3) Bomb: Testo e soluzione

Testo

You need to detonate the bomb.
It has several security levels.

This challenge is composed by 7 levels;
however, we ask you to solve only the first four levels.

Soluzione

Si noti che, oltre alla soluzione ufficiale e considerazioni/aggiunte/riscritture mie, si possono prendere come riferimento i link:
https://www.cnblogs.com/sinkinben/p/12397430.html
https://github.com/Ethan-Yan27/CSAPP-Labs/tree/master/yzf-Bomb%20lab (sole soluzioni)

Possiamo iniziare eseguendo il codice.

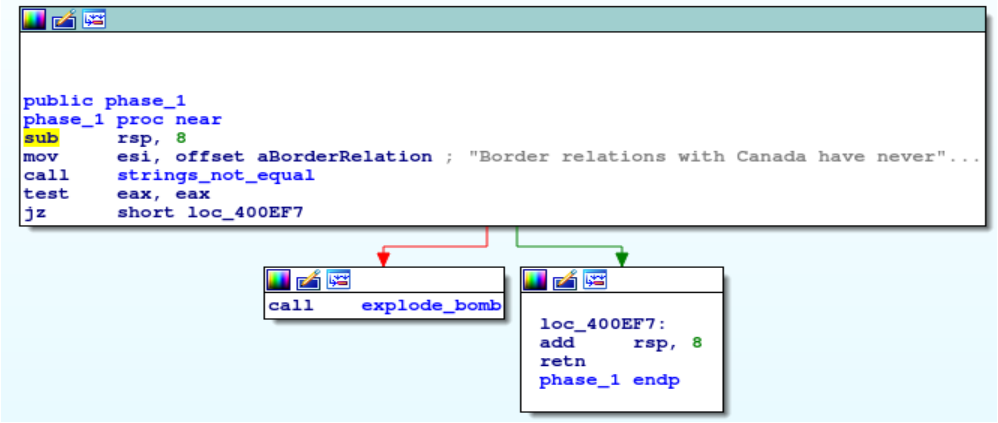
```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
123

BOOM!!!
The bomb has blown up.
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$
```

Possiamo capire quanto segue:
1. Devono essere superate 6 fasi;
2. Se fai qualcosa di sbagliato la bomba esploderà.

Speriamo che IDA possa aiutarci. In primo luogo, vediamo che ci sono 6 funzioni, una per ogni fase; quindi, la mia idea in questo momento è di studiare una fase alla volta (suppongo che non siano correlate).
Apriamo l'eseguibile in IDA e si può notare che dal main vengono chiamate una serie di funzioni.
La cosa logica da fare, dato che la bomba è composta da 6 fasi, è esaminare le singole funzioni.

Fase 1 – Si clicca tra la lista delle funzioni su *phase_1*



Mmm... "Relazioni di confine [...]" sembra sospetto. Se qualcosa va storto, la bomba esplode.

La stringa dice:

"Border relations with Canada have never been better".

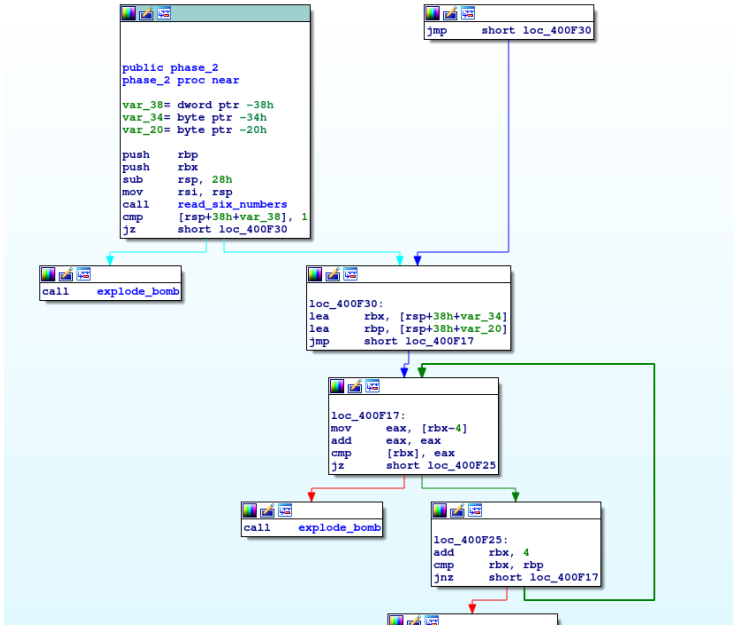
Possiamo provare a metterlo come valore.

```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
```

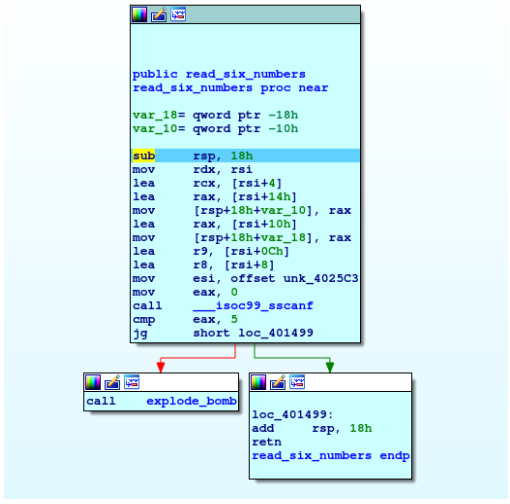
La bomba non è esplosa (buon segno).

Fase 2 - Si clicca tra la lista delle funzioni su *phase_2*

Prima di tutto, nel *main* abbiamo che il programma utilizza *read_line* per ottenere l'input dell'utente e metterlo su *rdi*. Quindi, chiama phase_2.



Innanzitutto, sta sottraendo 28 dal puntatore dello stack, il che probabilmente significa che sta preparando lo spazio per memorizzare alcune variabili. Si sposta *rsp* in *rsi* e chiama *read_six_numbers*. Facciamo doppio clic su questa funzione.



Qui è di nuovo sottraendo 0x18 da *rsp*, quindi probabilmente verranno utilizzate altre variabili.

Ora segue una serie di *mov* e *lea*, e infine una chiamata alla funzione *sscanf*, funzione che legge l'input formattato da una stringa. Essa ha il seguente formato:
*int sscanf(const char *str, const char *format, ...)*

Parametri:

1. *str* − Questa è la stringa C che la funzione elabora come origine per recuperare i dati.
2. *format* − Questa è la stringa C che contiene uno o più dei seguenti elementi: Caratteri spazi vuoti, Caratteri non spazi vuoti e Identificatori formato

In questo caso, si può pensare il *format* come segnaposto, es. %s per le stringhe, %d per i decimali, etc. Per ogni specificatore di formato nella stringa di formato che recupera i dati, è necessario specificare un argomento aggiuntivo. Se si desidera memorizzare il risultato di un'operazione *sscanf* su una variabile regolare, è necessario precedere il suo identificatore con l'operatore di riferimento, cioè un segno di e commerciale (&), come: *int n; sscanf (str,"%d",&n);*

Possiamo vedere che i registri utilizzati prima delle chiamate sono:

1. *Rdi*: che contiene la stringa inserita dall'utente
2. *Esi*: contiene una stringa caricata da un indirizzo specifico. Ispezionandolo, è, si ricava una variabile *unk_4025C3* e, facendone doppio clic, ha come contenuto *%d %d %d %d %d %d*
3. *Rdx, rcx, r8, r9* che sono pieni di indirizzi a partire da *rsi* e aggiungendo *+0x4* ogni volta.
4. Notiamo che *rsi+10* e *rsi+14* vengono spostati nello stack

In pratica, quello che succede è che si hanno esattamente 6 numeri decimali immessi, dato che *sscanf* restituisce il numero di input letti correttamente in base al formato passato *(%d %d %d %d %d %d*), ovvero sei interi. Se sono 5 o meno, la bomba esploderà.

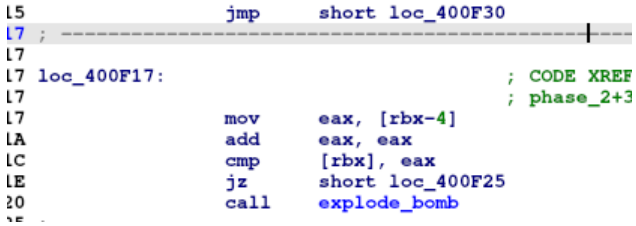
Tornando alla *fase2*, abbiamo capito che il programma si aspetta 6 numeri interi, ognuno ogni 4 byte a partire da *rsp*. L'input quindi dovrebbe essere qualcosa del genere:

x1 x2 x3 x4 x5 x6

Nel primo confronto leggiamo il primo numero *(rsp + 18h + var_18 (che è -18h) == rsp+0h)* e lo confrontiamo con 1, se True la bomba non esplode.

1 x2 x3 x4 x5 x6

Ritornando nel punto in cui la funzione è caricata, esaminiamo attentamente cosa sta succedendo:



Carichiamo il secondo numero *(rsp+4d)* nel registro *rbx* e in *rbp* sta caricando *rsp+18h (rsp+24d)* . Non sappiamo ancora cosa sia questo, ma andiamo avanti. Quindi, il blocco *loc_400F17* fondamentalmente fa il seguente confronto: (*array[i - 1]*2) == array[i]* dove:

1. *mov eax, [rbx - 4]*: *eax* contiene *array[i - 1]* poiché *rbx* contiene *array[i]* (e la struttura dell'istruzione è *dest – src*, pertanto sappiamo che il contenuto dell'array è sottratto quando si trova in *eax*
2. *add eax, eax* è equivalente ad *array[i - 1]*2*, dato che aggiungiamo lo stesso dato
3. *cmp [rbx], eax* è equivalente a (*array[i - 1]*2) == array[i]*, dato che confrontiamo il dato che ha subito una sottrazione e a cui sono stati aggiunti due volte lo stesso dato,

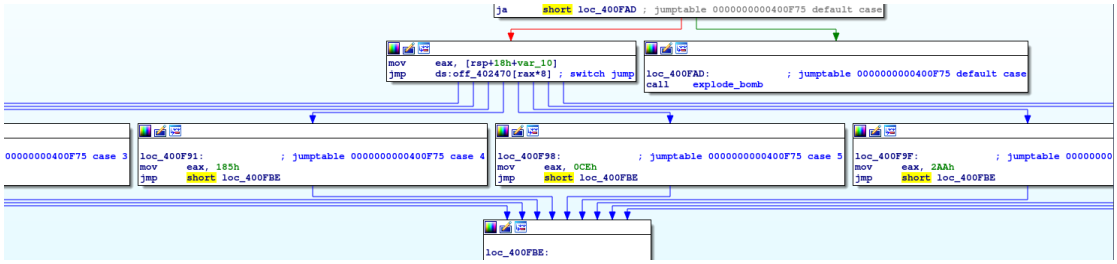
E lo fa ciclando attraverso i sei numeri (aumentando *rsi* di 4) fino a quando *rbx* è diverso da *rbp*. Poiché *rbx* incrementa di 4 ogni variabile, per leggere 6 numeri avremo *rbx+20d*, dato che parte da *rbx+0*, e quindi a *rbx+24d* ci sarebbe il 7° numero, che non esiste a causa del controllo precedente. Quindi, quando vengono confrontati 6 numeri corretti, la funzione termina.
Quindi, a partire da 1, l'input dovrebbe essere il seguente:

1 2 4 8 16 32

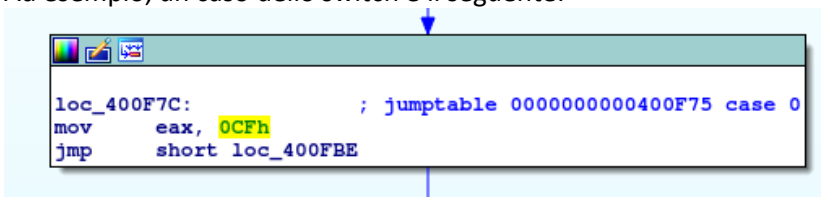
```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

Fase 3 - Si clicca tra la lista delle funzioni su *phase_3*

In questa funzione, salta subito all'occhio un'istruzione *switch*, completa di una serie di casi di scelta. In particolare, scorrendo in orizzontale, si va da *case 0* fino a *case 7* per poi avere *case 1*.

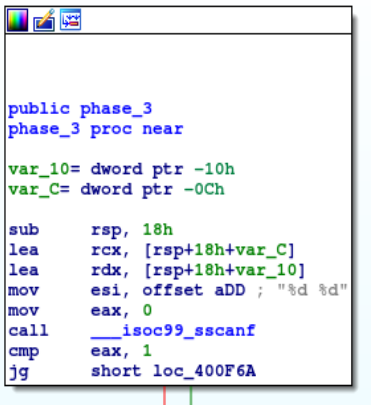


Ad esempio, un caso dello switch è il seguente:



```
loc_400F7C:                ; jumptable 0000000000400F75 case 0
mov     eax, 0CFh
jmp     short loc_400FBE
```

Ok, prima dobbiamo capire il formato di input:



```
public phase_3
phase_3 proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch

sub     rsp, 18h
lea     rcx, [rsp+18h+var_C]
lea     rdx, [rsp+18h+var_10]
mov     esi, offset aDD ; "%d %d"
mov     eax, 0
call    ___isoc99_sscanf
cmp     eax, 1
jg      short loc_400F6A
```

Dal commento "%d %d" immagino che stiamo parlando di due interi. Nell'offset con "18h + var_10" memorizziamo il primo numero, in "18h+var_C" il secondo.

I due valori memorizzati in "*18h + var_10*" e "*18h + var_C*" vengono utilizzati come segue:
1. "*18h + var_10*": viene utilizzato per scegliere lo switch case

2. "*18h+var_C*": viene confrontato con un valore costante salvato nel caso specifico dello switch (ad esempio, *0CFh, 185h, 0CEh*); si può quindi immaginare C stia per "costante"

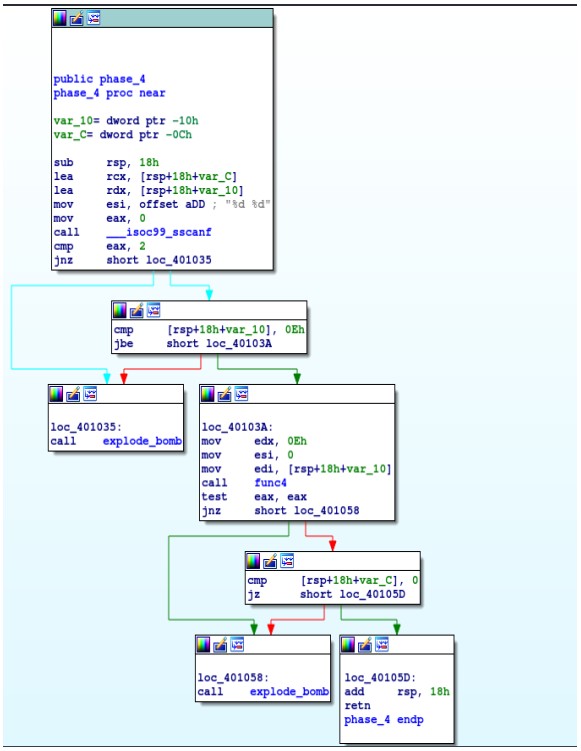
Pertanto, se scegliamo il caso "0" dello switch, possiamo vedere che il valore costante utilizzato per il confronto è *0CFh*, che equivale a 207. Proviamo con il seguente input:

0 207

```
(base) pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 207
Halfway there!
```

Alternativamente, basta scegliere uno qualsiasi degli switch case; la condizione è scritta in modo tale che, dando un qualsiasi numero da 0 a 7 e uno dei numeri del confronto dei registri tradotti da esadecimale a decimale, si passa senza problemi.
Esempio caso 2, con 2C3 che in decimale equivale a 707. Si inserirà *2 707* e si passa alla fase 4.

Fase 4 - Si clicca tra la lista delle funzioni su *phase_4*



Come abbiamo fatto in precedenza, possiamo notare che dobbiamo indovinare due numeri, come si vede da %d %d. Possiamo vedere che:

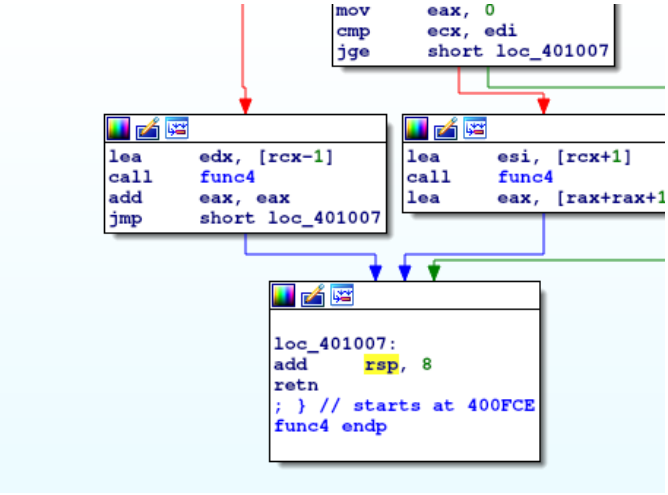
1. *[18h + var_10]* deve essere <= *0Eh* (cioè 15), dato che si ha l'istruzione *cmp*;
2. *[18h + var_C]* deve essere *0*, dato che abbiamo capito che *var_C* deve essere valore costante e dopo si ha una *mov* che richiede almeno uno 0 come valore.

In *loc_40103A*, viene chiamato *func4*, con *var_10* come parametro. Il suo valore di ritorno è testato contro zero (nel test di istruzioni *eax, eax*). Quindi, vorremmo avere un valore di ritorno per func4 pari a zero.
Dobbiamo capire cosa fa la funzione *func4* presente nella lista delle funzioni a sinistra.

```
public func4
func4 proc near
; __unwind {
sub     rsp, 8
mov     eax, edx
sub     eax, esi
mov     ecx, eax
shr     ecx, 1Fh
add     eax, ecx
sar     eax, 1
lea     ecx, [rax+rsi]
cmp     ecx, edi
jle     short loc_400FF2
```

Al di là delle singole operazioni di sottrazione (sub), shift logico a destra (shr), shift aritmetico a destra (sar), Quello che dovresti notare è che:

1. È ricorsiva (dopo le operazioni, tende a richiamare *func4* partendo dall'indirizzo 400FCE)



Si nota infatti che l'indirizzo finale delle chiamate viene anche dopo l'istruzione finale di *func4*.

2. Se andiamo sul caso base (che restituisce 0) evitiamo tutte le altre chiamate e quindi, dobbiamo solo capire come viene definito il caso base.

Dati i due numeri interi di prima e un numero che deve essere 0, si sa che la funzione ricorre e si ha quando si ha un numero costante; questo può essere quello del registro *esi*, quindi *0*. La struttura della chiamata di *func4(edx=0xe, esi=0x0, edi=x)*, e questa chiamata di funzione passa i parametri per registro.
Le condizioni per il passaggio di phase_4 sono l'input x y per far sì che func(0xe, 0x0, x) restituisca 0 e che x sia inferiore a 15 e y sia 0.
Essendoci tre registri principali, possiamo quindi inserire una cosa del tipo *7 0* e risolvere la fase.

```
pajola@pajola-XPS-13-9370:~/Documents/CyberChallenges/reverse/8_bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 207
Halfway there!
7 0
So you got that one.  Try this one.
```

Il codice di func4 può essere tradotto in questo modo in C:

```
int func4(int a, int b, int x)
{
   x;                    // edi
   int t1 = a - b;           // eax
```

```
   int t2 = ((unsigned int)t1) >> 31; // ecx
   t1 = (t1 + t2) >> 1;
   t2 = t1 + b;
   if (t2 <= x)
   {
      t1 = 0;
      if (t2 < x)
         return 2 * func4(a, t2 + 1, x) + 1;
      return t1;
   }
   else
      return 2 * func4(t2 - 1, b, x);
}
```

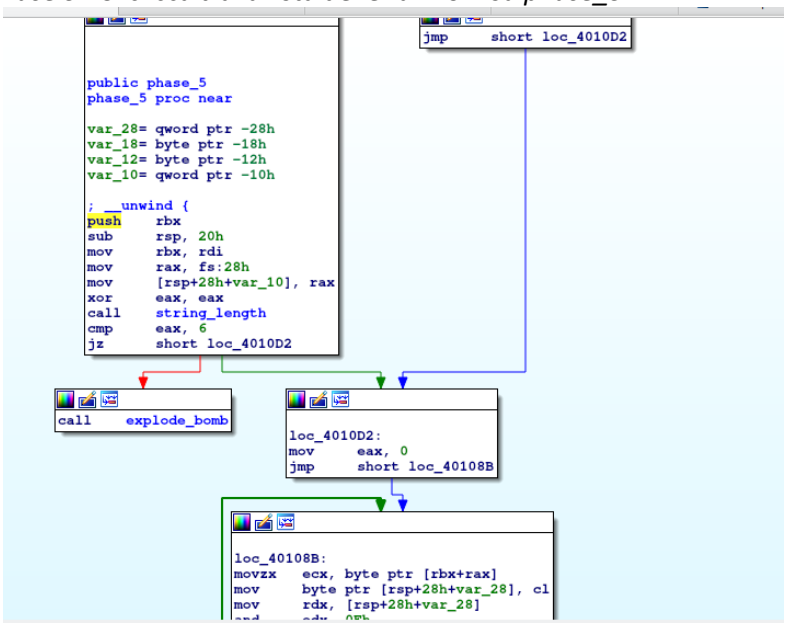
Dobbiamo prestare attenzione a questa istruzione*: 400fd8: shr $0x1f,%ecx*. Il termine shr implica un'operazione su un tipo unsigned (si legga il codice C sopra).
Pertanto, possiamo trovare x tramite loop:

```
int main()
{
   int x = 0;
   for (x = 0; x <= 15; x++)
   {
      int t = func4(0xe, 0x0, x);
      if (t == 0)
         printf("%d ", x);
   }
}
```

Con risultato:
(x,y) = (0,0), (1,0), (3,0), (7,0)

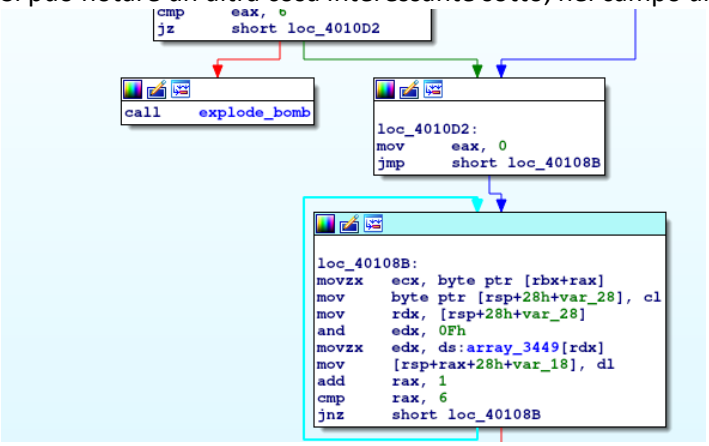
Loro consigliano di fermarsi alla 4; io provo anche le altre.

Fase 5 - Si clicca tra la lista delle funzioni su *phase_5*



Questo pezzo controlla semplicemente che la lunghezza della stringa sia 6.

Si può notare un'altra cosa interessante sotto, nel campo *array_3449* (sx) e, facendo doppio clic (dx):



Fondamentalmente, il programma sta creando una hash table nel seguente modo:

hex index:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
hash str :	m	a	d	u	i	e	r	s	n	f	o	t	v	b	y	l

La corrispondenza è quindi:
`f(x) = hash_str[x & 0xf]`

`flyers` corrisponde ai caratteri `9, F, E, 5, 6, 7` nella tabella hash, perciò, `str[i] & 0xf (i=0,...,5)` dovrebbe essere `9, F, E, 5, 6, 7`. Possiamo controllare la tabella ASCII per ottenere i caratteri (componendo come struttura 0x3[carattere], tipo 0x3(9), etc.):
Conversione con https://www.rapidtables.com/convert/number/hex-to-ascii.html non mettendo 0x davanti:
0x39 → 9
0x3F → ?
0x3E → >
0x35 → 5
0x36 → 6
0x37 → 7

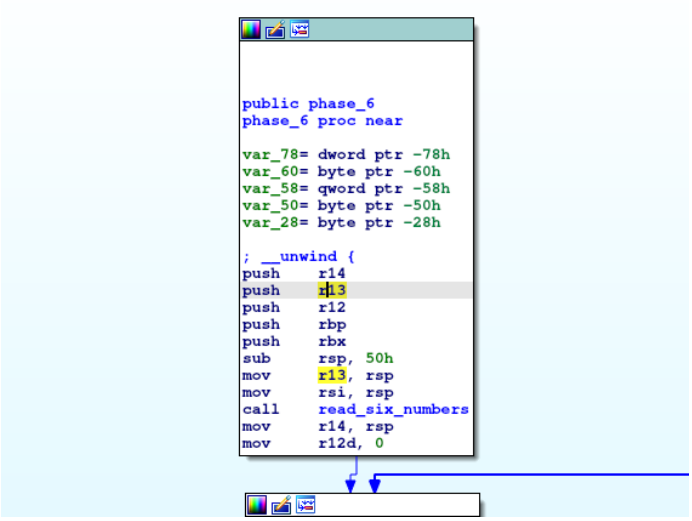
A livello di codice C, potrebbe equivalere a:

```
void phase5(const char *input)
{
   const char *hash_str = "maduiersnfotvbyl";
   char array[7];
   int i = 0;
   for (i = 0; i < 6; i++)
      array[i] = hash_str[input[i] & 0xf];
   if (strings_not_equal(array, "flyers"))
      explode();
}
```

Pertanto, un input come riportato sopra, quindi: `9?>567` basterà per farci passare questa fase.

Nota: IDA Pro, con tasto F5, fa vedere tutto il codice in un colpo solo.

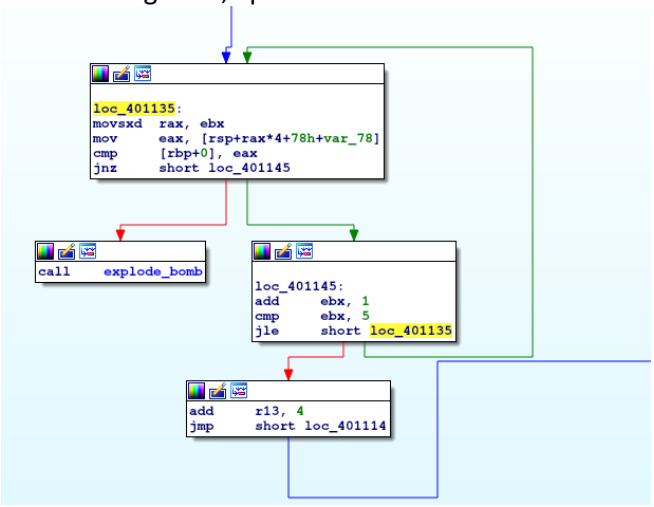
Fase 6 - Si clicca tra la lista delle funzioni su *phase_6*



Analizziamo attentamente cosa fa tutto il codice; fondamentalmente, esegue una push di tre dati in sequenza decrescente, (14-13-12), dopodiché viene chiamata la funzione *read_six_numbers*, stessa esaminata prima. Sapremo quindi che la funzione chiama esattamente 6 numeri e, presumibilmente i primi 3, sono in sequenza decrescente.
Continuando con l'analisi, si nota che sottraiamo *1* da *r13* e compariamo a 5; se <= 5 allora avanziamo e consideriamo che si aggiunge 1, comparando a 6. Se questo non succede, la bomba esplode.
Quello che capiamo è che:

- ogni numero deve essere <=6 e i primi numeri devono essere <= 4

Successivamente, vengono fatti vari calcoli/spostamenti di registri di cui non è utile fornire i dettagli, essendo lunghetta; i primi 3 numeri devono essere <= 5.



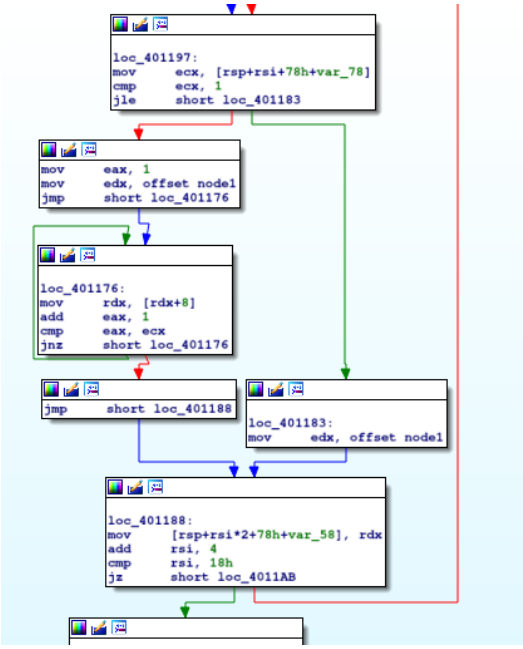
Si consideri una spiegazione estesa presente al link sotto, alla quale mi aggancio per spiegare il resto del funzionamento del codice:
https://yieldnull-com.translate.goog/blog/ed9209f92effdad6f9fe997fdfc120ecc89ea212/?_x_tr_sl=auto&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp

https://yuhan2001-github-io.translate.goog/2021/04/01/bomblab/?_x_tr_sl=auto&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp

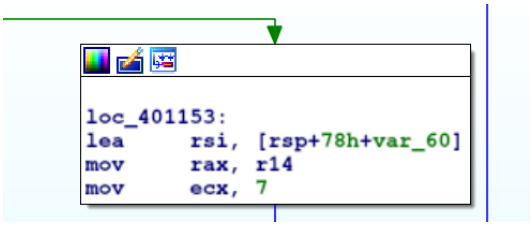
Il primo pezzo può essere tradotto in C++ come:

```
for(int i=0;i<=5;i++)
{
        if(a[i]-1>5||a[i]-1<0)
        explode_bomb();
    else
        for(int j=i+1;j<=5;j++)
            if(a[j]==a[i])
               explode_bomb();
}
```

Il successivo blocco di codice controlla che i numeri messi usino 7 per operare con i registri; ci sono una serie di spostamenti successivi, come evidenziato dal seguente disastro:



Ci son vari spostamenti che fondamentalmente sistemano il formato dei registri tra 4 e 8 byte; la cosa utile da sapere è *mov 7* del blocco sovrastante.

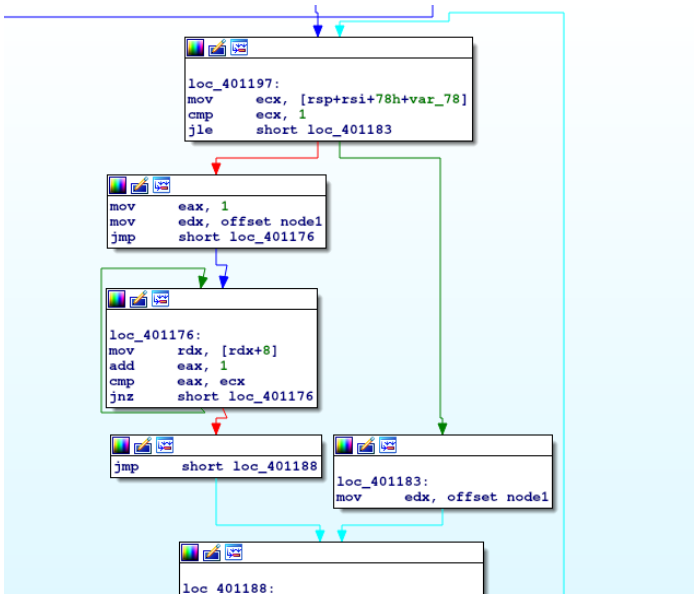


Tradotto in *dump* esadecimale (quanto contiene attualmente la memoria):

```
0x08048dd4 <+82>:   lea    0x10(%esp),%eax            ;eax=(0x10+esp)
0x08048dd8 <+86>:   lea    0x28(%esp),%ebx            ;ebx=(0x28+esp)
0x08048ddc <+90>:   mov    $0x7,%ecx                  ;ecx=7
0x08048de1 <+95>:   mov    %ecx,%edx                  ;edx=ecx=7
0x08048de3 <+97>:   sub    (%eax),%edx                ;edx=edx-M[eax]=7-M[eax]
0x08048de5 <+99>:   mov    %edx,(%eax)                ;M[eax]=edx
0x08048de7 <+101>:  add    $0x4,%eax                  ;eax=eax+4
0x08048dea <+104>:  cmp    %ebx,%eax
0x08048dec <+106>:  jne    0x8048de1 <phase_6+95>     ;若eax!=ebx, 则回跳至<phase_6+95>
```

A livello di codice, si traduce come segue

```
for(int i=0;a[i]!=a[6];i++)
{
    a[i]=7-a[i];
}
```



In dump corrisponde al seguente codice:

```
   0x08048e22 <+160>:   mov    0x28(%esp),%ebx          ;ebx=M[0x28+esp]    --node[0]的地
   0x08048e26 <+164>:   mov    0x2c(%esp),%eax          ;eax=M[0x2c+esp]    --node[1]的地
   0x08048e2a <+168>:   mov    %eax,0x8(%ebx)           ;M[ebx+0x8]=eax         --node[0]
                                                        ;以下至<+196>都为连接结点、形成链表
   0x08048e2d <+171>:   mov    0x30(%esp),%edx
   0x08048e31 <+175>:   mov    %edx,0x8(%eax)
   0x08048e34 <+178>:   mov    0x34(%esp),%eax
   0x08048e38 <+182>:   mov    %eax,0x8(%edx)
   0x08048e3b <+185>:   mov    0x38(%esp),%edx
   0x08048e3f <+189>:   mov    %edx,0x8(%eax)
   0x08048e42 <+192>:   mov    0x3c(%esp),%eax
   0x08048e46 <+196>:   mov    %eax,0x8(%edx)
   0x08048e49 <+199>:   movl   $0x0,0x8(%eax)           ;M[0x8+eax]=0, node[5].next=0(空)
   0x08048e50 <+206>:   mov    $0x5,%esi                ;esi=5
   0x08048e55 <+211>:   mov    0x8(%ebx),%eax           ;eax=M[0x8+ebx]     --下一节点的地
   0x08048e58 <+214>:   mov    (%eax),%edx              ;edx=M[eax]     --下一结点的值
   0x08048e5a <+216>:   cmp    %edx,(%ebx)
   0x08048e5c <+218>:   jge    0x8048e63 <phase_6+225>
                        ;若M[ebx]>=edx，即某结点的值不小于其后一个结点的值,则不会爆炸
   0x08048e5e <+220>:   call   0x80490e6 <explode_bomb>    ;否则爆炸
   0x08048e63 <+225>:   mov    0x8(%ebx),%ebx           ;ebx=M[ebx+0x8]     --下一个结点的
   0x08048e66 <+228>:   sub    $0x1,%esi                ;esi=esi-1
   0x08048e69 <+231>:   jne    0x8048e55 <phase_6+211> ;当esi!=0时，回跳至<phase_6+211>,
                                                        ;故执行<+211>到<+231>的次数为6。
   0x08048e6b <+233>:   add    $0x44,%esp               ;恢复栈帧，过关。
   0x08048e6e <+236>:   pop    %ebx
   0x08048e6f <+237>:   pop    %esi
  0x08048e70 <+238>:    ret
End of assembler dump.
```

L’ultima parte richiede, usando una serie di 6 numeri che, sottratti successivamente di 1 (*sub ebp, 1*)

- Si può osservare che la struttura è composta da un valore (val), un numero (id) e l'indirizzo del nodo successivo (*next), formando una struttura a lista concatenata.

- Pertanto, in base al valore dell'array a[], riordina i 6 nodi del nodo: lascia che l'a[i-1]esimo nodo originale diventi nodo[i-1]. dove a[i-1]=(7-l'i-esimo valore dell'input).

L'ordinamento segue questa logica (https://yieldnull-com.translate.goog/blog/ed9209f92effdad6f9fe997fdfc120ecc89ea212/?_x_tr_sl=auto&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=wapp)

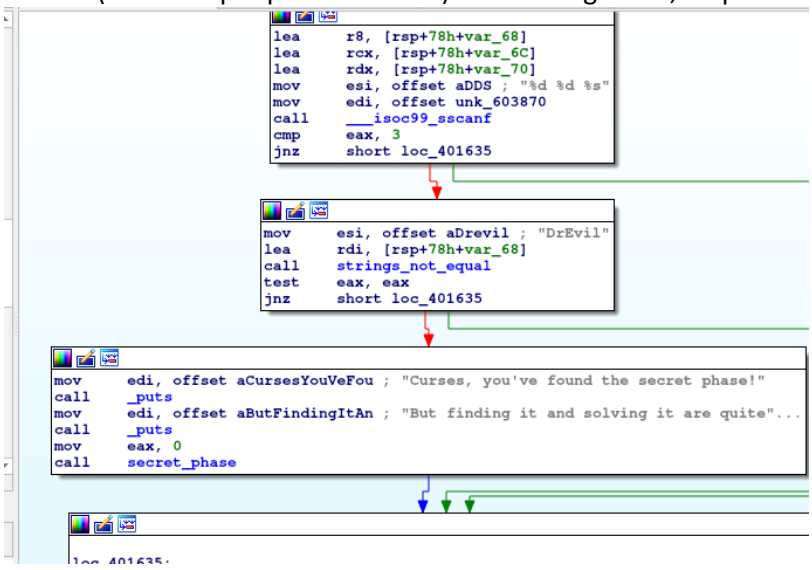
```
/* ordiniamo la lista originale:
*
* Value Index Calculation
* 0x39c 2 7 - a - 1 = 2 => a = 4
* 0x2b3 3 7 - b - 1 = 3 => b = 3
* 0x1dd 4 7 - c - 1 = 4 => c = 2
* 0x1bb 5 7 - d - 1 = 5 => d = 1
* 0x14c 0 7 - e - 1 = 0 => e = 6
* 0x0a8 1 7 - f - 1 = 1 => f = 5
* */
```

Phase_6 richiede quindi l'inserimento di numeri in modo decrescente, seguendo l'ordine $(7 - x_i - 1)$
Usando il comando *p/x* (print hexadecimal) in *gdb*, notiamo che le variabili che gli indirizzi delle variabili corrispondono al seguente mapping:

- p/x *(0x804c13c)@3
- $1 = {0xf3, 0x6, 0x0)

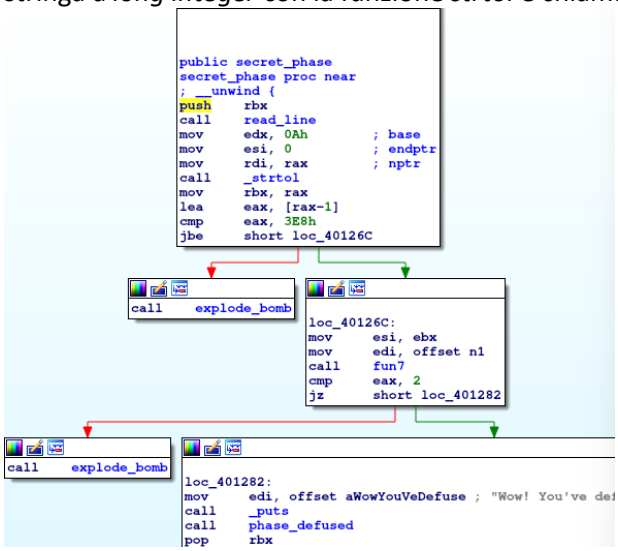
Così via fino al 6, ottenendo come struttura:
1|2|3|4|5|6
Valore originale: |0xf3|0x252|0x2fb|0x376|0xdf|0xea|
Destinazione: |4|3|2|1|6|5|

Secret_phase → Fase segreta (Ultima fase, dopo aver sconfitto tutti i livelli)
Si può notare dalla funzione *phase_defused* che quando il numero di stringhe immesse è maggiore di 6, si arriva alla fase segreta, inserendo però un input fisso (stringa), nel livello 4, che ricordiamo leggeva 2 numeri (bastevoli per passare la fase) e una stringa fissa, in questo caso come si vede *DrEvil*.



Inserendo ad esempio *0 0 DrEvil* su quella fase, si passa senza problemi alla fase segreta.

Quello che viene fatto in questa funzione è leggere l'input, eseguire spostamenti di registri, convertire una stringa a *long integer* con la funzione *strtol* e chiamare *fun7*.



La funzione *fun7* formatta inizialmente i registri e confronta con *test* (che serve a fare l'and bit a bit dei registri) il contenuto di *rdi*; se sono uguali, mette tutto a 0.
Altrimenti, si nota che la funzione procede comparando due registri e:

- si ha una moltiplicazione per 2 (data da *add eax eax*)
- viene aggiunto 1 ad *eax*
- viene chiamata *fun7* ricorsivamente, fino a quando la chiamata di *fun7* è 0x8, che corrisponde ad 8.

Concludiamo che il flusso della funzione debba avere valore:
`2 * (1 + 2 * fun(7))` che sarà `2 * (1 + 2 * 0)` per effetto del ramo *else* che azzera.
Avremo quindi che inserendo *22* tutto funziona.

Input di tutte le fasi:

1) Border relations with Canada have never been better.
2) 1 2 4 8 16 32
3) 7 327
4) 0 0 DrEvil
5) 9?>567
6) 4 3 2 1 6 5
7) 22

Welcome to my fiendish little bomb. You have 6 phases with which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
Border relations with Canada have never been better.
That's number 2. Keep going!
7 327
Halfway there!
0 0 DrEvil
So you got that one. Try this one.
9?>567
Good work! On to the next...
4 3 2 1 6 5
Curses, you've found the secret phase!
But finding it and solving it are quite different...
22
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!