=',diff_c)
```

Resulting in:

```
~$ python3 case7.py
965430230028113001938973344982619785557931154230713182124743631896345103044702112
227468490872418851656730989695274605681298514992808997098967333175054263827234593
827587262496045568882414765058723825936123323142493724498645519090187822265833646
079244558247792378063109974899594564392885622804601840412904874767 !=
114571901634333065744321849158945190449032331270343085891134360211702900662738864
727492948295006371731935775700557744313775111939349170299951661623972047165807478
242684485123061716755731864226173465276051849722753328060339760367754463691319040
998246529460922576323941044222243222833436210378016339202016490947
```

What's happening here is that the modification of the exponent (`e`) results in a different ciphertext `c`, because $c \equiv m^e (mod\ n)$, and the exponent---

Wait woah wait, m? n?

Or: *a rapidfire rundown of how RSA be*

RSA starts with a piece of plaintext, like e.g. `"CDDC20{your_flag_here}"`. That flag is coerced into a raw integer, `m` (meaning *message*), and is encrypted using the public exponent `e` (which is usually `0x10001` because of a few convoluted necessities) and the *modulus* `n`. The resultant encrypted data (*ciphertext*) is the value of the integer `c` obtained by the formula displayed on top.

Got none of that? Doesn't matter; it ain't important. What *is* important is this thing called Bézout's identity, which states that *two integers* **a** *and* **b** *with the common divisor* **d** *can be expressed in the form* **ax + by = d**, *where* **x** *and* **y** *are both integers*. The integers **x** and **y** can be obtained via the Extended Euclidean algorithm.

In the case where **a** and **b** are *coprime*, $ax + by = 1$.

How does that play into the challenge? The initial value of e is `0x10001 == 65537`, and the new value of the exponent, e', is `7`. `65537` and `7` are **coprime**, meaning that --- by Bézout --- there exists a pair of integers **x** and **y** satisfying the equation $ex + e'y = 1$.

Anything raised to the power of 1 is itself. $m^{ex+e'y} = m$.

By various power laws, $m^{ex+e'y} = (m^e)^x * (m^{e'})^y$

Remembering that $c \equiv m^e (mod\ n)$, $(m^e)^x * (m^{e'})^y \equiv c^x * c'^y (mod\ n)$, where c' is `diff_c`, the altered `c` produced by changing the exponent from `0x10001` to `7`.

Conclusion? $m \equiv c^x * c'^y (mod\ n)$. We've[3] just proven a relationship between the *ciphertexts* c & c', and the plaintext m.

Wrapping it all up

Following the python code written above, we need to grab the public key,

```
diff_e = 7 #just adding this here
from Crypto.PublicKey.RSA import import_key
sendopt(4)
r.recvuntil('Public key : \n')
k = import_key(r.recvuntil('END_PUBLIC_KEY-----'))
```

compute the integers **x** and **y**,

```
def xgcd(a,b): #copy/import this from somewhere
x,y = xgcd(k.e, diff_e)
```

and solve for m:

```
m = pow(orig_c,x,k.n) * pow(diff_c,y,k.n)
m %= k.n
print(m.to_bytes(300,'big').strip(b'\x00'))
```

Running the code locally, you'll get something like this:

```
~$ python3.8 rsa_without_integrity_checks.py
[+] Starting local process './EncryptSvc2': pid 2
e, e_err = 65537, 7
x, y = -2, 18725
oc =
96543023002811300193897334498261978555793115423071318212474363189634510304470211
22746849087241885165673098969527460568129851499280899709896733317505426382723459
38275872624960455688824147650587238259361233231424937244986455190901878222658336
46079244558247792378063109974899594564392885622804601840412904874767
dc =
11457190163433306574432184915894519044903233127034308589113436021170290066273886
47274929482950063717319357757005577443137751119393491702999516616239720471658074
78242684485123061716755731864226173465276051849722753328060339760367754463691319
040998246529460922576323941044222243222833436210378016339202016490947
n =
13705893009529177473128266281861747328591005086001407754489672155264927783294589
22022475432878460588569818854790214487601254902733980300332574248000185370439466
42741246367422453196396690063266227084127861597976202007921509942158359214496102
718177128062335501707083001961040391539525717334763079823863140244381
b'CDDC20{fake_flag_this_is_an_extra_buffer}\n'
```

The server's down, and I don't have the flag saved.

That'll be it.

Flag

`CDDC20{we_dont_have_it_saved__sorry_lads}`

Code

```python
from Crypto.PublicKey.RSA import import_key
from base64 import b64decode as bd
from pwn import *
#copypaste from online
def xgcd(a, b):
    """return (g, x, y) such that a*x + b*y = g = gcd(a, b)"""
    x0, x1, y0, y1 = 0, 1, 1, 0
    while a != 0:
        (q, a), b = divmod(b, a), a
        y0, y1 = y1, y0 - q * y1
        x0, x1 = x1, x0 - q * x1
    return x0, y0

#r = remote('encsvc2.chall.cddc2020.nshc.sg', 9005)
r = process('./EncryptSvc2')
def sendopt(o: int): r.sendlineafter(': ', str(o))
sendopt(1)
r.recvuntil('Encrypted Text : \n')
orig_c = int.from_bytes(bd(r.recvuntil('=\n')), 'big')
sendopt(7)
sendopt(7)
sendopt(1)
r.recvuntil('Encrypted Text : \n')
diff_c = int.from_bytes(bd(r.recvuntil('=\n')), 'big')
sendopt(4)
```

```
r.recvuntil('Public key : \n')
k = import_key(r.recvuntil('END PUBLIC KEY-----'))
diff_e = 7

x,y = xgcd(k.e, diff_e)
print('e, e_err = %d, %d' % (k.e, diff_e))
print('x, y = %d, %d' % (x,y))
print('oc = %d\ndc = %d\nn = %d' % (orig_c, diff_c, k.n))
m = pow(orig_c,x,k.n) * pow(diff_c,y,k.n)
m %= k.n
print(m.to_bytes(300,'big').strip(b'\x00')) #<-- this is expected to produce plaintext!
```

Footnotes

1. All of these are transcripted (with renamed variables, and cleaned typedefs) from IDA Pro.
2. Note that we're running the binary with a fake `flag` and a fake `public.pem`. Generate the latter with openssl.
3. And by "we", I mean this presentation I found online that describes the technique letter-by-letter.