➢ WRITE-UP picoCTF 2019 (*Only Reverse Engineering*)

Author: b1n4Rhy

I. Chuỗi bài Vault-Door (training → 8):

1. vault-door-training - Points: 50

```
// -Minion #9567
public boolean checkPassword(String password) {
    return password.equals("w4rm1ng_Up_w1tH_jAv4_f845e860d96");
}
```

¯_(ツ)_/¯

Đọc code thì bài yêu cầu ta nhập vào một chuỗi kí tự rồi đem String password nhập vào đó truyền vào hàm **checkPassword** để so sánh với chuỗi flag, tuy dễ nhưng các bài level sau đều theo mô-túyp này, nên các bạn để ý nhé, mình sẽ không nhắc lại chuyện này ở các bài tới.

Flag : **picoCTF{w4rm1ng_Up_w1tH_jAv4_f845e860d96}**

2. vault-door-1 - Points: 100

```
public boolean checkPassword(String password) {
    return password.length() == 32 &&
           password.charAt(0)  == 'd' &&
           password.charAt(29) == '3' &&
           password.charAt(4)  == 'r' &&
           password.charAt(2)  == '5' &&
           password.charAt(23) == 'r' &&
           password.charAt(3)  == 'c' &&
           password.charAt(17) == '4' &&
           password.charAt(1)  == '3' &&
           password.charAt(7)  == 'b' &&
           password.charAt(10) == '_' &&
           password.charAt(5)  == '4' &&
           password.charAt(9)  == '3' &&
```

Sau bài Training dễ như ăn kẹo thì bài này cũng khiến ta phải suy nghĩ. Đề bài cho ta biết cần phải xây dựng lại String password để nhập thỏa mãn yêu cầu cũng như kiếm được flag. Tuy nghiên thay vì ngồi sắp xếp từng kí tự bằng tay thì ta sử dụng phương thức **String Builder** có sẵn của **Java** để xây dựng lại.

Java:

```java
public static void main(String args[]) {
    VaultDoor1 vaultDoor = new VaultDoor1();
    System.out.println(getFlag());
   }
public static String getFlag(){
    String str = "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~";
    StringBuilder sb = new StringBuilder(str);
        sb.insert(0,   'd' );
        sb.insert(29,  '3' );
        sb.insert(4,   'r' );
        sb.insert(2,   '5' );
        sb.insert(23,  'r' );
      /*...
      Do dài quá, các bạn tự pass hết phần còn lại nhé <3
      ...*/
        sb.insert(30,  '4' );
        sb.insert(25,  '_' );
        sb.insert(22,  '3' );
        sb.insert(28,  '8' );
        sb.insert(26,  'f' );
        sb.insert(31,  '3');
      return sb.toString().replace("~","");
  }
```

Flag : **picoCTF{d35c4mlb3r3HtH__4r3Tcf_833c454r3}**

3. vault-door-3 - Points: 200

Chả hiểu bài số 2 đâu rồi =))

```
public boolean checkPassword(String password) {
    if (password.length() != 32) {
        return false;
    }
    char[] buffer = new char[32];
    int i;
    for (i=0; i<8; i++) {
        buffer[i] = password.charAt(i);
    }
    for (; i<16; i++) {
        buffer[i] = password.charAt(23-i);
    }
    for (; i<32; i+=2) {
        buffer[i] = password.charAt(46-i);
    }
    for (i=31; i>=17; i-=2) {
        buffer[i] = password.charAt(i);
    }
    String s = new String(buffer);
    return s.equals("jU5t_a_sna_3lpm13gf49_u_4_mar24c");
}
```

Okey bài tiếp theo người ta yêu cầu mình *reverse* lại chuỗi **jU5t_a_sna_3lpm13gf49_u_4_mar24c** , đồng nghĩa với việc cho ta biết đây là flag luôn.

Để ý một chút ở các vòng lặp **for**:

- **Từ 0 tới 8**: 8 kí tự giữ nguyên và được gán vào mảng buffer.
- **Từ 8 tới 16**: các kí tự được gán vào mảng **buffer** là 23 trừ cho i, tức là i từ (23-8) tới (23-16) tương ứng.
- **Từ 16 tới 32**, i += 2 nên sẽ là các vị trí có i **chẵn**, tương tự với vòng lặp trên các kí tự được gán vào mảng sẽ là các kí tự thuộc i từ (46-16) tới (46 – 32) và i chẵn.
- **Từ 32 đổ dần về 17** với i -= 2, lần này nó sẽ lấy các kí tự có i **lẻ**, thay vì lấy chiều xuôi thì bây giờ lấy chiều ngược thôi.

Nhận xét: **Đây chỉ là bài kiểm tra kĩ năng lập trình đơn giản.**

C++:

```cpp
#include <bits/stdc++.h>
using namespace std;
string pw = "jU5t_a_sna_3lpm13gf49_u_4_mar24c";
string flag ="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
int solve(){
    int i;
    for (i=0; i<8; i++) {
        flag[i] = pw[i];
    }
       for (i=8; i<16; i++) {
           flag[i] = pw[23-i];
       }
       for(i=16;i<32;i+=2){
           flag[46-i] = pw[i];
       }
       for(i=17;i<=31;i+=2){
           flag[i]=pw[i];
       }
       cout<<flag;
  }
  int main(){
       solve();
  }
```

Flag : picoCTF{jU5t_a_s1mpl3_an4gr4m_4_u_9af23c}

4. vault-door-4 - Points: 250

```
public boolean checkPassword(String password) {
    byte[] passBytes = password.getBytes();
    byte[] myBytes = {
        106 , 85  , 53  , 116 , 95  , 52  , 95  , 98  ,
        0x55, 0x6e, 0x43, 0x68, 0x5f, 0x30, 0x66, 0x5f,
        0142, 0131, 0164, 063 , 0163, 0137, 0146, 062 ,
        '6' , 'a' , '8' , '9' , '8' , '1' , '5' , '1' ,
    };
    for (int i=0; i<32; i++) {
        if (passBytes[i] != myBytes[i]) {
            return false;
        }
    }
    return true;
}
```

String password được chuyển thành các Bytes và gán vào mảng passBytes. Lần này đề bài cho một mảng myBytes, và yêu cầu mình nhập vào sao cho các kí tự của mảng passBytes phải lần lượt bằng với myBytes.

Nhận xét: **Đây cũng là dạng bài kiểm tra kĩ năng lập trình, lần này kiểm tra bạn có thông thạo việc chuyển đổi các kiểu dữ liệu qua lại hay không.**

Java:

```
1.      public static void printflag(){
2.        byte[] myBytes = {
3.            106 , 85  , 53  , 116 , 95  , 52  , 95  , 98  ,
4.            0x55, 0x6e, 0x43, 0x68, 0x5f, 0x30, 0x66, 0x5f,
5.            0142, 0131, 0164, 063 , 0163, 0137, 0146, 062 ,
6.            '6' , 'a' , '8' , '9' , '8' , '1' , '5' , '1' ,
7.        };
8.        System.out.print("picoCTF{");
9.        for(int i =0;i<32;i++){
10.                char ch = (char)myBytes[i];
11.                System.out.print(ch);
12.
13.             }
14.             System.out.print("}");
15.     }
```

Flag : picoCTF{jU5t_4_bUnCh_0f_bYt3s_f26a898151}

5. Vault-Door-5 – Points 300

```
public String base64Encode(byte[] input) {
    return Base64.getEncoder().encodeToString(input);
}
public String urlEncode(byte[] input) {
    StringBuffer buf = new StringBuffer();
    for (int i=0; i<input.length; i++) {
        buf.append(String.format("%%%2x", input[i]));
    }
    return buf.toString();
}
public boolean checkPassword(String password) {
    String urlEncoded = urlEncode(password.getBytes());
    String base64Encoded = base64Encode(urlEncoded.getBytes());
    String expected = "JTYzJTMwJTZlJTc2JTMzJTcyJTc0JTMxJTZlJTY3JTVm"
                    + "JTY2JTcyJTMwJTZkJTVmJTYyJTYxJTM1JTY1JTVmJTM2"
                    + "JTM0JTVmJTM4JTMxJTY1JTMxJTY2JTM3JTYxJTYx";
    return base64Encoded.equals(expected);
}
```

Thoạt nhìn có vẻ dài và rối rắm nhưng đọc code thì thấy String **expected** được encode **Base64**. Người ta encode thì mình decode lại thôi. Hiện có rất nhiều công cụ online để decode ngược lại. Sau khi decode thì ta thu được một mảng các giá trị **hex**, việc cần làm là góp nhặt lại và viết code để in ra các char từ những giá trị đó:

Nhận xét: **Tương tự bài trước, bài này cũng kiểm tra kĩ năng code của các bạn. Tuy nhiên có làm khó hơn một chút ở phần encode Base 64.**

C++:

```
1. #include <bits/stdc++.h>
2. using namespace std;
3. int main(){
4.     char flag[32]={
5.         0x63,0x30,0x6e,0x76,0x33,0x72,0x74,0x31,0x6e,0x67,0x5f
6.         ,0x66,0x72,0x30,0x6d,0x5f,0x62,0x61,0x35,0x65,0x5f,0x36
7.         ,0x34,0x5f,0x38,0x31,0x65,0x31,0x66,0x37,0x61,0x61
8.         };
9.     for(int i =0;i<32;i++){
10.
11.           cout<<flag[i];
12.       }
13.   }
```

Flag : picoCTF{c0nv3rt1ng_fr0m_ba5e_64_81e1f7aa}

6. Vault-Door-6 Points 350

```java
public boolean checkPassword(String password) {
    if (password.length() != 32) {
        return false;
    }
    byte[] passBytes = password.getBytes();
    byte[] myBytes = {
        0x3b, 0x65, 0x21, 0xa , 0x38, 0x0 , 0x36, 0x1d,
        0xa , 0x3d, 0x61, 0x27, 0x11, 0x66, 0x27, 0xa ,
        0x21, 0x1d, 0x61, 0x3b, 0xa , 0x2d, 0x65, 0x27,
        0xa , 0x36, 0x64, 0x61, 0x62, 0x33, 0x67, 0x36,
    };
    for (int i=0; i<32; i++) {
        if (((passBytes[i] ^ 0x55) - myBytes[i]) != 0) {
            return false;
        }
    }
    return true;
}
```

Lần này để bài yêu cầu giải mã hóa XOR một mảng kiểu **byte** cho trước.
Theo những gì các bạn được học thì ta có: **a XOR b = c thì để tìm a, ta có: a = b XOR c**
Áp dụng công thức, chuyển vế đổi dấu, ta suy ra được: **passBytes[i] = myBytes[i] ^ 0x55**

Nhận xét: **Tiếp tục kiểm tra kĩ năng code và một chút kiến thức căn bản.**
C++ :

```cpp
#include <bits/stdc++.h>
using namespace std;
int main(){
    int myBytes[32] = {
            0x3b, 0x65, 0x21, 0xa , 0x38, 0x0 , 0x36, 0x1d,
            0xa , 0x3d, 0x61, 0x27, 0x11, 0x66, 0x27, 0xa ,
            0x21, 0x1d, 0x61, 0x3b, 0xa , 0x2d, 0x65, 0x27,
            0xa , 0x36, 0x64, 0x61, 0x62, 0x33, 0x67, 0x36,
        };
      for(int i = 0;i<32;i++){
          char p = myBytes[i]^0x55;
          cout<<p;
      }
  }
```

Flag : picoCTF{n0t_mUcH_h4rD3r_tH4n_x0r_c147f2c}

7. Vault-Door-7-Points 400:

```
public int[] passwordToIntArray(String hex) {
    int[] x = new int[8];
    byte[] hexBytes = hex.getBytes();
    for (int i=0; i<8; i++) {
        x[i] = hexBytes[i*4]   << 24
             | hexBytes[i*4+1] << 16
             | hexBytes[i*4+2] << 8
             | hexBytes[i*4+3];
    }
    return x;
}

public boolean checkPassword(String password) {
    if (password.length() != 32) {
        return false;
    }
    int[] x = passwordToIntArray(password);
    return x[0] == 1096770097
        && x[1] == 1952395366
        && x[2] == 1600270708
        && x[3] == 1601398833
        && x[4] == 1716808014
        && x[5] == 1734292024
        && x[6] == 926443108
        && x[7] == 825569586;
}
```

Trong file đề bài đã có hướng dẫn cụ thể bằng tiếng Anh, nhưng mình sẽ dịch lại cho các bạn tiện dễ hiểu:

*Mỗi kí tự đều có thể được biểu diễn dưới dạng giá trị của 1 byte sử dụng ASCII encoding của nó. Mỗi byte có 8 bits, và một số nguyên gồm 32 bits, vì thế nên chúng ta có thể "**pack**" 4 bytes thành một số nguyên. Dưới đây là một ví dụ: nếu string hexa là "01ab", thì nó có thể được biểu diễn dưới dạng các bytes **{0x30, 0x31, 0x61, 0x62}.** Khi các bytes được biểu diễn dưới dạng nhị phân, chúng lần lượt là:*

0x30: 00110000

0x31: 00110001
0x61: 01100001
0x62: 01100010

Nếu ta đặt 4 số nhị phân trên thành một chuỗi liên tục thì sẽ thu được 32 bits, tức là được một số nguyên.

00110000011000101100001011000010 -> 808542562

Khi mà 4 kí tự có thể biểu diễn thành một số nguyên, ***password*** *có 32 kí tự có thể được biểu diễn thành một mảng có 8 số nguyên.*

- Đã lâu rồi mình mới lại ngồi dịch tay tâm huyết như thế này! (ಥ‿‿ಥ)

Nhận xét: **Đây chỉ là 1 trick khá thú vị. Nhưng không có gì đánh đố cả.**
Đá qua Hints của đề bài thì họ có chỉ tới trang web : https://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html để convert.
Tuy nhiên mình không ngố đâu, làm vậy mất công lắm, trên https://kt.gy/ có mục chuyển thẳng từ **INT** về **ASCII** luôn, **copy & paste** lần lượt các giá trị của mảng **x** vào mục **INT** rồi ghép các chuỗi ở mục **ASCII** lại ta sẽ thu được flag, bài này không yêu cầu code gì cả, à cứ code nếu thích ;) .

Flag: picoCTF{A_b1t_0f_b1t_sh1fTiNg_2878fd1512}

8. Vault-Door-8-Points 450

Khi nhận đề , bạn mở ra sẽ thấy một đống một code rối rắm, không xuống dòng, không TAB, nhìn rất chướng mắt và khó hiểu. Điều đầu tiên là sử dụng một công cụ **Beautify** có sẵn online hoặc tích hợp sẵn dưới dạng **package** của các **IDE** để "*làm đẹp*" lại. Chỉ khi này bạn mới dễ hiểu code làm gì.
Code khá là dài nên mình sẽ không chụp đề và đi vào ngay trọng tâm hai hàm chính là hàm **scramble** và **switchBits**

a/ Hàm switchBits:

- Đọc comment mà tác giả để lại, thấy điều kiện : **p1 < p2**
- Thật khó cho newbies để tìm hiểu sâu hoặc debug xem chức năng của các dòng lệnh của hàm này, nhưng hiểu một cách nôm na chức năng của hàm này là hoán đổi bit của kí tự **c** tại vị trí **p1** và **p2** cho nhau.
    - Ví dụ: kí tự "**a**" **:** có mã nhị phân là **01100001** được truyền vào hàm switchBits với **p1 = 2** và **p2 = 5** thì sau khi return ta sẽ thu được một kí tự có mã nhị phân là **01000101.**

b/ Hàm scramble:

Hàm này nhận tham số là String password mà bạn nhập vào xong thực hiện **switchBits** với các vị trí không trùng nhau và thỏa điều kiện **p1<p2**. Sau đó trả về một mảng a gồm các kí tự đã được hoán đổi vị trí bits.

Việc còn lại của hàm **checkPassword** là lấy mảng **scrambled** tức là mảng **a** return về từ hàm **scramble** về sau đó đem đi so sánh với mảng **expected** cho sẵn.

Nhận xét: **Đây cũng là một dạng kiểm tra kỹ năng lập trình của bạn. Cùng với đó là khả năng tư duy.**

Cách giải: Đối với những dạng bài này, ta không thể tính ngược lại từ mảng **expected** cho trước, nên phương pháp giải dễ nhất là **brute-force**, mình sẽ tìm kiếm giá trị **ASCII** từ 32 tới 128 vì biết chắc chắn tồn tại một kí tự bằng với phần tử của mảng expected tương ứng. Tại sao lại từ 32 tới 128 mà không phải là 0 tới 128? Vì những kí tự trong khoảng nhìn thấy là từ 32 trở đi (Hãy tham khảo bảng mã **ASCII**), thực ra chạy từ 0 lên cũng được nhưng vì chuyện tối ưu thuật toán nên mới làm vậy. ʃつ◕_◕ʅつ *< Qua môn CTDL-GT để làm gì khi không biết tối ưu thuật toán dễ như này? >*

Cùng với đó, đề bài cũng có hints gợi ý rằng ta nên tận dụng lại các hàm đã có. Nên thuật toán rất đơn giản, tìm kiếm một chuỗi các kí tự sao cho sau khi **switchBits** thỏa mãn mảng **expected.**

Java:

```
protected void bruteforce(){
        for(int i=0;i<32;i++){
            for(int j=32;j<128;j++){
                char s=(char)j;
                if(switchBits(s)==expected[i]){
                    System.out.print(s);
                    break;
                }
            }
          }
        }
```

Flag: picoCTF{s0m3_m0r3_b1t_sh1fTiNg_b7a40645d}
p.s: *Không hiểu sao khi mình code bằng C++ thì lại không giải được bài này, dù hai script y chang nhau. ʊ_ʊ Bạn đọc nào có tâm thì test lại dùm với ạ.*

II. Chuỗi bài asm (asm1 → asm4):

1. asm1 - Points: 200

Đề bài :

```
asm1:
<+0>:   push   ebp
<+1>:   mov    ebp,esp
<+3>:   cmp    DWORD PTR [ebp+0x8],0x35d
<+10>:  jg     0x512 <asm1+37>
<+12>:  cmp    DWORD PTR [ebp+0x8],0x133
<+19>:  jne    0x50a <asm1+29>
<+21>:  mov    eax,DWORD PTR [ebp+0x8]
<+24>:  add    eax,0xb
<+27>:  jmp    0x529 <asm1+60>
<+29>:  mov    eax,DWORD PTR [ebp+0x8]
<+32>:  sub    eax,0xb
<+35>:  jmp    0x529 <asm1+60>
<+37>:  cmp    DWORD PTR [ebp+0x8],0x53e
<+44>:  jne    0x523 <asm1+54>
<+46>:  mov    eax,DWORD PTR [ebp+0x8]
<+49>:  sub    eax,0xb
<+52>:  jmp    0x529 <asm1+60>
<+54>:  mov    eax,DWORD PTR [ebp+0x8]
<+57>:  add    eax,0xb
<+60>:  pop    ebp
<+61>:  ret
```

Đề bài của mình thì yêu cầu truyền giá trị **0x53e** vào chương trình trên, tuy nhiên có thể đề bài của bạn sẽ khác, vì thế nên flow chương trình sẽ khác ¯_(ツ)_/¯.

Chúng ta dễ nhận ra [**ebp + 0x8]** sẽ chứa giá trị **0x53e** được truyền vào. Bài này đơn giản nên mình không cần tìm cách compile, mà có thể làm tay luôn. Nào, debug bằng cơm với mình nhé:

```
<+0>:   push   ebp
<+1>:   mov    ebp,esp
```

Bạn có thể hiểu nôm na hai dòng này là khởi tạo khung stack. Với những bạn chưa làm quen với ngôn ngữ *assembly* cũng như các kiến thức về *kiến trúc máy tính* cơ bản thì có thể tham khảo tại : https://www.tutorialspoint.com/assembly_programming/index.html.

```
<+3>:   cmp    DWORD PTR [ebp+0x8],0x35d
<+10>:  jg     0x512 <asm1+37>
```

So sánh **[ebp+8]** và **0x35d**, nếu nó lớn hơn(**jg = jump greater)** thì nhảy về **<asm1 + 37>**, trường hợp này [**ebp + 8]** đang cầm giá trị **0x53e** nên lớn hơn **0x35d**, sẽ nhảy tới **<+37>** :

```
<+37>:  cmp    DWORD PTR [ebp+0x8],0x53e
<+44>:  jne    0x523 <asm1+54>
```

Lần này nó so sánh **[ebp+8]** và **0x53e ,** nếu **không bằng (jne = jump not equal)** thì sẽ nhảy về **<+54>**. Tuy nhiên hai cái bằng nhau nên nó sẽ thực thi tiếp lệnh **<+46>**:

```
<+46>:  mov    eax,DWORD PTR [ebp+0x8]
<+49>:  sub    eax,0xb
<+52>:  jmp    0x529 <asm1+60>
```

Tiếp đến, thanh ghi **eax** sẽ cầm giá trị của **[ebp + 0x8]**, tức là : **eax=0x53e**. Sau đó tới lệnh **sub eax,0xb**, lúc này:

eax = 0x53e – 0xb = 0x533 . Rồi nhảy về **<asm1+60> (jmp: là lệnh nhảy không điều kiện):**

```
<+60>:  pop    ebp
<+61>:   ret
```

Xong phim, chương trình hủy thanh stack (**pop ebp)** và trả về, tức là trả về (**ret**) giá trị của thanh ghi **eax.**

Flag: 0x533

2. asm2 – Points : 250:

Đề bài :

```
asm2:
    <+0>:   push   ebp
    <+1>:   mov    ebp,esp
    <+3>:   sub    esp,0x10
    <+6>:   mov    eax,DWORD PTR [ebp+0xc]
    <+9>:   mov    DWORD PTR [ebp-0x4],eax
    <+12>:  mov    eax,DWORD PTR [ebp+0x8]
    <+15>:  mov    DWORD PTR [ebp-0x8],eax
    <+18>:  jmp    0x50c <asm2+31>
    <+20>:  add    DWORD PTR [ebp-0x4],0x1
    <+24>:  add    DWORD PTR [ebp-0x8],0x86
    <+31>:  cmp    DWORD PTR [ebp-0x8],0x14bc
    <+38>:  jle    0x501 <asm2+20>
    <+40>:  mov    eax,DWORD PTR [ebp-0x4]
    <+43>:  leave
    <+44>:  ret
```

Lần này đề của mình yêu cầu truyền vào 2 tham số là **0xd** và **0x1e**. Tiếp tục là một bài đơn giản nhưng thay vì truyền 1 tham số thì lần này nó truyền vào tận 2 cái. Các bạn hoàn toàn có thể tiếp tục debug bằng cơm như bài asm1 nhưng mình sẽ chỉ cho bạn cách compile kết hợp đoạn code asm trên với 1 đoạn code C nhỏ (Cách làm này mình tham khảo từ 1 trang github).
Tuy nhiên, để nguyên đoạn code asm trên thì máy sẽ không compile được, nên các bạn phải xóa đi toàn bộ offset (**<+12>,<+31>,…** offset là mấy cái này nè), và chỉnh sửa một chút:

loop.s:

```
.intel_syntax noprefix
.global asm2

asm2:
    push   ebp
    mov    ebp,esp
    sub    esp,0x10
    mov    eax,DWORD PTR [ebp+0xc]
    mov    DWORD PTR [ebp-0x4],eax
    mov    eax,DWORD PTR [ebp+0x8]
    mov    DWORD PTR [ebp-0x8],eax
    jmp    part1
part2:
    add    DWORD PTR [ebp-0x4],0x1
    add    DWORD PTR [ebp-0x8],0x86
part1:
    cmp    DWORD PTR [ebp-0x8],0x14bc
    jle    part2
    mov    eax,DWORD PTR [ebp-0x4]
    mov    esp, ebp
    pop    ebp
    ret
```

Và một đoạn code C nhỏ để nhận asm:

solve.c:

```c
#include <stdio.h>


int main(void) {
    printf("flag : 0x%x\n", asm2(0xd, 0x1e));
    return 0;
```

```
7. }
```

Compile bằng terminal của linux, (bạn nào chưa có **gcc** thì có thể cài bằng lệnh:
sudo apt install gcc, ngoài ra nếu không compile 32 bit được là do máy bạn dùng 64 bit, google là ra cách sửa nhé, mình cũng quên ời) :

- **gcc -m32 -c loop.s -o loop.o**
- **gcc -m32 -c solve.c -o solve.o**
- **gcc -m32 -o result.out solve.o loop.o**
- **./result**

Flag : 0x46

3. asm3 – Points: 300:

Thôi chán compile rồi, bài này có vẻ dễ, làm chén cơm đi rồi debug tiếp này:

Đề bài:

```
.intel_syntax noprefix
.global asm3

asm3:
	push	ebp
	mov	ebp,esp
	xor	eax,eax
	mov	ah,BYTE PTR [ebp+0xb]
	shl	ax,0x10
	sub	al,BYTE PTR [ebp+0xe]
	add	ah,BYTE PTR [ebp+0xd]
	xor	ax,WORD PTR [ebp+0x12]
	nop
	pop	ebp
	ret
```

Đề của mình yêu cầu truyền vào 3 tham số (**0xaeed09cb**,**0xb7acde91**,**0xb7facecd**):
Sau khi chạy lệnh **mov ebp, esp** thì thanh stack nó giống như sau:

old ebp	← ebp
ret	← ebp + 0x4
0xaeed09cb	← **ebp + 0x8 (arg1)**
0xb7acde91	← **ebp + 0xC (arg2)**
0xb7facecd	← **ebp + 0x10 (arg3)**

Vì là little-endian (Tham khảo tại : https://en.wikipedia.org/wiki/Endianness) nên các tham số được chia nhỏ trong thanh stack như sau:

0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF	0x10	0x11	0x12	0x13
cb	09	ed	ae	91	de	ac	b7	cd	ce	fa	b7

Debug bằng **emulator**(https://carlosrafaelgn.com.br/asm86/)và theo dõi giá trị của thanh ghi **eax** (Vì khi ret, flag sẽ luôn nằm trong thanh ghi eax):

```
xor	eax,eax
```

(xor với chính nó sẽ trả thanh ghi về 0) ; **eax** = 0x00000000

```
mov     ah,BYTE PTR [ebp+0xb]
```

ah = 0xae ; **eax** = 0x0000ae00

```
shl     ax,0x10
```

shl ax, 0x10 là dịch chuyển bit của thanh ghi ax sang 10 về phía bên trái (**shl = shift logical left**).

; **eax** = 0x00000000

```
sub     al,BYTE PTR [ebp+0xe]
```

al = 0 - 0xac ; **eax** = 0x00000054

```
add     ah,BYTE PTR [ebp+0xd]
```

ah = ah + 0xde ; **eax** = 0x0000de54

```
xor     ax,WORD PTR [ebp+0x12]
```

ax = ax ^ 0xb7fa ; **eax** = 0x000069ae

Flag: 0x69ae

4. asm4 – Points: 400

Bài này truyền tham số là một string "**picoCTF_d899a**" cho xịn xò, với asm cỡ này thì thôi mình xin phép compile vậy, chứ ngồi debug hết chỗ code này chắc mẹ nuôi không nổi cơm mình mất! ʘ_ʘ

Bạn cũng cần chỉnh sửa đôi chút để có thể compile được với gcc:

```
.intel_syntax noprefix

.global asm4


asm4:
    push   ebp
    mov    ebp,esp
    push   ebx
    sub    esp,0x10
    mov    DWORD PTR [ebp-0x10],0x27d
    mov    DWORD PTR [ebp-0xc],0x0
    jmp    p1
p4:
    add    DWORD PTR [ebp-0xc],0x1
p1:
    mov    edx,DWORD PTR [ebp-0xc]
    mov    eax,DWORD PTR [ebp+0x8]
    add    eax,edx
    movzx  eax,BYTE PTR [eax]
    test   al,al
    jne    p4
    mov    DWORD PTR [ebp-0x8],0x1
    jmp    p2
p3:
    mov    edx,DWORD PTR [ebp-0x8]
    mov    eax,DWORD PTR [ebp+0x8]
    add    eax,edx
    movzx  eax,BYTE PTR [eax]
    movsx  edx,al
    mov    eax,DWORD PTR [ebp-0x8]
    lea    ecx,[eax-0x1]
    mov    eax,DWORD PTR [ebp+0x8]
    add    eax,ecx
    movzx  eax,BYTE PTR [eax]
    movsx  eax,al
    sub    edx,eax
    mov    eax,edx
```

```
    mov     edx,eax
    mov     eax,DWORD PTR [ebp-0x10]
    lea     ebx,[edx+eax*1]
    mov     eax,DWORD PTR [ebp-0x8]
    lea     edx,[eax+0x1]
    mov     eax,DWORD PTR [ebp+0x8]
    add     eax,edx
    movzx   eax,BYTE PTR [eax]
    movsx   edx,al
    mov     ecx,DWORD PTR [ebp-0x8]
    mov     eax,DWORD PTR [ebp+0x8]
    add     eax,ecx
    movzx   eax,BYTE PTR [eax]
    movsx   eax,al
    sub     edx,eax
    mov     eax,edx
    add     eax,ebx
    mov     DWORD PTR [ebp-0x10],eax
    add     DWORD PTR [ebp-0x8],0x1
p2:
    mov     eax,DWORD PTR [ebp-0xc]
    sub     eax,0x1
    cmp     DWORD PTR [ebp-0x8],eax
    jl      p3
    mov     eax,DWORD PTR [ebp-0x10]
    add     esp,0x10
    pop     ebx
    pop     ebp
    ret
```

Cú pháp compile & file code C tương tự bài asm2 mà mình đã chia sẻ ở trên. Chúc bạn may mắn (ᵔᴥᵔ)

Flag : 0x23e

III. Chuỗi bài droids (zero→four):

1. Zero: (droids0 - Points: 350):

Nhận được một file apk, việc đầu tiên mà mình làm là decompile ra các file code Java để đọc cho dễ hiểu. Hiện có nhiều công cụ online mà bạn có thể tìm thấy, mình recommend trang này: http://www.javadecompilers.com/apk

Thông thường tại folder sources/com/ sẽ là nơi chứa source code được decompile. Sau khi mở và đọc lần lượt các file java mình chú ý tới chỗ FlagstaffHill.java này:

```
public class FlagstaffHill {
    public static native String paprika(String str);

    public static String getFlag(String input, Context ctx) {
        Log.i("PICO", paprika(input));
        return "Not Today...";
    }
}
```

Dễ thấy lệnh Log.i sẽ chứa flag của chúng ta vào Log. Để xem log khi chạy file apk, bạn cần một emulator. Theo như đề bài hint, mình sẽ dùng Android Studio. Sau khi cài đặt phần mềm, bạn cần cài thêm một máy điện thoại ảo, mình thì sử dụng **Pixel 2 API 28**. Chạy file apk và click vào nút "HELLO, I'M A BUTTON", bạn để ý chỗ khung Log cat:

```
I/GnssLocationProvider: WakeLock released by handleMessage(REPORT_
a.hellocmu.picoctf I/PICO: picoCTF{a.moose.once.bit.my.sister}
W/audio_hw_generic: Not supplying enough data to HAL, expected pos
I/GnssLocationProvider: WakeLock acquired by sendMessage(REPORT_SV
I/GnssLocationProvider: WakeLock released by handleMessage(REPORT_
```

Đúng như dự đoán, flag nằm ở khung Log:

Flag : picoCTF{a.moose.once.bit.my.sister}

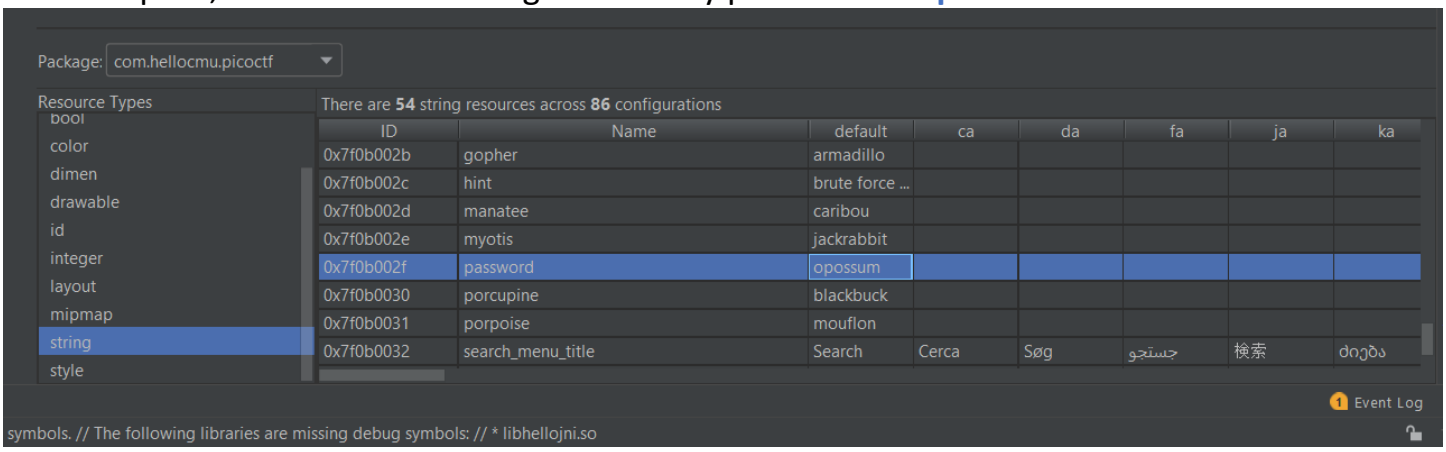
2. One (droids1 - Points: 400):

Tiếp tục decompile file one.apk, tiếp tục đọc file FlagstaffHill.java :

```
public class FlagstaffHill {
    public static native String fenugreek(String str);

    public static String getFlag(String input, Context ctx) {
        if (input.equals(ctx.getString(C0272R.string.password))) {
            return fenugreek(input);
        }
        return "NOPE";
    }
}
```

Lần này đề bài muốn chúng ta tìm ra password để thỏa mãn **C0272R.string.password.** Tuy nhiên, nếu bạn tìm trong file **C0272R.java** sẽ chỉ thấy toàn những con số. Lần này ta tiếp tục dùng tới Android Studio, nhấp chọn **resources.arsc**, đây là nơi chứa resources của chương trình. Tiếp tới, lần mò tới chỗ string và tìm thấy password là **opossum**



Nhập password vào chương trình và bấm nút, ta có được Flag.

Flag : picoCTF{pinning.for.the.fjords}

3. Two (droids2 - Points: 400):

Vẫn cứ như thế, lần này đề bài bắt chúng ta chơi trò ghép chuỗi, nó là ez để có được password, mình đã tính và chỉnh sửa lại nhìn cho đỡ rối:

```
1. public static String getFlag(String input, Context ctx) {
2.         String[] witches = {"weatherwax",
3.         "ogg", "garlick", "nitt", "aching", "dismass"};
4.         int second = 0;
5.         int third = 1;
6.         int fourth = 2;
7.         int fifth = 5;
8.         int sixth = 4;
9.         String str = ".";
10.             if (input.equals(
11.             ""          .concat(witches[fifth])          //dismass
12.             .concat(str).concat(witches[third])          //ogg
13.             .concat(str).concat(witches[second])         //weatherwax
14.             .concat(str).concat(witches[sixth])          //aching
15.             .concat(str).concat(witches[3])              //nitt
16.             .concat(str).concat(witches[fourth]))) {     //garlick
17.                 return sesame(input);
18.             }
19.             //password: dismass.ogg.weatherwax.aching.nitt.garlick
20.             return "NOPE";
21.         }
```

Done, nhập password vào khung input sẽ nhận được flag:
Flag : picoCTF{what.is.your.favourite.colour}

4. Three (droids3 - Points: 450):

Lần này đề bài yêu cầu hơn một chút, đó là patch file apk. Đọc code mà decompile ta dễ thấy hàm **nope** luôn được chạy trong khi đó cái mình cần là hàm **yep**.

```
public class FlagstaffHill {
    public static native String cilantro(String str);

    public static String nope(String input) {
        return "don't wanna";
    }

    public static String yep(String input) {
        return cilantro(input);
    }

    public static String getFlag(String input, Context ctx) {
        return nope(input);
    }
}
```

Phương án giải quyết đầu tiên mình nghĩ tới là patch hàm **nope** thành hàm **yep** trong hàm **getFlag**(). May thay, có một công cụ hỗ trợ mình làm điều đó: **apktool.jar** . Để chạy được tool này, máy bạn phải cài sẵn **JDK** nhé.
Bật **cmd** lên và khởi động tool:

- **apktool d three.apk**

Lúc này bạn sẽ thu được một folder **three** mà tool decompile ra được, di chuyển tới thư mục chứa đoạn code smailli của FlagstaffHill. Dùng một Text Editor bất kì để mở , sửa lệnh **nope** trong hàm getFlag thành **yep**, lưu lại. Ta tiến hành compile lại:

- **apktool b three**

Tuy nhiên compile xong không có nghĩa là bạn có thể chạy liền, ta phải đăng kí signature cho nó nữa:

- **keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA -keysize 2048 -validity 10000**
- **jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-key.keystore three.apk alias_name**

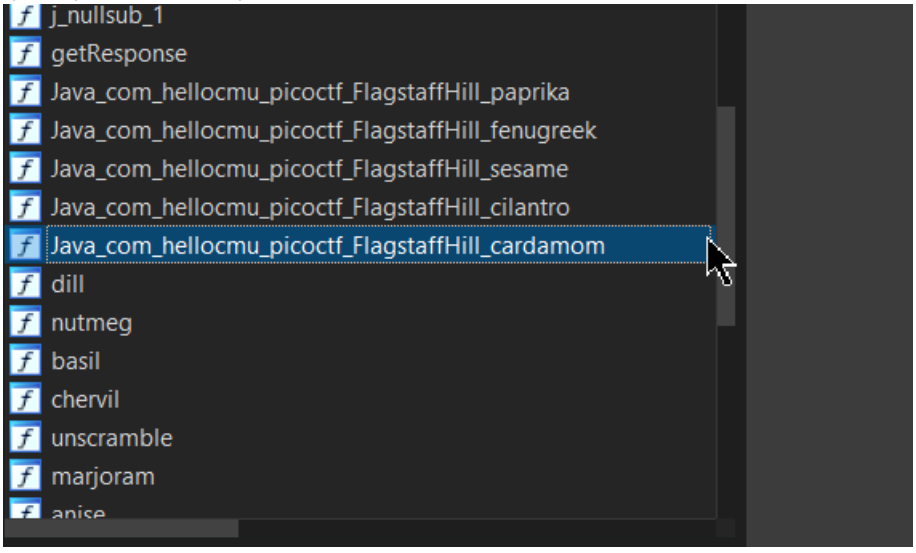
Xong, lúc này bạn có thể chạy file apk đã được patch với Android Studio, click cái nút để chạy hàm **getFlag**():
Flag: picoCTF{tis.but.a.scratch}

5. Four (droids4 - Points: 500):

Đây tiếp tục là một bài yêu cầu kỹ năng patch của chúng ta, nhưng lần này bạn sẽ không nhìn thấy bất cứ manh mối gì nếu chỉ decompile hay đọc code smalli thuần. Mình cũng luẩn quẩn một thời gian mới tìm ra phương án giải bài này.

Các bạn mở file apk với winrar sẽ thấy loạt folder, đây là các thư mục chứa resources và binary cho ứng dụng.

Mình thử load file .so được tìm thấy trong thư mục lib/x86-64 vào IDA và phát hiện ra một sự thật rất thú vị:



Các bạn thấy không, ở đây nó có đầy đủ tất cả các hàm mà getFlag() gọi, tức là nếu chúng ta patch hàm **paprika** ở bài zero thành **cardamom** thì có thể tìm thấy flag luôn của của bài four. Thiệt tình, ông nào ra đề hết sức lười biếng. Đúng là mấy người lười thường là thiên tài.
Theo ý tưởng đấy mình patch hàm cilantro của bài three thành hàm cardamom, cách patch và compile cũng như signature thì mình đã chỉ các bạn ở trên bài 3, và nó đã không làm mình thất vọng.

Password tính được vô cùng dễ bằng cách decompile thành file java: **alphabetsoup**
Flag : picoCTF{not.particularly.silly}

IV. Các bài khác:

1. reverse_cipher – Points: 300:

Bài này mình nhận được 1 file binary và 1 text, cat file text này ra ta được: **picoCTF{w1{1wq80haib767}** . Dường như flag đã được mã hóa. Tiếp tục ta thử kiểm tra file binary:

$ **file rev**

rev: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0, BuildID[sha1]=523d51973c11197605c76f84d4afb0fe9e59338c, not stripped

Hmm, okey không thu được gì nhiều ngoài việc file được build trên 64 bits. Ta load file vào IDA 7.2 64 bits.

F5, đọc code một chút thì dễ nhận ra flag được giấu như sau:

```
1. for ( j = 8; j <= 22; ++j )
2. {
3.   v11 = ptr[j];
4.   if ( j & 1 )
5.     v11 -= 2;
6.   else
7.     v11 += 5;
8.   fputc(v11, v7);
9. }
```

Vòng lặp for này sẽ lấy giá trị của mảng flag và thực hiện thuật toán sau đó viết ra file. Để Reverse lại hàm này lại, các bạn hãy để ý chỗ v11 -= 2 và v11+=5. Các ký tự có vị trí j lẻ sẽ được trừ 2, ngược lại cộng cho 5, sau đó được đưa ra file. Như vậy, mình chỉ cần lấy đúng ký tự có j thỏa mãn điều kiện và làm điều ngược lại: cộng 2 hoặc trừ 5 theo điều kiện của j thì sẽ thu được chuỗi ban đầu.

C:

```
1. #include <stdio.h>
2.
3. char flag[] = "w1{1wq8a8_g/fb6";
```

```
int main(){
    for(int i = 0;i<15;i++){
        if(i%2!=0){
            flag[i]+=2;
        }
        else{
                flag[i]-=5;
            }
         }
         printf("picoCTF{%s}",flag);
}
```

Flag: picoCTF{r3v3rs3c3ab1ad1}

2. Need For Speed – Points: 400

Nhận được một binary tên "**need-for-speed**" cùng một video Trailer "siêu quen thuộc" (ᵔ ͜ʖ ᵔ)
Nếu bạn nào có làm **picoCTF2018** thì bài này là một bản copy sửa mỗi cái tên.
Có rất nhiều cách để giải bài này, nào là các kiểu patch, chỉnh lại thông số của vòng lặp vô tận, dùng pwntools và python để patch, vân vân và mây mây...
Nhận xét: **Mục đích của tác giả có lẽ là để bạn làm quen với việc patch trong RE nên mới lặp lại đề năm ngoái.**
Mình xin phép chọn cách patch lại điều kiện của vòng lặp nhé:
Load file IDA, di chuyển tới hàm calculate_key(), F5 để đọc mã giả chứ đọc assembly chi cho cực:

```
__int64 calculate_key()
{
  int i; // [rsp+0h] [rbp-4h]

  for ( i = -1124885118; i != -562442559; --i )
    ;
  return 3732524737LL;
}
```

Dễ thấy, 3732524737LL là **key** nhưng hàm này chạy tới tết cũng không **return** về được. Nên mình sẽ patch thành :

```
__int64 calculate_key()
{
  int i; // [rsp+0h] [rbp-4h]

  for ( i = -562442558; i != -562442559; --i )
    ;
  return 3732524737LL;
}
```

Như vậy, chỉ cần chạy 1 nhịp là sẽ return về được ngay!
Làm sao để **patch** với IDA?

- Chọn View > Open Subviews > Dissasembly và chọn lại hàm **calculate_key()**, ấn phím cách để chuyển về chế độ đọc **asm** thuần.
- Chọn dòng : 0000000000000845 mov [rbp+var_4], 0DE79CEC1h
- Chọn Edit > Patch Program > Patch Bytes và sửa thành như sau:
  [81 7D FC C1 CE 79 DE 75 F3 8B 45 FC 5D C3 55 48]
- Click OK
- Chọn Edit > Patch Program > Apply patches to input file > OK để lưu các patch vào file của input, nếu bạn test thì nên tạo file backup .bak để tiện phục hồi nếu lỗi.

Tắt IDA đi và chạy file ở môi trường linux để tận hưởng thành quả :

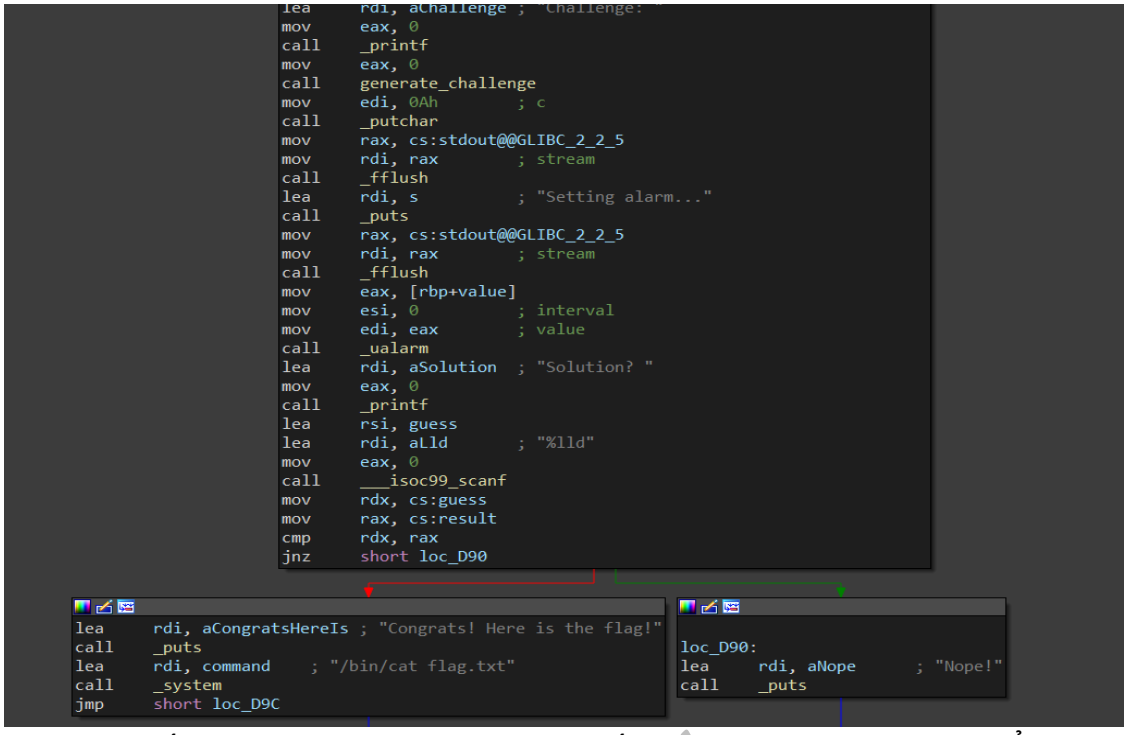
```
$ ./need-for-speed
Keep this thing over 50 mph!
============================
Creating key...
Finished
Printing flag:
PICOCTF{Good job keeping bus #079e482e speeding along!}
```

Flag : picoCTF{Good job keeping bus #079e482e speeding along!}

3. Times-up – Points: 400

Nhận được một binary tên : **times-up**
Kiểm tra file với lệnh **file times-up** trên linux thì cũng tương tự các bài phía trên, được build trên 64bits.

Kiểm tra trên IDA 64:



Bỏ ngoài tai mấy hàm random phép toán, biết vậy thôi vì mình cũng chẳng làm gì được nó cả. Dễ nhận thấy để lấy được flag thì ta phải nhập sao đó cho thỏa mãn điều kiện **guess = result**. Tuy nhiên, lần này flag không còn nằm ở **local** mà nằm tít trên **server**. Quay sang đọc hint thì người ta gợi ý rằng mình phải tìm cách tương tác với file bằng script.

Khi thử chạy file thì luôn bị trả về **SIGALRM**, đây là một loại signal trong hệ điều hành linux, trả về khi quá **timeout** của hàm **alarm.** Cùng với đó là một chuỗi các phép toán được **random**.

Signal là gì? Đúng như tên gọi thì đây là tín hiệu được gửi cho hệ điều hành hoặc ngược lại. Ví dụ khi bạn ấn tổ hợp phím **Ctrl + C** để hủy 1 tiến trình đang chạy, thì signal **SIGINT** sẽ được gửi cho hệ điều hành để xử lý (***SIGINT = Signal Interupt)***

Túm cái váy lại, cái mình cần là làm sao đó gửi giá trị **guess** trước khi bị **timeout** (5000). Ban đầu mình định patch lại nhưng lại nhận ra phải lên **server** mới có flag, vậy thì patch ở **local** có tác dụng gì! ¯_(ツ)_/¯

Ngoài ra, khi kiểm tra bằng lệnh **checksec** của gdb thì thấy rằng:

gef? **checksec**
[+] checksec for '**/home/rhy/Reverse/times-up**'
Canary : Yes
NX : Yes
PIE : Yes
Fortify : No
RelRO : Full

Nhận thấy **PIE** được bật, tức là nó đã giấu hết địa chỉ đi rồi, đó là lý do tại sao khi bạn load file vào IDA hay gdb thì không còn thấy các địa chỉ đầy đủ nữa.

Bên cạnh đó thì server cũng bật chế độ **ASLR** (**A**ddress **s**pace **l**ayout **r**andomization) : tức là mỗi lần chạy thì các địa chỉ của các dòng lệnh đều thay đổi. Tuy nhiên khi **disassemble main** ra thì nó có cho mình offset. Mình đã lên server, khởi động gdb, dùng lệnh **handle SIGALRM ignore** (Để cho chương trình không thấy signal này nữa), đặt **breakpoint** tại *<main + 174>*, và các bạn nhìn đoạn code sau là hiểu:

0x0000000d63 <+160>:mov rdx,QWORD PTR [rip+0x203a06] # 0x204770 <guess>
0x0000000d6a <+167>:mov rax,QWORD PTR [rip+0x203a07] # 0x204778 <result>
0x0000000d71 <+174>:cmp rdx,rax

Khi chương trình chạy tới breakpoint tức là trước lệnh **cmp rdx, rax** thì kết quả của thuật toán đã được mov vào **result**, trong khi đó số mình nhập vào được **mov** vào **guess**. Mình thay đổi giá trị của **guess** sao cho bằng **result** thử:

- **set $rdx = $rax**

Lúc này, hai thanh ghi đều đã bằng nhau, hàm **cmp** chắc chắn cho nhảy về lệnh **system(bin/cat flag.txt).** Tuy nhiên đời không như là mơ, mình bị **permission denied**, có vẻ gdb không đủ quyền để *cat flag*. ¯_(ツ)_/¯ Khôn như bạn quê tới đây, server said.
Quay lại với phương án viết script, đây là code giải:
Mình có sử dụng thư viện **pwntools** của python, tìm hiểu & hướng dẫn cài đặt tại :
http://docs.pwntools.com/en/stable/install.html
Python:

```
1. from pwn import *  # Nạp thư viện pwntools
2. r=process("./times-up") # Khởi động process cho times-up
3. r.recvuntil("Challenge: ") # Nhận dữ liệu từ sau chuỗi "Challenge:"
4. a=r.recvline() # Nhận chuỗi phép toán
5. result = eval(a) # Tính toán và trả về giá trị cho biến Result
6. r.sendline(str(result)) # Gửi lên process string Result
7. r.interactive() # Dừng để tương tác với process
```

Okey, test ở local thì có trả về file flag, tuy nhiên local thì làm gì có flag. Vô mục **Shell** của picoCTF, nhập tài khoản và mật khẩu của mình vào, sao đó **cd** tới thư mục mà btc đã cho trước. Và khởi động python:

$ python
>>> ((paste toàn bộ script python trên vào))
[*] Switching to interactive mode
Setting alarm...
[*] Process './times-up' stopped with exit code 0 (pid 58630)
Solution? Congrats! Here is the flag!
picoCTF{Gotta go fast. Gotta go FAST. #3c3eaafb}
[*] Got EOF while reading in interactive

Flag : picoCTF{Gotta go fast. Gotta go FAST. #3c3eaafb}