HACKRUSH CTF

Team: Hackpunk

Members:
Harshit Ramolia
Harshil Purohit
Shashwat Parashar
Shreyash Agarawal

Binary Exploitation

Cliff :

We used format string vulnerability to obtain the hex values of the flag. As mentioned in this article printf can leak information from the stack if we input %llx. This gives the long long hex value from the stack. Using this information, we gave a long string - %llx %llx %llx %llx …. as input to the nc 3.142.26.175 12345.

We then ran the above input multiple times and noticed that only some values repeated whereas other values changed on each execution. This meant that the values that were changing must be garbage values whereas the values that did not change must represent some actual information. We observed that value starting from %10$llx remained the same. We then converted the output hex to Ascii and found “hsuRkcaH” as the Ascii value. This result motivated us and we felt that we were on the right track. We then fetched values uptill %15$llx after which the values again changed on different iterations. On decoding the hex values obtained from %10$llx to %15$llx, and reversing each string, we got the desired flag.

Flag: **HackRushCTF{N0w_Y0u_kn0w_ab0ut_form4t_5tr1ng5}**

Reverse Engineering

Simple_check :

Since we had been provided the code (c file) of this problem, solving this became very easy.

Manually find each character by finding its ascii value and converted to its character.

Key is **HackRushCTF{x86_f1r5t_t1m3?}**

mixed_up:

First we tried to break down the code using the assembly tools such as gdb and radare2. But none of them gave us the desired result and understanding the code wasn't that much easy.

Then searching for similar questions we came across a video https://www.youtube.com/watch?v=RCgEIBfnTEI which uses Ghidra to break down binary files into c code we used to see.

After breaking the binary file through Ghidra we found out the mixup function through which we could get the key. We wrote it as it is in c++ for getting the key.

We created an array of decimal values and then used a for loop to compare those values with local_14. However, that code seems to be lost.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{

    int local_14, param_1=1;
    int local_10;
    int local_c;
    while (local_14 != 190)  // 190 is decimal value to corresponding
character stored in FLAG array
    {
        local_14 = 0;
        local_10 = 0;
```

```
        while (local_10 < 4)
        {
            local_14 = local_14 | (1 << (local_10 & 31) & param_1) <<
((local_10 *( -2) + 7 & 31) & 255);
            local_10 = local_10 + 1;
        }
        local_c = 4;
        while (local_c < 8)
        {
            local_14 = local_14 | (int)(1 << (local_c & 31) & param_1) >>
(local_c * '\x02' - 7 & 31) & 0xffU;
            local_c = local_c + 1;
        }
        param_1++;
    }

    param_1--;
    cout<<local_14<<" "<<(char)param_1;

    return 0;
}
```

Flag: **HackRushCTF{Gh1dr4_1s_Tru!y_4w3s0m3}**

Cryptography

Ancient :

□□□□□□□□□ □□□□□ □□□□□ □□ □□□□□□□□□□ □□□□□□□̂ □□̂ □□ □□ □□ □□ □□ □□ □□□□□ □□□□□

After searching few letters it was easy to know that symbols belongs to brahmin script

Comparison of North Semitic and Brahmi scripts[58][note 2]

Phoenician	Aramaic	Value	Brahmi	Value
𐤀	𐡀	*	𑀅	a
𐤁	𐡁	b [b]	𑀩	ba
𐤂	𐡂	g [g]	𑀕	ga
𐤃	𐡃	d [d]	𑀥	dha
𐤄	𐡄	h [h], M.L.	𑀳	ha
𐤅	𐡅	w [w], M.L.	𑀯	va
𐤆	𐡆	z [z]	𑀚	ja
𐤇	𐡇	ḥ [ħ]	𑀖	gha
𐤈	𐡈	ṭ [tˤ]	𑀞	tha
𐤉	𐡉	y [j], M.L.	𑀬	ya
𐤊	𐡊	k [k]	𑀓	ka
𐤋	𐡋	l [l]	𑀮	la
𐤌	𐡌	m [m]	𑀫	ma
𐤍	𐡍	n [n]	𑀦	na
𐤎	𐡎	s [s]	𑀱	ṣa
𐤏	𐡏	ʿ [ʕ], M.L.	𑀏	e
𐤐	𐡐	p [p]	𑀧	pa
𐤑	𐡑	ṣ [sˤ]	𑀘	ca
𐤒	𐡒	q [q]	𑀐	kha
𐤓	𐡓	r [r]	𑀭	ra
𐤔	𐡔	š [ʃ]	𑀰	śa
𐤕	𐡕	t [t]	𑀢	ta

Image credits : https://en.wikipedia.org/wiki/Brahmi_script

Using above table we decrypted □ and □ then we found this fb post
https://www.facebook.com/1094200484098434/posts/1655395287978948/

Where it is mentioned 𑀅𑀲𑁄𑀓, Aśōka.
So we get the key and key is **HackRushCTF{asoka}**

Prime magic 1:

After reading resource about RSA through this
https://ctf101.org/cryptography/what-is-rsa/

We get to know that we have to find two prime numbers product of which is equal to

```
big_number =
25992347861099219061069221843214518860756327486173319027118759091795941826
930677
```

Using this website to factorized above number https://www.alpertron.com.ar/ECM.HTM
We get two prime numbers

```
a = 375716079290975467394539222629547559486 3
b = 691808237490131385512539766532597713557 9
```

Doing lcm of (a-1) and (b-1) using the site
https://goodcalculators.com/gcd-lcm-calculator/

```
c =
12996173930549609530534610921607259430372826121502753979294844150952160187
100118
```

Then using simple code we found out private key

```
for i in range(int(10e4)):
    z = (1+i*c)//exponent
    if (1+i*c)%exponent==0:
        d=z
        print(i,z,(1+i*c)%exponent)
        break
```

Private key (d) = 41431414610216059923624765550016093394384768322711909133681306566648379203973 75

Then using this code we found out the hex of flag

```
i = int(10e300)
# z = pow(10,(math.log10(big_number*i+encrypted_flag))/exponent)
# print(z)
message = pow(encrypted_flag,d,big_number)
m = hex(message)
print(m)
```

0x4861636b527573684354467b5253415f31735f6330306c7d

Converted to ascii characters we got the key and key is **HackRushCTF{RSA_1s_c00l}**

Prime magic 2:

Again for this https://www.alpertron.com.ar/ECM.HTM we found the factorization of the big_number, but unfortunately it wasn't 2 this time. The big_number was a multiplication of multiple primes.

There was an algorithm for decrypting multi prime RSA. It can be found here.

Flag: **HackRushCTF{10_1s_b3tt3r_th4n_2?}**

Double the trouble:

We first proceeded by taking a for loop that was nested 6 times to find the last three bytes in both the keys.

```
L = list(string.printable)

print(len(L))

for i in L:
    for j in L:
        for k in L:
            for m in L:
```

```
                for n in L:
                    for o in L:


                        key1 = b"0"*29 + i.encode() + j.encode() +
k.encode()
                        key2 = b"0"*29 + m.encode() + n.encode() +
o.encode()
```

This was followed by checking whether the key is correct and then decrypting. As the number of printable characters in Ascii is 100, the code had to go through a total of 100^6 iterations. This is clearly not feasible but still we kept the code running for quite some time. However, as expected we did not get any answer.

After searching for resources online, we edited the above code as follows:

```
import base64
from Crypto.Cipher import AES
import hashlib
from Crypto.Random import get_random_bytes
import string
import base64

test = b'testing 1..2..3!'
encrypted_test = b'v8MshgtU1CfDNDuajMHzkQ=='
flag = b'aUyXnj4SxFmYht39qIppFKIVDjQ/tTBbPwpSLo2IoHo='

encrypted_test = base64.b64decode(encrypted_test)
flag = base64.b64decode(flag)

iv = hashlib.md5(b"Goodluck!").digest()
alphabet = string.printable
key_base = '0'*29

# decrypting
phase1 = {}
data =  encrypted_test
for a in string.printable:
   for b in string.printable:
```

```
        for c in string.printable:
            key = key_base+a+b+c
            key = key.encode()
            cipher_decrypt = AES.new(key, AES.MODE_CBC, IV=iv)
            ciphertext = cipher_decrypt.decrypt(data)
            phase1[key] = ciphertext
print('P1 done')

# encrypting
phase2 = {}
data = test
for a in string.printable:
    for b in string.printable:
        for c in string.printable:
            key = key_base+a+b+c
            key = key.encode()
            cipher_encrypt = AES.new(key, AES.MODE_CBC, IV=iv)
            ciphertext = cipher_encrypt.encrypt(data)
            phase2[key] = ciphertext
print('P2 done')

s1 = set(phase1.values())
s2 = set(phase2.values())
s3 = s1 & s2
match = s3.pop()

for k,v in phase1.items():
    if v == match:
        key1 = k
        print(f'Key1: {key1}')
for k,v in phase2.items():
    if v == match:
        key2 = k
        print(f'Key2: {key2}')

# decrypt flag
data = flag
cipher_decrypt = AES.new(key1, AES.MODE_CBC, IV=iv)
ciphertext = cipher_decrypt.decrypt(data)
cipher_decrypt = AES.new(key2, AES.MODE_CBC, IV=iv)
```

```
ciphertext = cipher_decrypt.decrypt(ciphertext)
print(ciphertext.decode('utf-8'))
```

Source: https://vulndev.io/writeup/2019/12/28/ctf-2aes.html

The algorithm above is very clever as it encrypts with one key and decrypts with the other. Now, both must be equal and we can find the corresponding keys. Once we get the keys, we can find the flag.

Flag: **HackRushCTF{7w1c3_1s_n0t_b3tt3r}**