BÁO CÁO THỰC HÀNH

Lab 04: Reverse Engineering

**Môn học: Lập trình hệ thống**

**Tên chủ đề: Kỹ thuật dịch ngược**

*GVHD: Đỗ Thị Thu Hiền*

* **THÔNG TIN CHUNG:**

*(Liệt kê tất cả các thành viên trong nhóm)*

Lớp: NT334.M21.ANTN

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và tên** | **MSSV** | **Email** |
| 1 | Nguyễn Văn Tài | 19520250 | 19520250@gm.uit.edu.vn |
| 2 | Trần Hoàng Khang | 19521671 | 19521671@gm.uit.edu.vn |

* **NỘI DUNG THỰC HIỆN:[[1]](#footnote-1)**

|  |  |  |
| --- | --- | --- |
| **STT** | **Công việc** | **Kết quả tự đánh giá** |
| 1 | Phase 1 | 100% |
| 2 | Phase 2 | 100% |
| 3 | Phase 3 | 100% |
| 4 | Phase 4 | 100% |
| 5 | Phase 5 | 100% |
| 6 | Phase 6 | 100% |
| 7 | Secret Phase |  |

**Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.**

BÁO CÁO CHI TIẾT

Sử dụng IDA Pro để phân tích file **bomp**, và chương trình bao gồm các hàm chính sau đây:

A picture containing table

Description automatically generated

Phân tích từng phase và tìm ra các flag cần thiết để vượt qua các yêu cầu

## Phase\_1

Xem mã giả của phase\_1:

int \_\_cdecl phase\_1(int a1) {

    int result; // eax

    result = strings\_not\_equal(a1, "I was trying to give Tina Fey more material.");

    if (result)

        explode\_bomb();

    return result;

}

Đầu vào của hàm phase\_1 là biến a1 và so sánh với một chuỗi được gán sẵn trong hệ thống:

* Nếu chuỗi nhập vào giống nhau, thì sẽ vượt qua được phase\_1
* Ngược lại, nhập chuỗi sai sẽ khiến quả bom kích hoạt và thất bại

Vậy flag phase\_1 là: **I was trying to give Tina Fey more material.**

Kết quả:

Text

Description automatically generated

## Phase\_2

Xem mã giả phase\_2

unsigned int \_\_cdecl phase\_2(int a1)

{

  int i; // [esp+*10h*] [ebp-*28h*]

*int* v3[6]; // [esp+*14h*] [ebp-*24h*] BYREF

  unsigned *int* v4; // [esp+*2Ch*] [ebp-Ch]

  v4 = \_\_readgsdword(*0x14u*);

  read\_six\_numbers(a1, v3);

  if ( v3[0] != 1 )

    explode\_bomb();

  for ( i = 1; i <= 5; *++*i )

  {

    if ( v3[i] != 2 \* v3[i - 1] )

      explode\_bomb();

  }

  return \_\_readgsdword(*0x14u*) ^ v4;

}

Nhìn trông cũng khá easy, chưa tăng đô được bao nhiêu, hàm thực hiện lấy 6 số từ chuỗi nhập vào **read\_six\_numbers(a1, v3);** với a1 là chuỗi input có dạng **num1 num2 num3 num4 num5 num6** và đưa 6 số này vào mảng **v3[6]**.

* **v3[0] != 1** thì bomb nổ => v3[0] = 1
* Đưa vào vòng lặp *5 lần*, với mỗi *số phía trước gấp 2 lần số sau*, nếu không sẽ nổ

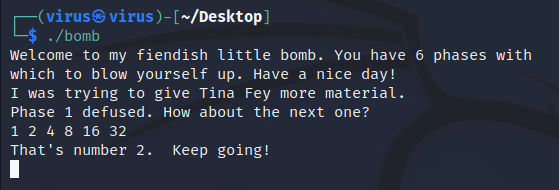
if ( v3[i] != 2 \* v3[i - 1] )

      explode\_bomb();

Vậy các số mình nhập vào chỉ cần thỏa mã điều kiện này và số đầu tiên phải bằng 1.

Ví dụ một chuỗi input hợp lệ cho phase 2: **1 2 4 8 16 32**

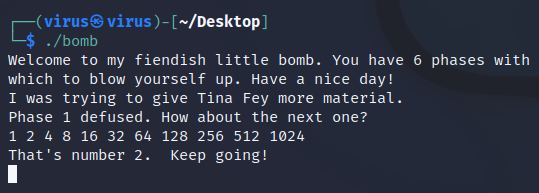
Test thực nghiệm:



Ngoài ra hàm không kiểm tra gì thêm mà chỉ kiểm tra với 6 số đầu, vậy nên nếu ta có nhập thêm thì cũng chả bị sao, hihi 😊

Ví dụ như: **1 2 4 8 16 32 64 128** cũng là một chuỗi hợp lệ

Vậy valid input có dạng tổng quát: **1 2 4 8 16 32 (64 128 … 2^n)**

****

## Phase\_3

Xem psedocode của phase\_3:

unsigned int \_\_cdecl phase\_3(int a1)

{

  unsigned int result; // eax

  int v2; // [esp+1Ch] [ebp-1Ch] BYREF

  int v3; // [esp+20h] [ebp-18h] BYREF

  int v4; // [esp+24h] [ebp-14h]

  int v5; // [esp+28h] [ebp-10h]

  unsigned int v6; // [esp+2Ch] [ebp-Ch]

  v6 = \_\_readgsdword(0x14u);

  v4 = 0;

  v5 = 0;

  v5 = \_\_isoc99\_sscanf(a1, "%d %d", &v2, &v3);

  if (v5 <= 1 )

    explode\_bomb();

  switch (v2)

  {

    case 0:

      v4 += 431;

      goto LABEL\_5;

    case 1:

LABEL\_5:

      v4 -= 858;

      goto LABEL\_6;

    case 2:

LABEL\_6:

      v4 += 437;

      goto LABEL\_7;

    case 3:

LABEL\_7:

      v4 -= 578;

      goto LABEL\_8;

    case 4:

LABEL\_8:

      v4 += 578;

      goto LABEL\_9;

    case 5:

LABEL\_9:

      v4 -= 578;

      goto LABEL\_10;

    case 6:

LABEL\_10:

      v4 += 578;

      break;

    case 7:

      break;

    default:

      explode\_bomb();

      return result;

  }

  v4 -= 578;

  if ( v2 > 5 || v4 != v3 )

    explode\_bomb();

    return \_\_readgsdword(0x14u) ^ v6;

}

Ở phase này, chương trình sẽ đọc vào 2 số nguyên (v2 và v3).

* v2 nằm trong khoảng giá trị từ [0, 5], nếu không sẽ kích hoạt quả bom
* v4 sẽ được tính toán tương ứng với từng giá trị của v2 và so sánh với giá trị của v3 (nếu khác nhau thì bom sẽ nổ)

Như vậy thì ứng với mỗi giá trị của v2 thì sẽ có một cặp kết quả {v2, v3} hợp lệ

* Có tất cả 6 input đầu vào hợp lệ để bypass qua phase\_3 này

Các input hợp lệ:

|  |  |  |
| --- | --- | --- |
| v2 | v4 | {v2, v3} |
| 0 | -568 | 0 -568 |
| 1 | -999 | 1 -999 |
| 2 | -141 | 2 -141 |
| 3 | -578 | 3 -578 |
| 4 | 0 | 4 0 |
| 5 | -578 | 5 -578 |

Kết quả:

Text

Description automatically generated

## Phase\_4

Xem mã giả phase\_4

unsigned int \_\_cdecl phase\_4(int a1)

{

  int v2; // [esp+*18h*] [ebp-*20h*] BYREF

*int* v3; // [esp+*1Ch*] [ebp-*1Ch*] BYREF

*int* v4; // [esp+*20h*] [ebp-*18h*]

*int* v5; // [esp+*24h*] [ebp-*14h*]

*int* v6; // [esp+*28h*] [ebp-*10h*]

  unsigned *int* v7; // [esp+*2Ch*] [ebp-Ch]

  v7 = \_\_readgsdword(*0x14u*);

  v4 = \_\_isoc99\_sscanf(a1, "%d %d", &v2, &v3);

  if ( v4 != 2 *||* v2 < 0 *||* v2 > 14 )

    explode\_bomb();

  v5 = 15;

  v6 = func4(v2, 0, 14);

  if ( v6 != v5 *||* v3 != v5 )

    explode\_bomb();

  return \_\_readgsdword(*0x14u*) ^ v7;

}

Tại phase\_4, hàm \_\_isoc99\_sscanf(a1, "%d %d", &v2, &v3) có chức năng tương tự như trong phase\_3 -> Đọc 2 số input đầu vào:

if ( v4 != 2 *||* v2 < 0 *||* v2 > 14 )

    explode\_bomb();

* Kiểm tra số lượng input đầu vào != 2 => Bomb nổ
* v2 nằm trong khoảng 0 <= v2 <= 14

v5 = 15;

v6 = func4(v2, 0, 14);

* Set giá trị v5 = 15
* v6 là giá trị trả về của hàm func4(v2, 0, 14) với các giá trị truyền vào như trên

if ( v6 != v5 *||* v3 != v5 )

    explode\_bomb();

Nếu giá trị trả về v6 != 15 (v5) và v3 != 15 => Nổ

Xem hàm ***func4()***

int \_\_cdecl func4(int a1, *int* a2, *int* a3)

{

  int v4; // [esp+Ch] [ebp-Ch]

  v4 = (a3 - a2) / 2 + a2;

  if ( v4 > a1 )

    return func4(a1, a2, v4 - 1) + v4;

  if ( v4 >= a1 )

    return (a3 - a2) / 2 + a2;

  return func4(a1, v4 + 1, a3) + v4;

}

Cơ bản thì *func4* là một hàm có liên quan đến giải thuật đệ quy, nhưng ta không quan tâm lắm, mục đích là tìm giá trị **v2** đầu vào ứng với tham số thứ 1 (**int a1**) của hàm *func4* để đầu ra giá trị trả về là 15 và được gán vào v6, còn điều kiện **v3 = 15** mình có thể set bằng tay. Vì giá trị của v2 hữu hạn, số nguyên và nhỏ - trong khoảng [0, 14]. Vậy ta có thể chạy chương trình vét cạn và xem trường hợp khả thi. Code solve:

def func4(*a1*, *a2*, *a3*):

    v4 = (*a3* - *a2*) / 2 + *a2*

    if v4 > *a1*:

        return func4(*a1*, *a2*, v4 - 1) + v4

    if v4 >= *a1*:

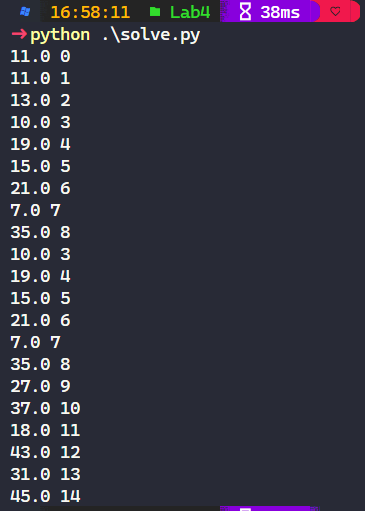
        return (*a3* - *a2*) / 2 + *a2*

    return func4(*a1*, v4 + 1, *a3*) + v4

for v2 in range(15):

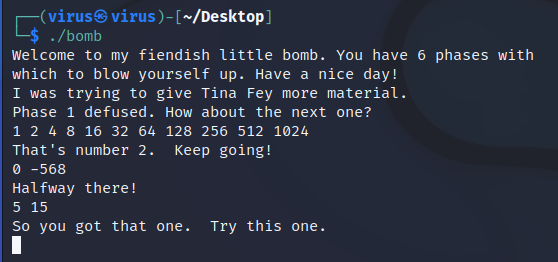
    print(str(func4(v2, 0, 14)) + " " + str(v2))

Giá trị output:



Có giá trị *v2 = 5* thì output bằng *15*, tức là *v6 = 15*. Vậy cặp giá trị này thỏa điều kiện, *v3* thì mình set bằng input luôn là *15*. Vậy giá trị input hợp lệ cho phase 4 là: **5 15**

Test thực nghiệm:



## Phase\_5

Xem psudocode của phase\_5:

int \_\_cdecl phase\_5(int a1)

{

  int result; // eax

  int i; // [esp+4h] [ebp-14h]

  int v3; // [esp+8h] [ebp-10h]

  result = string\_length(a1);

  if ( result != 6 )

    explode\_bomb();

  v3 = 0;

  for ( i = 0; i <= 5; ++i )

  {

    result = array\_2705[\*(\_BYTE \*)(i + a1) & 0xF];

    v3 += result;

  }

  if ( v3 != 48 )

    explode\_bomb();

  return result;

}

Ở phase này, input đầu vào là một chuỗi gồm 6 ký tự để thỏa mãn câu lệnh **if** đầu tiên.

Ở vòng lặp **for**, ta có luồng hoạt động như sau

* Từng ký tự của chuỗi input đầu vào sẽ được **&** với **0xF**
* Kết quả nhận được sẽ là **index** để truy xuất tới phần tử của mảng **array\_2705**

Mảng **array\_2705** chứa các giá trị:

A picture containing text

Description automatically generated

* Cộng giá trị của phần tử này vào **v3**
* Nếu tổng của các phần tử tìm được bằng 48 (v3 = 48) thì sẽ vượt qua phase\_5 này, ngược lại sẽ nổ bom.

Ta viết một đoạn code để tìm chuỗi cần nhập:

a = '000000'

while(len(a) == 6):

    v3 = 0

    for i in range(0, 6):

        res = arr[int(a[i]) & 0xF]

        v3 += res

    if v3 == 48:

        print(a)

    a = str(int(a) + 1)

    if(len(a) != 6):

        a = str(0)\*(6 - len(a)) + a

Kết quả: có rất nhiều kết quả phù hợp với điều kiện của phase\_5 này (từ 000000 -> 999999)

Text

Description automatically generated

Từ kết quả xuất hiện khi chạy đoạn code trên, ta chỉ cần lấy 1 giá trị là qua được phase này

Vậy flag là: **001414**

## Phase\_6

Xem psedocode của phase\_6, khá phức tạp:

unsigned int \_\_cdecl phase\_6(int a1)

{

  \_DWORD \*v2; // [esp+1Ch] [ebp-4Ch]

  int v3; // [esp+1Ch] [ebp-4Ch]

  int v4; // [esp+1Ch] [ebp-4Ch]

  int i; // [esp+20h] [ebp-48h]

  int k; // [esp+20h] [ebp-48h]

  int m; // [esp+20h] [ebp-48h]

  int n; // [esp+20h] [ebp-48h]

  int j; // [esp+24h] [ebp-44h]

  int l; // [esp+24h] [ebp-44h]

  int v11; // [esp+28h] [ebp-40h]

  int v12[6]; // [esp+2Ch] [ebp-3Ch] BYREF

  int v13[6]; // [esp+44h] [ebp-24h]

  unsigned int v14; // [esp+5Ch] [ebp-Ch]

  v14 = \_\_readgsdword(0x14u);

  read\_six\_numbers(a1, (int)v12);

  for ( i = 0; i <= 5; ++i )

  {

    if ( v12[i] <= 0 || v12[i] > 6 )

      explode\_bomb();

    for ( j = i + 1; j <= 5; ++j )

    {

      if ( v12[i] == v12[j] )

        explode\_bomb();

    }

  }

  for ( k = 0; k <= 5; ++k )

  {

    v2 = &node1;

    for ( l = 1; v12[k] > l; ++l )

      v2 = (\_DWORD \*)v2[2];

    v13[k] = (int)v2;

  }

  v11 = v13[0];

  v3 = v13[0];

  for ( m = 1; m <= 5; ++m )

  {

    \*(\_DWORD \*)(v3 + 8) = v13[m];

    v3 = \*(\_DWORD \*)(v3 + 8);

  }

  \*(\_DWORD \*)(v3 + 8) = 0;

  v4 = v11;

  for ( n = 0; n <= 4; ++n )

  {

    if ( \*(\_DWORD \*)v4 > \*\*(\_DWORD \*\*)(v4 + 8) )

      explode\_bomb();

    v4 = \*(\_DWORD \*)(v4 + 8);

  }

  return \_\_readgsdword(0x14u) ^ v14;

}

Phân tích code:

* Phần xử lý input:

v14 = \_\_readgsdword(0x14u);

  read\_six\_numbers(a1, (int)v12);

  for ( i = 0; i <= 5; ++i )

  {

    if ( v12[i] <= 0 || v12[i] > 6 )

      explode\_bomb();

    for ( j = i + 1; j <= 5; ++j )

    {

      if ( v12[i] == v12[j] )

        explode\_bomb();

    }

  }

* Chương trình gọi hàm **read\_six\_numbers ()** như ở phase\_2 và lưu vào biến **a1** => Input đầu vào là 6 số
* Ở vòng **for**, chương trình thực hiện các điều kiện input:
  + Giới hạn giá trị input nằm trong khoảng từ [0, 6]
  + Không có biến nào có giá trị giống nhau trong một lần nhập
* Nếu không phù hợp các điều kiện trên thì bom sẽ được kích hoạt
* Các phần tiếp theo khá khó hiểu, nên tiến hành phân tích thông qua code assembly

Sử dụng **gdb-peda** để debug.

Theo như đoạn mã psedocode trên, thì gần cuối chương trình mới kiểm tra điều kiện cụ thể.

v4 = v11;

  for ( n = 0; n <= 4; ++n )

  {

    if ( \*(\_DWORD \*)v4 > \*\*(\_DWORD \*\*)(v4 + 8) )

      explode\_bomb();

    v4 = \*(\_DWORD \*)(v4 + 8);

  }

**Tiến hành debug:**

Xem code assembly của phase\_6 bằng lệnh **disass phase\_6**

A screenshot of a computer

Description automatically generated with medium confidence

Đây là đoạn chương trình kiểm tra, ta có thể phác họa chương trình hoạt động như sau:

* Các giá trị được đưa vào hai thanh ghi **eax** và **edx**
* So sánh **edx** và **eax**, nếu giá trị **edx** nhỏ hơn **eax** thì sẽ thực hiện tiếp vòng lặp, nếu không thì sẽ kích hoạt quả bom

Đặt breakpoint và debug đoạn chương trình này:

* Truyền vào phase\_6 một đoạn input là “1 2 3 4 5 6”

A screenshot of a computer

Description automatically generated with medium confidence

* Thanh ghi **eax** lưu địa chỉ tại **[ebp-0x4c]** sau đó gán giá trị lưu tại **eax** vào **edx**, rồi cho **eax** chứa giá trị được lưu tại địa chỉ [**eax+0x8]** và so sánh hai giá trị lưu trong **eax** và **edx**

**A screenshot of a computer

Description automatically generated with medium confidence**

* Ta thấy địa chỉ 0x804c100 (**eax** đang giữ) là trỏ đến **node1**



* Tiếp tục chương trình, lúc này giá trị **eax** đã thay đổi

A screenshot of a computer

Description automatically generated with medium confidence

* Và **eax** đang trỏ đến **node2**

****

* Lúc này giá trị **eax** là 0x399 và **edx** là 0x124

A screenshot of a computer

Description automatically generated with medium confidence

Vì **edx** < **eax** nên chương trình tiếp tục chạy

* Tiếp theo, chương trình sẽ gán cho **eax** giá trị lưu tại node3 và edx giá trị tại **node2**

Text

Description automatically generated with medium confidence

* Vì **eax=0x22c**  < **edx=0x399** nên quả bom đã được kích hoạt

Sau khi phân tích trên, ta rút được các kết luận:

* Giá trị nhập vào sẽ liên quan đến **node** được sử dụng
* Để không kích hoạt quả bom, ta cần sắp xếp giá trị các **node** theo giá trị tăng dần

Sau một hồi debug, ta tìm được giá trị của các **node** lần lượt là:

**node1** có giá trị **124**

**node2** có giá trị **399**

**node3** có giá trị **22c**

**node4** có giá trị **0ed**

**node5** có giá trị **1c1**

**node6** có giá trị **15d**

Như vậy thứ tự sắp xếp lần lượt là: **node4 -> node1 ->** **node6->** **node5->** **node3** **->** **node2**

Tương tự, flag cần nhập là: **4 1 6 5 3 2**

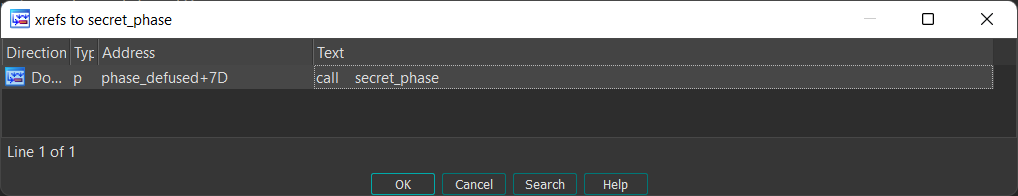
Kết quả:

Text

Description automatically generated

## 7. Secret Phase:

Ta thấy có hàm secret\_phase có vẻ là hàm ẩn của chương trình. Tìm các vị trí có sử dụng hàm này -> *Chuột phải -> Jump to xref …*



Ta thấy hàm được gọi ở một nơi duy nhất, vào xem thử. Ta thấy hàm được xuất hiện trong hàm phase\_defused():

unsigned *int* phase\_defused()

{

  char v1; // [esp+*0h*] [ebp-*68h*] BYREF

  char v2; // [esp+*4h*] [ebp-*64h*] BYREF

*int* v3; // [esp+*8h*] [ebp-*60h*]

  char v4[80]; // [esp+Ch] [ebp-*5Ch*] BYREF

  unsigned *int* v5; // [esp+*5Ch*] [ebp-Ch]

  v5 = \_\_readgsdword(*0x14u*);

  if ( num\_input\_strings == 6 )

  {

    v3 = \_\_isoc99\_sscanf(&unk\_804C530, "%d %d %s", &v1, &v2, v4);

    if ( v3 == 3 *&&* *!*strings\_not\_equal(v4, "DrEvil") )

    {

      puts("Curses, you've found the secret phase!");

      puts("But finding it and solving it are quite different...");

      secret\_phase();

    }

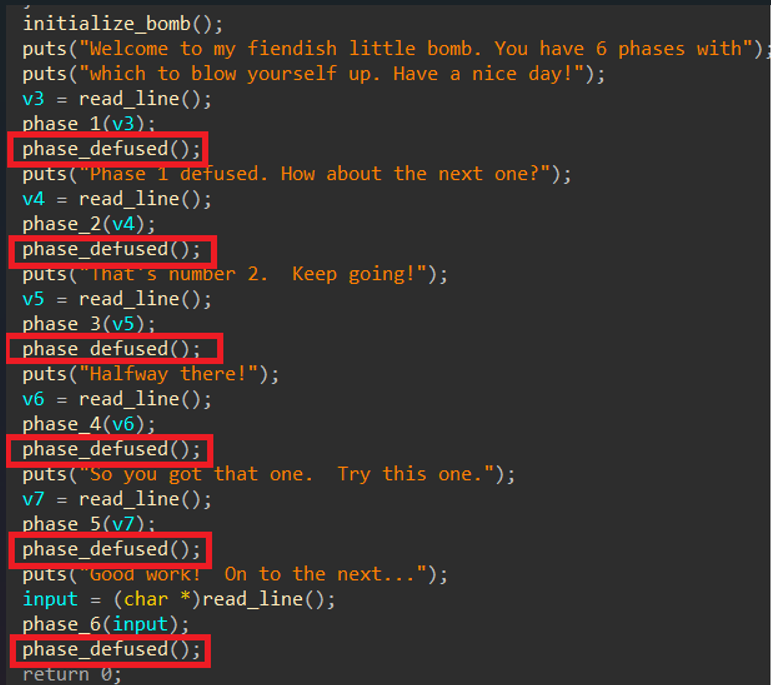
    puts("Congratulations! You've defused the bomb!");

  }

  return \_\_readgsdword(*0x14u*) ^ v5;

}

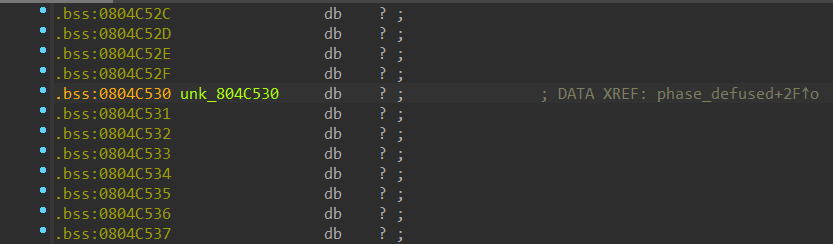
Và hàm phase\_defused() này được sử dụng xuyên suốt chương trình trong hàm main.



Xem lại đoạn chương trình trong hàm này. Ta thấy để vào được hàm secret\_phase() ta cần phải thỏa 2 điều kiện sau:

* num\_input\_strings = 6
* v3 = 3 && v4 = "DrEvil"

Và không có chuỗi chúng ta nhập vào rõ ràng. Giá trị truyền vào biến v1, v2 ,v4 lấy từ một biến lạ unk\_804C530 . Click vào thì thấy đây là *biến toàn cục* (nằm ở vùng ***.bss***) chưa được khởi tạo giá trị.

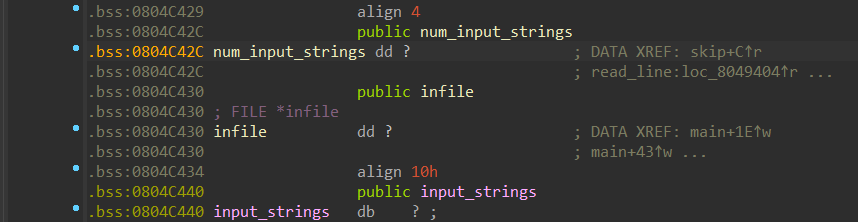


Vậy ta có thể mong muốn bằng một cách nào đó, với việc input vào các biến ở các phase trước, “có thể” dẫn đến việc ghi đè biến unk\_804C530 tại vùng nhớ này.

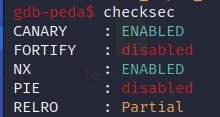
Vậy ta làm từng bước nhẹ nhàng nhưng kỹ càng:

* num\_input\_strings = 6

Để điều kiện này thỏa, tương tự ta xem biến num\_input\_strings -> Click:



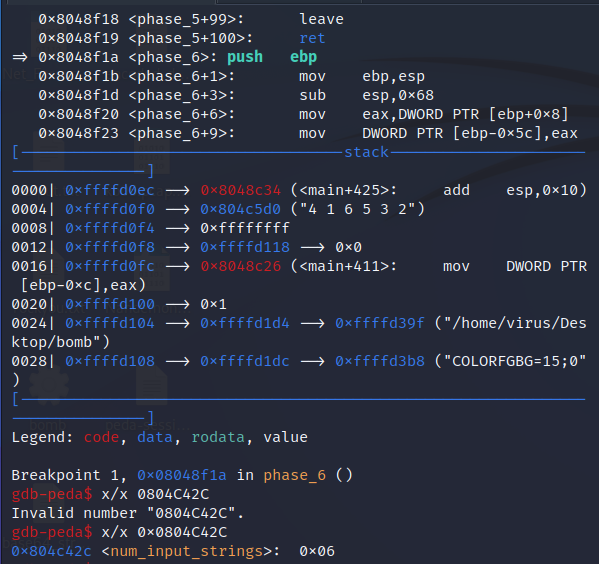
Vậy đây cũng là biến toàn cục nằm ở phân vùng **.bss**. Debug chương trình bằng gdb và đoán thử xem biến này được thay đổi theo điều kiện nào (vì đây là biến toàn cục nên mình có thể tùy ý xem giá trị bất cứ lúc nào). Hơn nữa tính năng random địa chỉ (PIE) của file này cũng bị tắt, nên địa chỉ của biến *num\_input\_strings (0x0804c42c)* và *unk\_804C530(0x0804c530)* là cố định và theo như trên:



Debug chương trình và set breakpoints (tại đầu phase\_6):

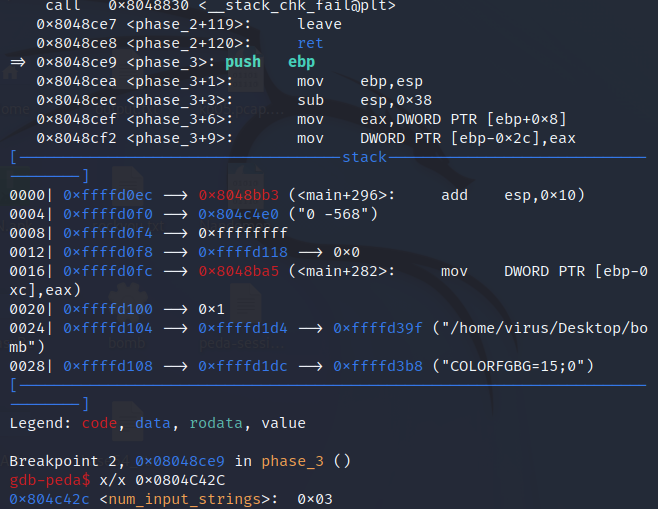
|  |
| --- |
| ┌──(virus㉿virus)-[~]  └─$ gdb bomb  gdb-peda$ b\*phase\_6 + 0  gdb-peda$ run  gdb-peda$ x/x 0x0804C42C |

Chạy chương trình cho kết quả sau:



Giá trị của *num\_input\_strings* là 6 (thỏa điều kiện). Thử đặt breakpoint tại các vị trí khác. Debug chương trình và set breakpoints (tại đầu phase\_3):

|  |
| --- |
| gdb-peda$ del breakpoints  gdb-peda$ b\*phase\_3 + 0  gdb-peda$ run  gdb-peda$ x/x 0x0804C42C |

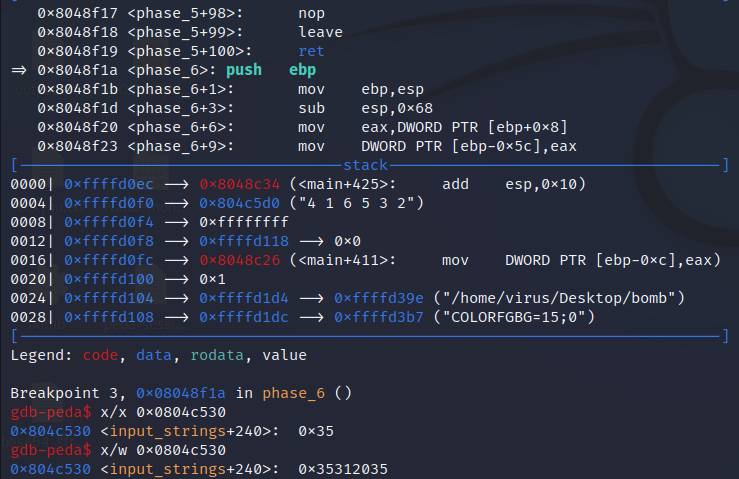


Lúc này giá trị của *num\_input\_strings* là 3. Vậy là giá trị của biến này là con số phase của bomb mà mình đang gỡ. Vậy ngay *phase\_6* là mình đã có giá trị thỏa mãn. Set debug với ngay tại đầu phase\_6 như trên và kiểm tra điều kiện thứ 2:

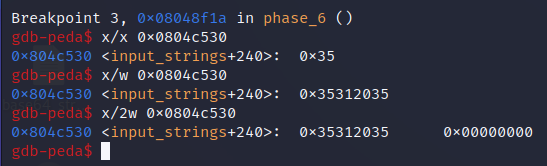
* v3 = 3 && v4 = "DrEvil"

Examine giá trị biến *unk\_804C530*  :

|  |
| --- |
| gdb-peda$ del breakpoints  gdb-peda$ b\*phase\_6 + 0  gdb-peda$ run  gdb-peda$ x/x 0x0804c530 |



Ta thấy giá trị của biến *unk\_804C530*  là **0x35312035**. Và giá trị này được viết dưới dạng Little-Endian(ie386-x86) nên giá trị là: “5 15”. Và rõ ràng đây là giá trị được nhập ở phase\_4. Xem các giá trị ở sau:



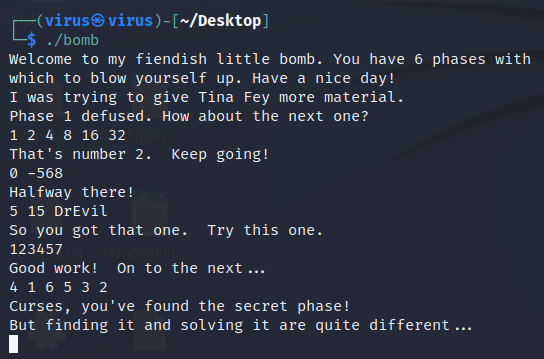
Vậy là input của phase\_4 hoàn toàn được ghi hết vào địa chỉ tại vùng nhớ này.

v3 = \_\_isoc99\_sscanf(&unk\_804C530, "%d %d %s", &v1, &v2, v4);

    if ( v3 == 3 *&&* *!*strings\_not\_equal(v4, "DrEvil") )

Chúng ta phải có 3 giá trị từ biến *unk\_804C530*  để thỏa điều kiện, một điều đáng mừng là ở các chương trình trên hoàn toàn không kiểm tra số lượng nhập vào nên ta có thể nhập thêm tham số thứ 3 tùy ý trong *phase\_4*. Tuy nhiên để thỏa điều kiện thì *tham số thứ 3(v4)* phải bằng với chuỗi “DrEvil”

Thực nghiệm chương trình:



Ok vậy là mình đã vào được secret\_phase. Xem mã giả chương trình:

unsigned int secret\_phase()

{

  char \*nptr; // [esp+*4h*] [ebp-*14h*]

*int* v2; // [esp+*8h*] [ebp-*10h*]

  nptr = (char \*)read\_line();

  v2 = atoi(nptr);

  if ( v2 <= 0 *||* v2 > 1001 )

    explode\_bomb();

  if ( fun7(&n1, v2) != 4 )

    explode\_bomb();

  puts("Wow! You've defused the secret stage!");

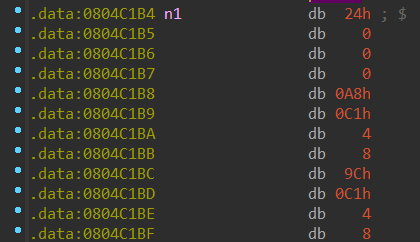
  return phase\_defused();

}

Chương trình khá đơn giản, nhập vào một chuỗi và hàm atoi() sẽ convert chuỗi đó thành số và đưa vào v2 -> Chuỗi nhập vào phải là định dạng số. Sau đó phải thỏa 2 điều kiện sau:

* 0 < v2 <=1001
* fun7(&n1, v2) = 4

*n1* là biến toàn cục được khởi tạo sẵn giá trị *(nằm trong phân vùng .data)* .



Vào hàm func7 :

int \_\_cdecl fun7(\_DWORD \*a1, int a2)

{

  if ( *!*a1 )

    return -1;

  if ( \*a1 > a2 )

    return 2 \* fun7((\_DWORD \*)a1[1], a2);

  if ( \*a1 == a2 )

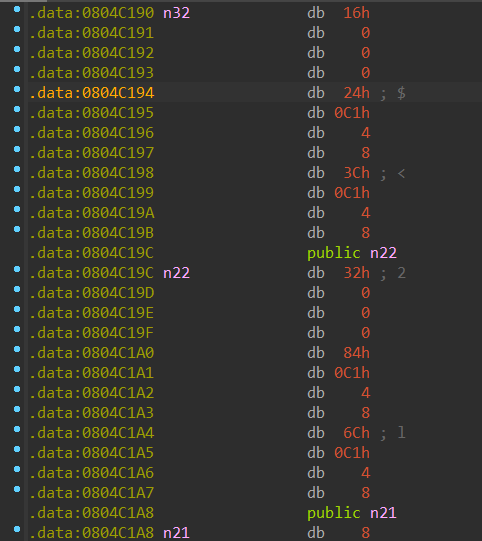
    return 0;

  return 2 \* fun7((\_DWORD \*)a1[2], a2) + 1;

}

Một hàm có sử dụng đệ quy “cũ rích” 🙃. Giá trị của v2 cũng không nhiều (0 < v2 <=1001) . Vậy đoạn này thì giống y đúc phase\_4. Chú ý tham số đầu vào thứ 1 là \_DWORD \*a1 (kiểu dữ liệu \_DWORD chiếm 4 bytes) sẽ lấy giá trị từ địa chỉ *n1* và lấy *4 byte* tại đó (tức là *0x00024*). Các tham số ta đã xác định được, nếu giá trị trả về bằng 4 là thỏa điều kiện. Tuy nhiên code trên trong hàm fun7 khó hơn mình tưởng.

Phân tích: Việc truy cập vào giá trị như **\_DWORD \*)a1[1]** làm rối đoạn code và nhảy tới nhiều ô địa chỉ như biến n1, ngoài ra còn có các biến n21, n22, … như sau nằm liền kề



Cụ thể tại n1 sẽ lưu giá trị của một số địa chỉ trỏ tới các biến ở trên (n21, n22, … ) và tương tự các biến n21, n22, … cũng lưu các địa chỉ tới các biến khác.

Ở đây điều kiện ta mong muốn lớn nhất là

* fun7(&n1, v2) = 4

Để có được giá trị 4, ta có flow chương trình sau:

Gọi giá trị của các dòng return theo thứ tự các dòng từ trên xuống dưới là ret1, ret2, ret3, ret4

Để có giá trị trả về là 4 ta suy nghĩ ra flow sau: ret2 -> ret2 -> ret4 -> ret3 -> ret1

Sau khi thực hiện đệ quy ta sẽ có giá trị trả về là từ sau lên trước (thứ tự ngược lại so với trên) là : *-1 -> 0 (+1) -> 1 (+1) -> 2 (\*2) -> 4(\*2)*

Vậy ta lần lượt pass theo các điều kiện:

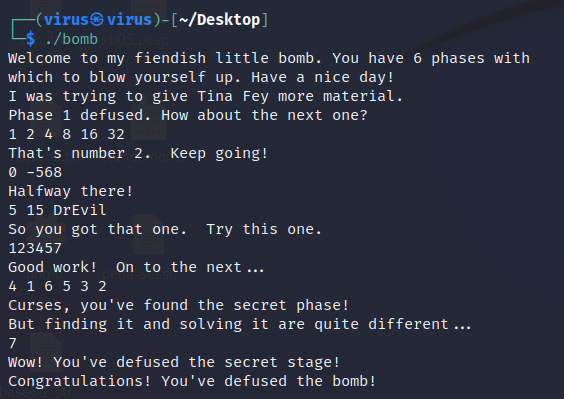
- **Mong muốn vào ret2** => Mới vào *\*a1 > a2 => a1 = 0x24* => a2 nằm trong khoảng [1, 0x24] => Lấy giá trị a1[1] , tức là n21[0]

- **Mong muốn vào ret2** => Lúc này *\*a1 > a2 => a1 = 8* => a2 nằm trong [1, 8]

- **Vào ret4** => Tương tự a2 khác 6

- **Vào ret1** => Nhảy đến các trường hợp còn lại cho đến khi *địa chỉ bị văng ra ngoài => NULL.* Vậy ta có thể chọn 1 giá trị. Ví dụ như 7 cho a2 là thỏa điều kiện

Kiểm nghiệm:



1. [↑](#footnote-ref-1)