
AHAU CTF 2025

cr4cknuch0

Tony Palma

2025-09-26

Índice

1	AHAU CTF 2025 - cr4cknuch0	1
1.1	Infocard	1
1.2	Solución	2
1.2.1	Reconocimiento	2
1.2.2	Análisis estático con Ghidra	5
1.2.3	Función <code>main.validation</code>	11
1.2.4	Función <code>main.part3</code>	15
1.2.5	Función <code>main.decodeHex</code>	17
1.2.6	De regreso a <code>main.part3</code>	18
1.2.7	Función <code>main.part4</code>	19
1.2.8	Construcción del solver	25
1.2.9	Final script	26
1.3	Flag	27

1 AHAU CTF 2025 - cr4cknuch0

1.1 Infocard

Nombre	cr4cknuch0
Descripción	crackme
Categoría	reversing
Puntos	300
Dificultad	Intermedio

1.2 Solución

1.2.1 Reconocimiento

Este reto tiene dos formas principales de solucionarse, una es mediante fuerza bruta usando angr o cualquier otra herramienta parecida (inclusive un script que pruebe todas las combinaciones), y la otra consiste en técnicas de análisis estático y dinámico donde se comprenda el código y se elabore un programa para crackear el binario y encontrar el texto protegido (flag). En este writeup, se buscará un enfoque de análisis estático para explorar el binario y comprender los mecanismos de protección.

Primero comenzamos por identificar tanta información como podamos del binario antes de ir a decompilarlo. Lo primero sera usar file o readelf para extraer el signature del binario:

```
xbytemx@holi:~$ file cr4cknuch0
cr4cknuch0: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, no section header
xbytemx@holi:~$ strings cr4cknuch0 | grep -iE 'ahau|flag'
xbytemx@holi:~$ |
```

Figura 1.1: Identificando el binario con file y strings

La salida de file nos devuelve algo muy extraño, el binario no tiene secciones. Al no tener secciones, la mayoría de las herramientas fallarán al no poder identificar la estructura del binario. Esta es una señal de que probablemente el binario se encuentre protegido, es decir, una manera de impedir que pueda ser analizado por ghidra, ida o radare2 directamente. Pasaremos a seguir enumerando el binario con DIE (DetectItEasy), el cual nos devuelve lo siguiente desde su version por consola:

```
xbytemx@holi:~$ ~/ws/tools/die/diec cr4cknuch0
ELF64
    Packer: UPX(4.24)[LZMA,brute]

xbytemx@holi:~$ █
```

Figura 1.2: Salida de diec.sh

DIE nos indica que este binario se encuentra protegido (comprimido) por un packer llamado UPX, proceso que podemos realizar en reverso y desempaquetar el binario utilizando el mismo programa UPX:

```
xbytemx@holi:~$ upx -h
                Ultimate Packer for eXecutables
                Copyright (C) 1996 - 2024
UPX 4.2.4      Markus Oberhumer, Laszlo Molnar & John Reiser   May 9th 2024

Usage: upx [-123456789dlthVL] [-qvfk] [-o file] file..

Commands:
  -1      compress faster                      -9      compress better
--best    compress best (can be slow for big files)
-d        decompress                          -l      list compressed file
-t        test compressed file                -V      display version number
-h        give this help                      -L      display software license
```

Figura 1.3: UPX -help

Ahora ejecutamos el desempaquetado y volvemos a ejecutar file en el binario descomprimido:

```
xbytemx@holi:~$ upx -d cr4cknuch0 -o cr4cknuch0.unpack
                Ultimate Packer for eXecutables
                Copyright (C) 1996 - 2024
UPX 4.2.4      Markus Oberhumer, Laszlo Molnar & John Reiser   May 9th 2024

      File size      Ratio      Format      Name
      -----
2042326 ← 1119116  54.80%  linux/amd64  cr4cknuch0.unpack

Unpacked 1 file.
xbytemx@holi:~$ file cr4cknuch0.unpack
cr4cknuch0.unpack: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, Go BuildID=a0uxKp7-NYmv8eyzW
xGC/FCD2tIZenq0GRsi0WxtK/VBZCaDS70mqY75vS0_wI/ISMEYr9CZ00SvmLuDgc_, with debug_info, not stripped
xbytemx@holi:~$
```

Figura 1.4: upx -d cr4cknuch0 -o cr4cknuch0.unpack

Como podemos ver, el binario es un ejecutable de ELF64 para x86_64, escrito en Go que contiene símbolos de debuggeo (nuestra mayor pista acerca del lenguaje en el cual fue escrito) y que también contiene todas sus dependencias (no está enlazado a otras bibliotecas).

Aquí también volvemos a ejecutar strings buscando directo la flag dentro del binario, pero por lo que vemos, nos devuelve dos ocurrencias del string ahau:

```
xbytemx@holi:~$ strings cr4cknuch0.unpack | egrep --color -i 'ahau'
m= sp= sp: lr: fp= gp= mp=) m= in /etc on 3125boolint8uintchanfunccallkindermssse3avx2bmi1bmi2AHAU{eazzy<nil>wr
iteclosefalseErrordefersewepetestRtestWexecWexecRschedhchansudoggscanmheaptracepanicsleep cnt=gcing MB, got= ...
stopTheWorld: not stopped (status != _Pgctest)runtime: name offset base pointer out of rangerruntime: type offset base p
ointer out of rangerruntime: text offset base pointer out of rangeunexpected error wrapping poll.ErrFileClosing: slice b
ounds out of range [::%x] with length %yP has cached GC work at end of mark terminationfailed to acquire lock to start
a GC transitionfinishGCTransition called without starting one?tried to sleep scavenger from another goroutineracy sudog
adjustment due to parking on channelfunction symbol table not sorted by PC offset: attempted to trace a bad status for
a goroutineattempting to link in too many shared librariesreflect.Value.Bytes of unaddressable byte arrayerror: wrong
flag format: %s, expected AHAU{...}slice bounds out of range [:%x] with capacity %yruntime: cannot map pages in arena a
ddress spacestrconv: illegal AppendFloat/FormatFloat bitSizenot enough significant bits after mult64bitPow10slice bound
s out of range [::%x] with capacity %yinvalid memory address or nil pointer dereferencepanicwrap: unexpected string aft
er package name: s.allocCount != s.nelems && freeIndex = s.nelemsdelayed zeroing on data that may contain pointersswee
per left outstanding across sweep generationsfully empty unfreed span set block found in resetcasgstatus: waiting for G
waiting but is Grunnableinvalid or incomplete multibyte or wide characternot enough significant bits after mult128bitPo
w103625382146e14c8558131e815edf18531a6d701d46f1688d224071510a7150584a4e0d5c0c577150095d63491d4e4b414f44mallocgc called
with gcphase = _GCmarkterminationrecursive call during initialization - linker skewattempt to execute system stack cod
e on user stacklimiterEvent.stop: invalid limiter event type foundpotentially overlapping in-use allocations detectedfa
tal: systemstack called from unexpected goroutinemallocgc called without a P or outside bootstrappingruntime: cannot di
sable permissions in address spaceruntime.SetFinalizer: pointer not in allocated blockruntime: use of FixAlloc_Alloc be
fore FixAlloc_Init
xbytemx@holi:~$
```

Figura 1.5: strings cr4cknuch0 | egrep -color -i 'ahau'

Vamos a ejecutar el binario tratando de usar lo que encontramos y buscar donde o como se usa algún de las partes que encontramos:

```
xbytemx@holi:~$ ./cr4cknuch0.unpack
2025/09/29 20:22:32 Enter the flag:
AHAU{...}
2025/09/29 20:22:37 Wrong length
xbytemx@holi:~$ ./cr4cknuch0.unpack
2025/09/29 20:22:40 Enter the flag:
AHAU{eazzy
2025/09/29 20:22:47 error: wrong flag format: AHAU{eazzy, expected AHAU{...}
xbytemx@holi:~$
```

Figura 1.6: First Try

Como podemos ver tenemos dos tipos de mensajes por parte del binario. El primer mensaje que tenemos de respuesta es de tamaño incorrecto y el segundo de formato incorrecto. Podemos deducir de esta primera ejecución, que el formato de la flag es AHAU{ más algún texto de tamaño N y termina en }.

Vamos a intentar calcular el tamaño de la flag con un pequeño script en bash+python:

```
1 for LEN in {1..100}; do python3 -c 'print("AHAU{" + "A"*$LEN + "}")' | ./cr4cknuch0.unpack
  ↪ 2>&1 | grep -vE 'Wrong length|Enter the flag' && echo La longitud es de $((($LEN+6))
  ↪ \($LEN mas 6 del formato de la bandera\);done
```

```
xbytemx@holi:~$ for LEN in {1..100}; do python3 -c 'print("AHAU{" + "A"*$LEN + "}")' | ./cr4cknuch0.unpack 2>&1 | gre  
p -vE 'Wrong length|Enter the flag' && echo La longitud es de $((($LEN+6)) \($LEN mas 6 del formato de la bandera));done  
2025/09/29 20:30:19 Wrong password  
La longitud es de 50 (44 mas 6 del formato de la bandera)  
xbytemx@holi:~$
```

Figura 1.7: Ingresando valores para entender que acepta el programa

Como podemos ver, la contraseña es la flag, y su longitud total es de 50 caracteres. Ahora necesitamos encontrar esos 44 caracteres restantes. También llegamos a una condición donde la flag es evaluada y obtuvimos el mensaje de que lo ingresado es incorrecto (Wrong password), por lo que empezaremos por analizar el binario estáticamente.

1.2.2 Análisis estático con Ghidra

Primero carguemos el binario en Ghidra para decompilarlo:

1. Creamos un nuevo proyecto llamado cr4cknuch0
2. Importamos el archivo cr4cknuch0.unpack dentro del proyecto

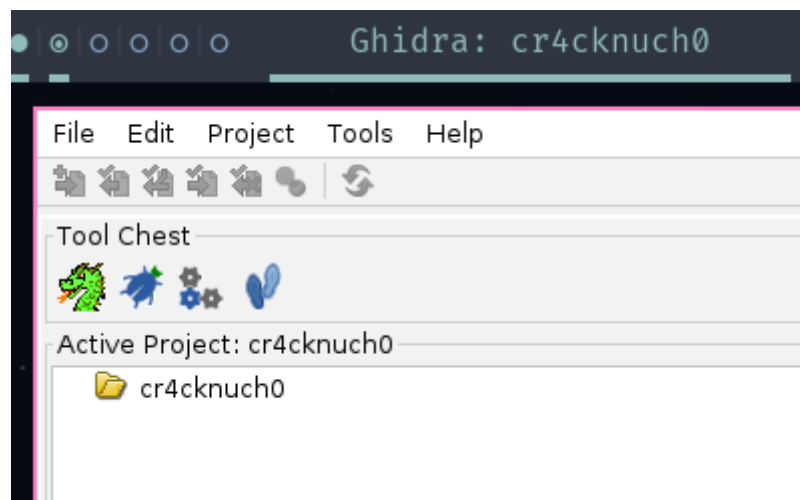


Figura 1.8: Proyecto de Ghidra

3. Seleccionamos el lenguaje de origen desde la lista justo al importarlo

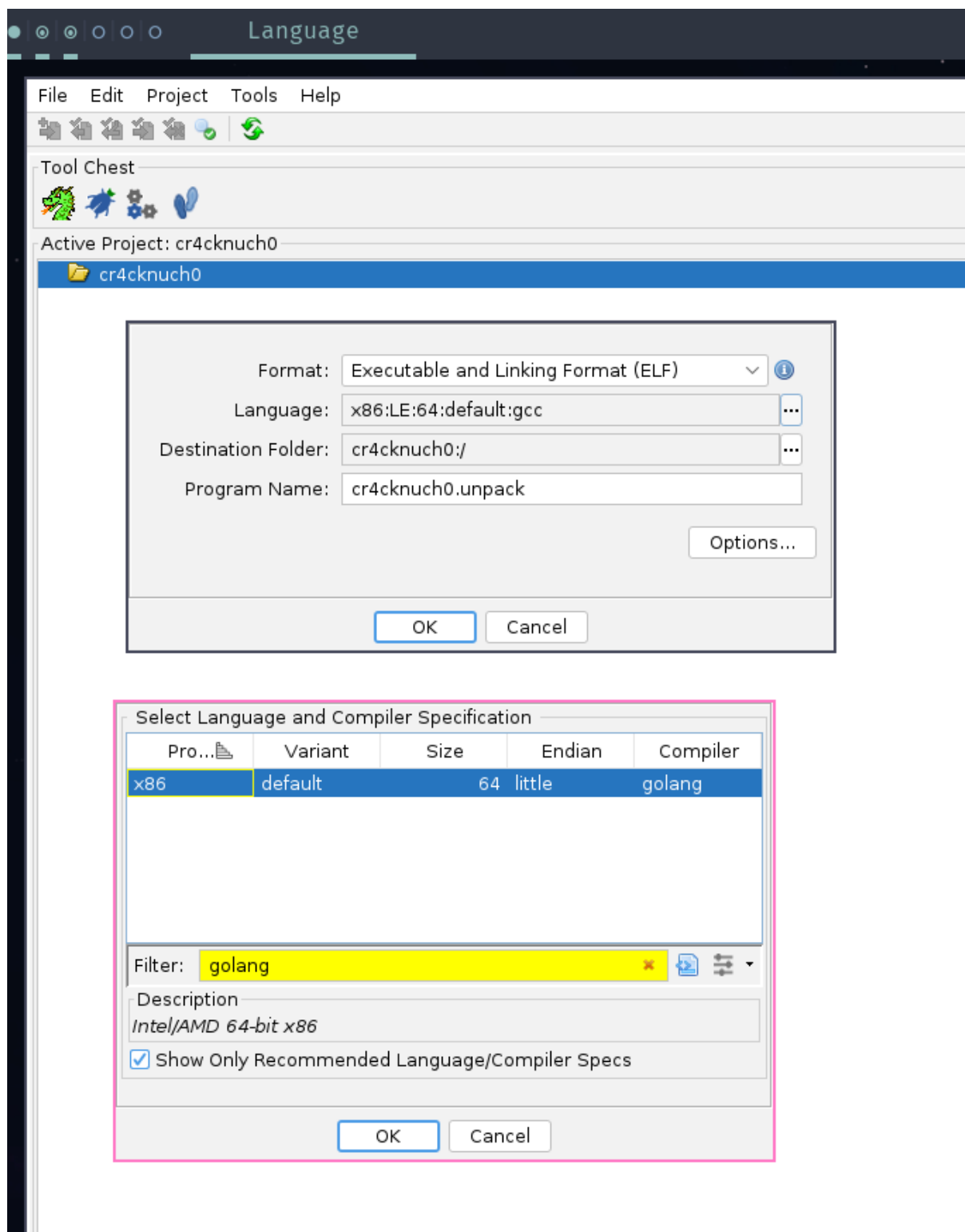


Figura 1.9: Seleccionamos golang de la lista

4. Analizamos con CodeBrowser y seleccionamos las opciones por defecto

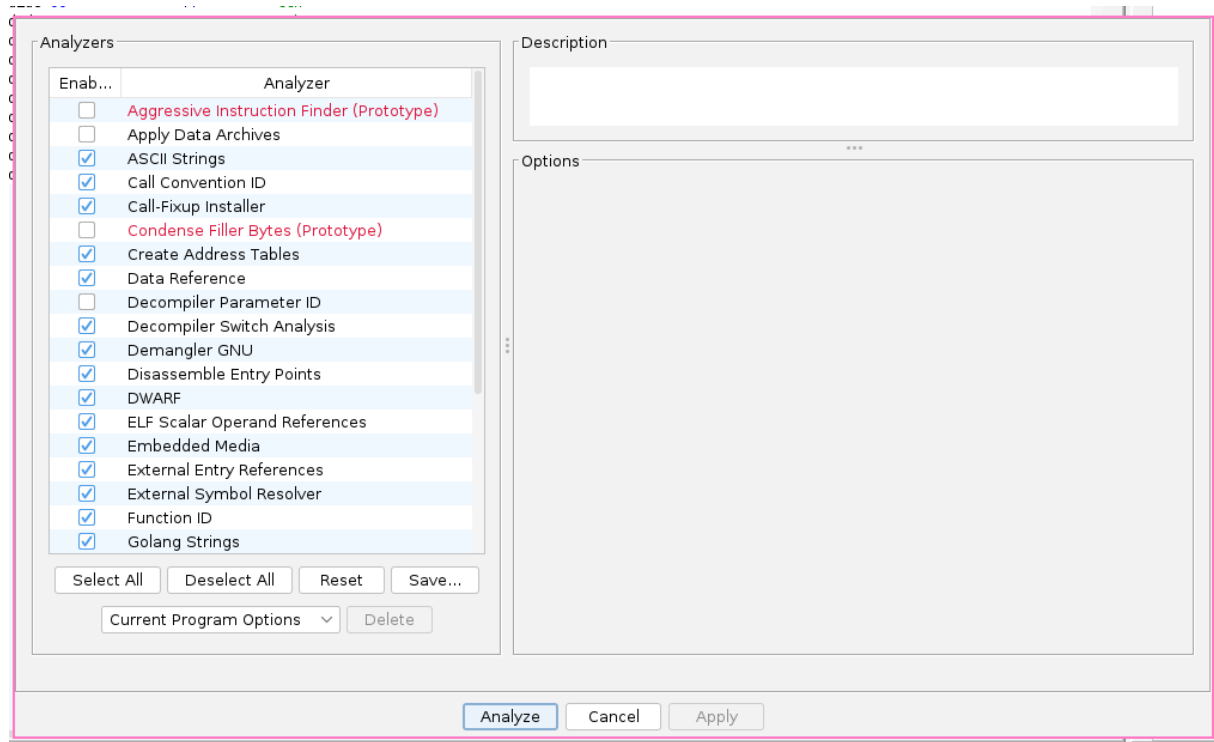


Figura 1.10: Opciones por defecto

5. Desde el menu de Search elegimos For strings y cargamos las opciones por defecto (mínimo 5 y terminación en null). Elegimos All Blocks para buscar en todo el binario.
6. Buscamos el string de Wrong password que es nuestra condición de fail

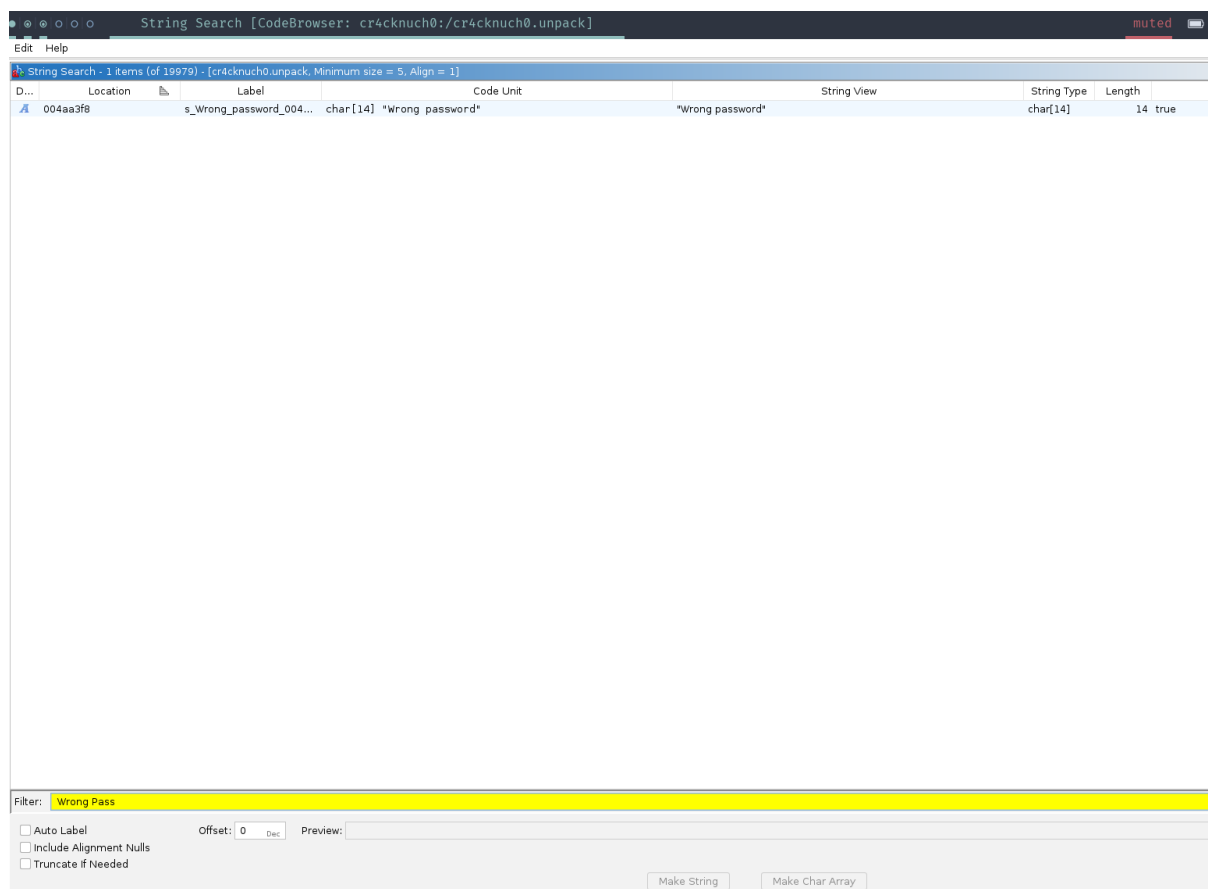


Figura 1.11: Búsqueda en todos los strings del binario

- Después guardamos la dirección en memoria donde se encuentra el string (0x004aa3f8), cerramos la búsqueda por strings y en la pantalla principal de CodeBrowser.

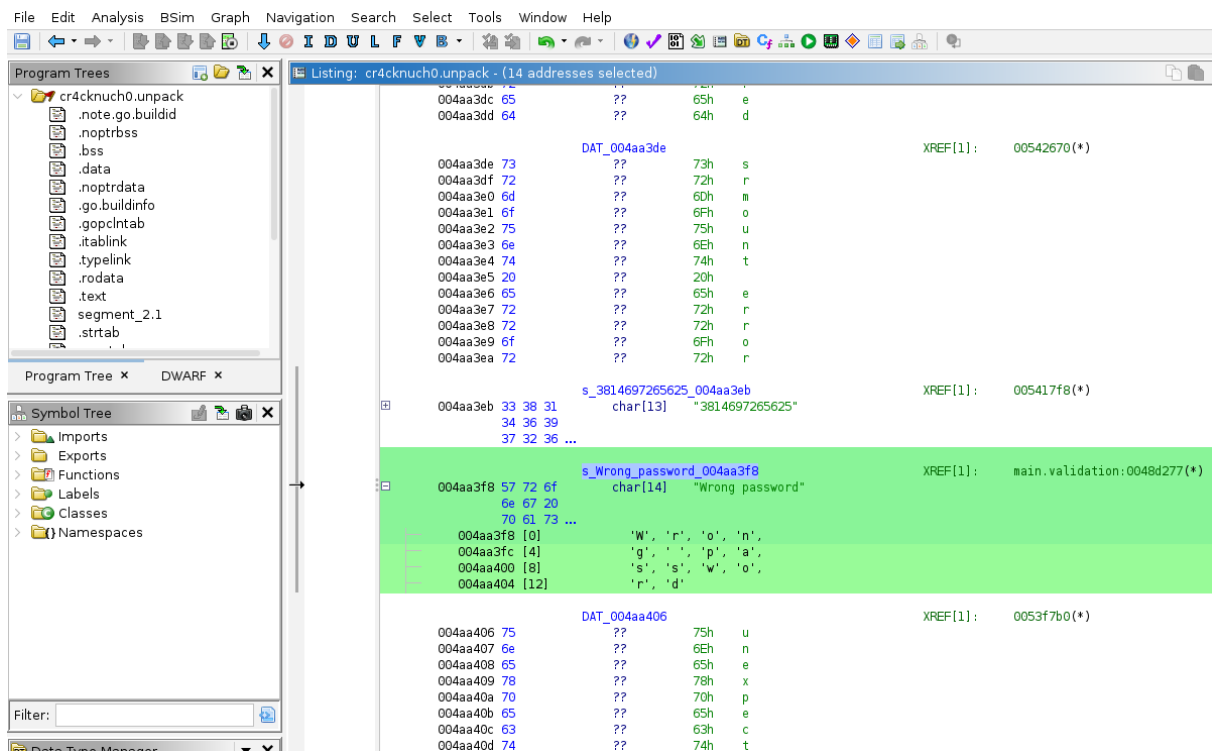


Figura 1.12: Ubicacion en memoria del string

8. Hacemos click derecho sobre la ubicación de memoria, seleccionamos referencias y luego show references to address:

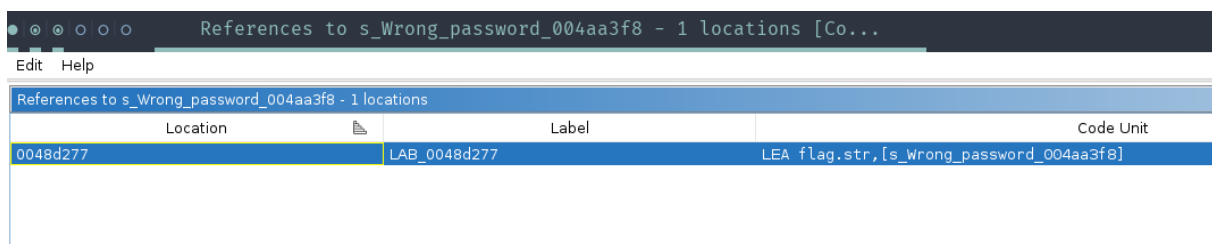


Figura 1.13: Dirección en memoria donde el mensaje de error es usado

9. Damos click en la dirección de memoria (0x0048d277) y cerramos la búsqueda.

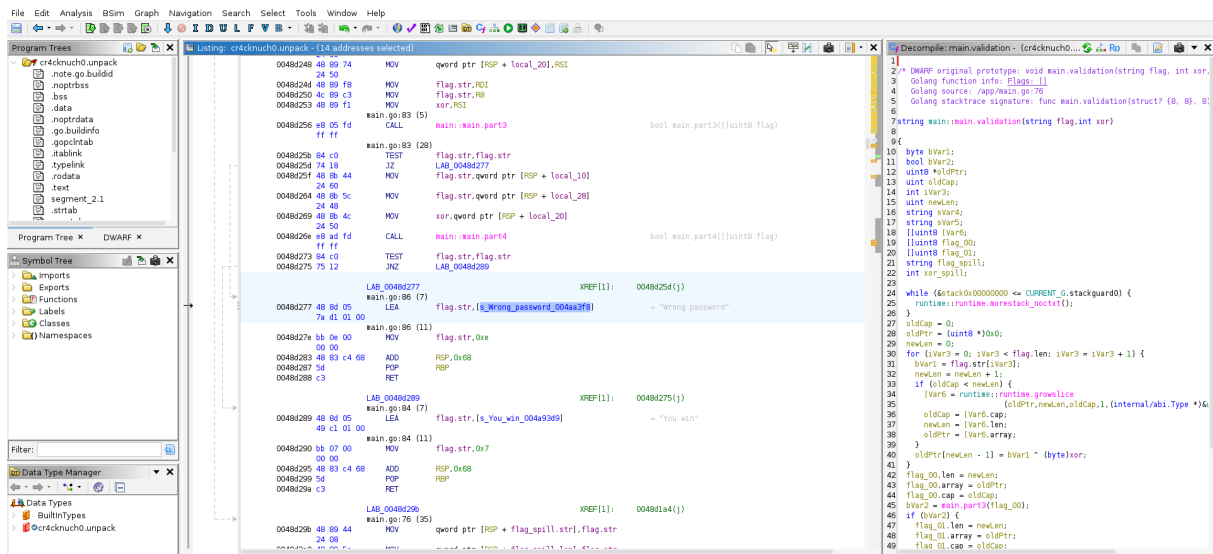


Figura 1.14: Direccion donde se carga el mensaje de error

Hemos llegado a una función donde el string es utilizado para desplegarlos aquel mensaje de nuestros primeros intentos.

Justo debajo podemos ver un string ‘You Win’ y si vemos rápidamente el pseudocódigo, podemos ver que aquí se encuentra un if-statement que válida si la contraseña es correcta:

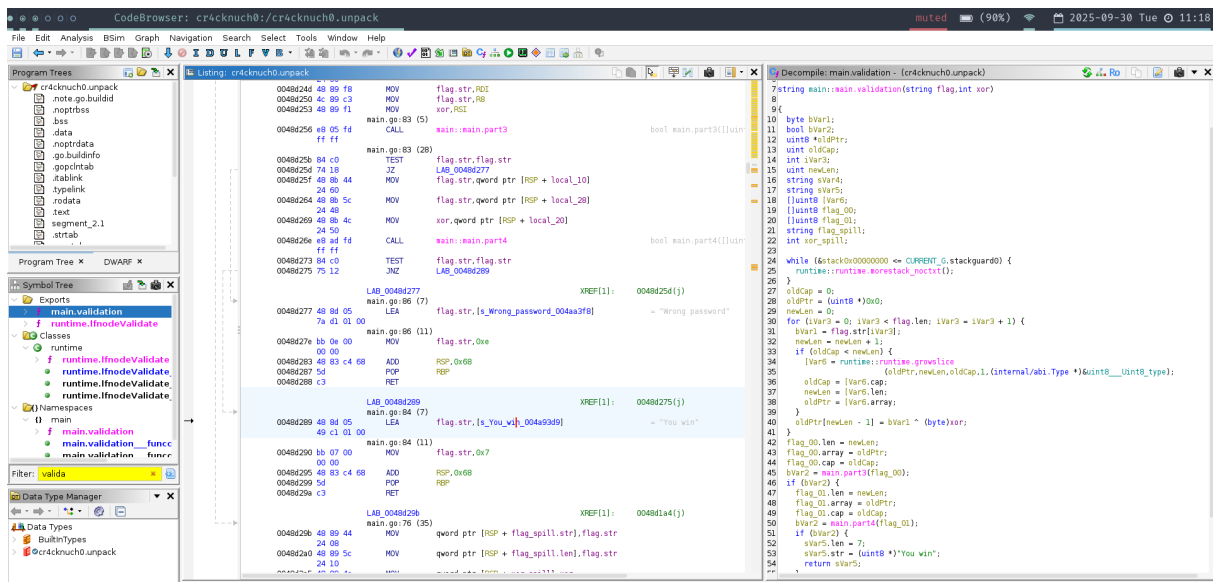


Figura 1.15: Condición de You Win

Como tenemos símbolos de debug, nos es posible observar nombres de funciones y algunas

propiedades del binario, haciendo nuestro análisis mas sencillo y mas detallado. Continuemos ahora con el análisis de la función `main.validation` (Como Go es un lenguaje que usa paquetes, todos los paquetes que son ejecutables deben llamarse `main`, por esta razón la función dentro del paquete `main` se llama `main.validation`. También el mismo paquete tiene su propia función `main`, que observaremos como `main.main`)

1.2.3 Función `main.validation`

El pseudocódigo completo que ahora debemos analizar es el siguiente:

```
1  /* DWARF original prototype: void main.validation(string flag, int xor, string ~r0)
2     Golang function info: Flags: []
3     Golang source: /app/main.go:76
4     Golang stacktrace signature: func main.validation(struct? {8, 8}, 8) ??? */
5
6  string main::main.validation(string flag,int xor)
7
8  {
9      byte bVar1;
10     bool bVar2;
11     uint8 *oldPtr;
12     uint oldCap;
13     int iVar3;
14     uint newLen;
15     string sVar4;
16     string sVar5;
17     []uint8 [Var6;
18     []uint8 flag_00;
19     []uint8 flag_01;
20     string flag_spill;
21     int xor_spill;
22
23     while (&stack0x00000000 <= CURRENT_G.stackguard0) {
24         runtime::runtime.morestack_noctxt();
25     }
26     oldCap = 0;
27     oldPtr = (uint8 *)0x0;
28     newLen = 0;
29     for (iVar3 = 0; iVar3 < flag.len; iVar3 = iVar3 + 1) {
30         bVar1 = flag.str[iVar3];
31         newLen = newLen + 1;
32         if (oldCap < newLen) {
33             [Var6 = runtime::runtime.growslice
34                 (oldPtr,newLen,oldCap,1,(internal/abi.Type *)&uint8___uint8_type);
35             oldCap = [Var6.cap;
36             newLen = [Var6.len;
37             oldPtr = [Var6.array;
38         }
39         oldPtr[newLen - 1] = bVar1 ^ (byte)xor;
40     }
```

```
41  flag_00.len = newLen;
42  flag_00.array = oldPtr;
43  flag_00.cap = oldCap;
44  bVar2 = main.part3(flag_00);
45  if (bVar2) {
46      flag_01.len = newLen;
47      flag_01.array = oldPtr;
48      flag_01.cap = oldCap;
49      bVar2 = main.part4(flag_01);
50      if (bVar2) {
51          sVar5.len = 7;
52          sVar5.str = (uint8 *)"You win";
53          return sVar5;
54      }
55  }
56  sVar4.len = 0xe;
57  sVar4.str = (uint8 *)"Wrong password";
58  return sVar4;
59 }
```

Como tenemos símbolos de debugging, algunos de los nombres de los parámetros que recibe la función `main.validation` se mantuvieron como `flag` y `xor`, así como sus tipos de datos, `string` y `int` respectivamente. La función devuelve un `string` al final de su ejecución.

Analizaremos bloque por bloque de código:

```
1  while (&stack0x00000000 <= CURRENT_G.stackguard0) {
2      runtime::runtime.morestack_noctxt();
3  }
```

La primera parte es una protección del stack implementada por el compilador para evitar que la ejecución sea afectada.

```
1  oldCap = 0;
2  oldPtr = (uint8 *)0x0;
3  newLen = 0;
4  for (iVar3 = 0; iVar3 < flag.len; iVar3 = iVar3 + 1) {
5      bVar1 = flag.str[iVar3];
6      newLen = newLen + 1;
7      if (oldCap < newLen) {
8          [Var6 = runtime::runtime.growslice(oldPtr,newLen,oldCap,1,(internal/abi.Type
          ↳ *)&uint8__Uint8_type);
9          oldCap = [Var6.cap;
10         newLen = [Var6.len;
11         oldPtr = [Var6.array;
12     }
13     oldPtr[newLen - 1] = bVar1 ^ (byte)xor;
14 }
```

Esta parte está bien interesante, primero se crea un arreglo vacío, después dentro de un ciclo for que rota cada carácter de `flag`, realiza una operación primero de agrandar el array y después agregar el resultado de un XOR entre el parámetro `xor` y el elemento de `flag` correspondiente al index del ciclo for.

Hay que descubrir cuál es el valor de `xor` porque es una operación sobre `flag`

```
1  flag_00.len = newLen;
2  flag_00.array = oldPtr;
3  flag_00.cap = oldCap;
4  bVar2 = main.part3(flag_00);
5  if (bVar2) {
6      flag_01.len = newLen;
7      flag_01.array = oldPtr;
8      flag_01.cap = oldCap;
9      bVar2 = main.part4(flag_01);
10     if (bVar2) {
11         sVar5.len = 7;
12         sVar5.str = (uint8 *) "You win";
13         return sVar5;
14     }
15 }
16 sVar4.len = 0xe;
17 sVar4.str = (uint8 *) "Wrong password";
18 return sVar4;
```

La siguiente sección tenemos dos if cascadeados, ambos dependen de la salida de las funciones `main.part3` y `main.part4`. Ambas funciones reciben una copia de `flag` (el nuevo valor de la `flag` después de la operación XOR).

El return final de la función `main.validation` es el mensaje de condición de éxito/fracaso; `You win` o `Wrong password`

Antes de analizar `main.part3` y `main.part4`, identifiquemos el valor del argumento `xor`, que como ya vimos, modifica el valor de `flag` antes de ser compartido a las demás funciones:

Desde el menu, seleccionamos `Graph` y después `Calls`:

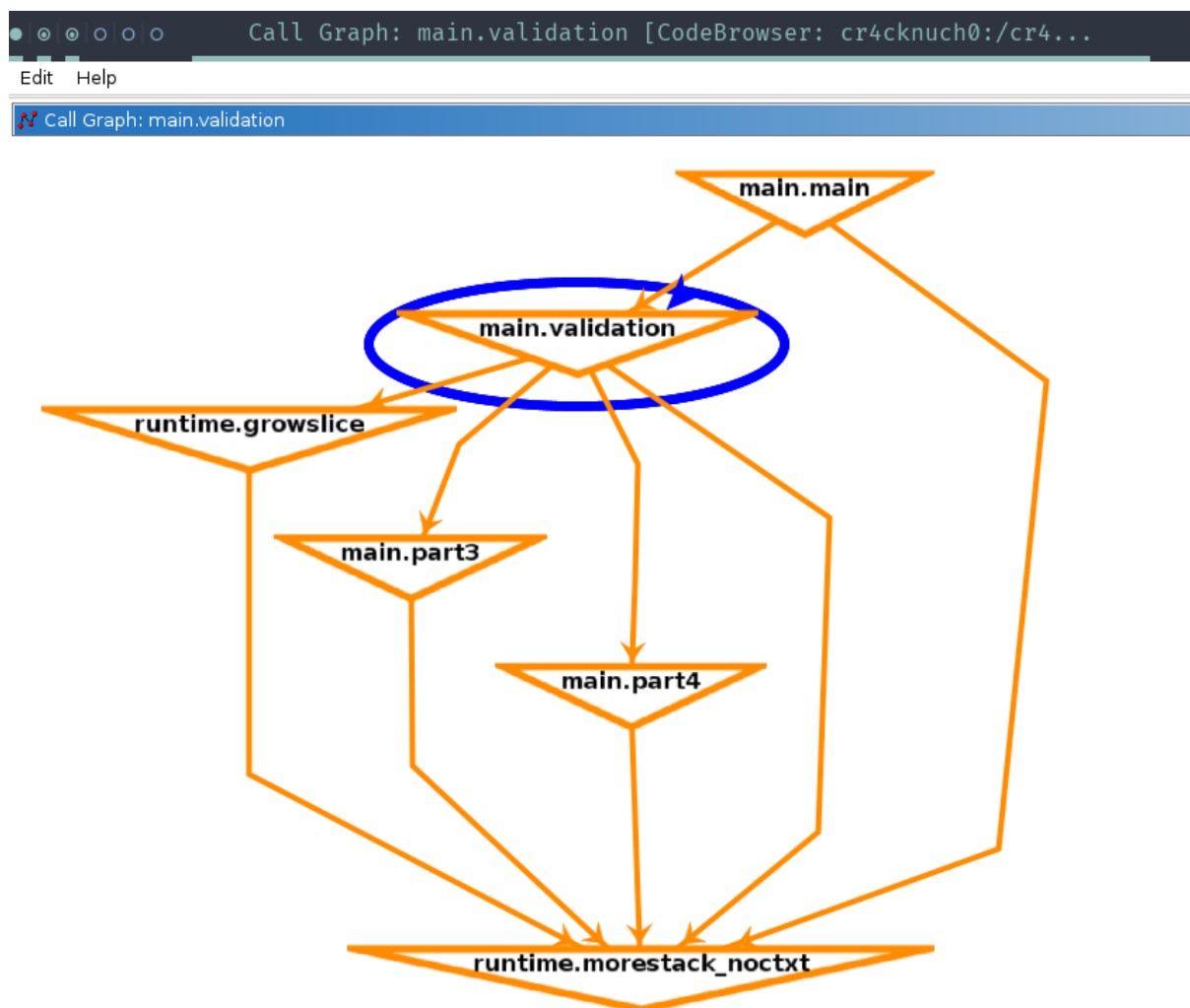


Figura 1.16: Grafo de llamadas entre funciones simplificado

Como podemos ver, la función `main.main` manda a llamar a `main.validation`. Busquemos en el pseudocódigo de la función `main.main`:

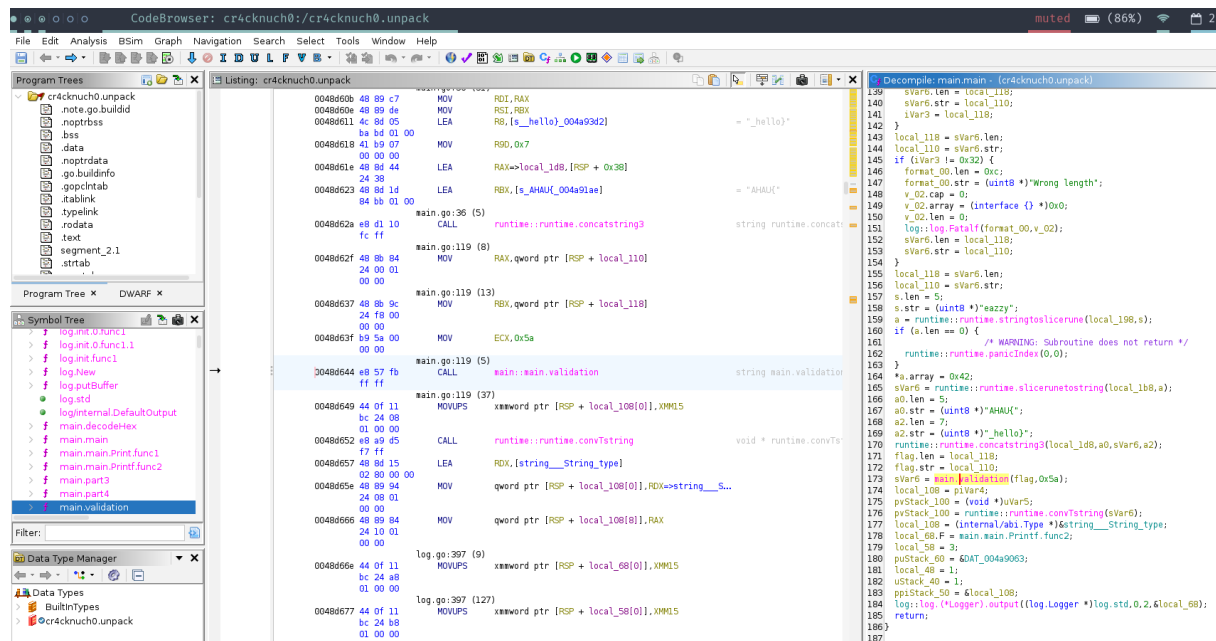


Figura 1.17: `main.main` llamando a `main.validation` con sus argumentos

Justo en la línea 173 podemos ver que el valor de xor es `0x5a` o lo que es en decimal 90.

Ahora ya podemos continuar. Lo que recibe `main.part3` y `main.part4` es el valor ingresado (`flag`) con una operación XOR de 90 en decimal.

1.2.4 Función `main.part3`

El pseudocódigo de `main.part3` es:

```
1  /* DWARF original prototype: void main.part3([[]uint8 flag, bool ~r0)
2   Golang function info: Flags: []
3   Golang source: /app/main.go:40
4   Golang stacktrace signature: func main.part3(struct? {8, 8, 8}) ??? */
5
6  bool main::main.part3([[]uint8 flag)
7
8  {
9      uint x;
10     []uint8 [Var1;
11     []uint8 flag_spill;
12
13     while (&stack0x00000000 <= CURRENT_G.stackguard0) {
14         runtime::runtime.morestack_noctxt();
15     }
16     if (0x18 < (uint)flag.cap) {
```

```

17 [Var1.len = 0x32;
18 [Var1.array = (uint8 *)"3625382146e14c8558131e815edf18531a6d701d46f1688d22";
19 [Var1.cap = 0x32;
20 [Var1 = main.decodeHex([Var1];
21 x = 0;
22 while( true ) {
23     if (0x18 < (int)x) {
24         return true;
25     }
26     if ((uint)[Var1.len <= x) break;
27     if ([Var1.array[x] != (uint8)((char)x + flag.array[x] * '\x02')) {
28         return false;
29     }
30     x = x + 1;
31 }
32 /* WARNING : Subroutine does not return */
33 runtime::runtime.panicIndex(x,[Var1.len];
34 }
35 /* WARNING : Subroutine does not return */
36 runtime::runtime.panicSliceAcap((int)flag.array,flag.len);
37 }

```

En las primeras líneas tenemos unas variables contextualizadas en la función y la protección del stack previamente mencionada.

Nos sigue un `if` que valida si `flag.cap` es mayor a `0x18`, 25 en decimal, que como sabemos es de 50 (longitud total de la flag). Posteriormente tenemos la declaración de un array de caracteres `[Var1` y se manda a llamar a la función `main.decodehex` con el argumento del slice que se declaró en esta función. Cabe señalar que el valor de `[Var1` se sobrescribió. Aquí es importante señalar que si la longitud de `flag.cap` fuera menor, el programa se terminaría en `panic`.

Ahora si continuamos con el siguiente bloque grande, tenemos un `return` en `true` y otro en `false`. Ambos están atrapados en un bucle `while` infinito, por lo que el `return` se debe dar en algún momento. `x` es un índice que se incrementa y que eventualmente nos llevara a `true` tras 25 validaciones (`0x18`). Esta es la condición que buscamos para poder conectarnos a `main.part4`.

Dentro del `while` tenemos los siguientes dos `if-statements`:

```

1     if ((uint)[Var1.len <= x) break;
2     if ([Var1.array[x] != (uint8)((char)x + flag.array[x] * '\x02')) {
3         return false;
4     }

```

El primer statement, compara `[Var1.len` (la longitud del array `[Var1`) con el contador, por lo que el `break` podría suceder cuando alcancemos la condición de que la longitud de `[Var1` es menor al contador. Esto causaría un `panic`.

El siguiente statement toma el elemento de [Var1 (el array después de salir de la función de `main.decodeHex`), se multiplica por 2 y se suma con el index `x`. Antes de verificar que hace `main.decodehex`, terminaremos de revisar el proceso actual de esta función

El proceso inverso para eliminar esta protección seria renombrar [Var1 como protegido, ya que representa una comparación en el `if`-statement directo con el valor ingresado (evaluación de la primera mitad de la flag):

```
1 flag[i] = ( protegido[i] - x ) / 2
```

Ahora, analicemos que hace la función `main.decodeHex`

1.2.5 Función `main.decodeHex`

El pseudocódigo de esta función es:

```
1  [uint8 main::main.decodeHex([uint8 src)
2
3  {
4      uint x;
5      uint len;
6      internal/abi.Type *y;
7      [uint8 [Var1;
8      multireturn{int;error} mVar2;
9      [interface {} v;
10     [uint8 [Var3;
11     [uint8 src_spill;
12     internal/abi.Type *local_18;
13     void *pvStack_10;
14     error_itab *extraout_RBX;
15
16     while (&stack0x00000000 <= CURRENT_G.stackguard0) {
17         runtime::runtime.morestack_noctxt();
18     }
19     len = (uint)src.len >> 1;
20     [Var1 = runtime::runtime.makeslice((internal/abi.Type *)&uint8__UInt8_type,len,len);
21     [Var1.len = len;
22     [Var1.cap = len;
23     mVar2 = encoding/hex::encoding/hex.Decode([Var1,([uint8)src);
24     y = mVar2.r1.tab;
25     x = mVar2.r0;
26     if (y != (internal/abi.Type *)0x0) {
27         if (y != (internal/abi.Type *)0x0) {
28             y = (internal/abi.Type *)y->PtrBytes;
29         }
30         v.len = 1;
31         v.array = (interface {} *)&local_18;
32         v.cap = 1;
```

```
33     local_18 = y;
34     pvStack_10 = mVar2.r1.data;
35     log::log.Fatal(v);
36     y = (internal/abi.Type *)extraout_RBX;
37 }
38 if (x <= len) {
39     [Var3.len = x;
40     [Var3.array = [Var1.array;
41     [Var3.cap = len;
42     return [Var3;
43 }
44 /* WARNING : Subroutine does not return */
45 runtime::runtime.panicSliceAcap(x,(int)y);
46 }
```

Esta función toma un array de `uint` y devuelve otro array de `uint`. Podemos ver que hace uso de la función nativa de `encoding.hex/Decode`, la cual almacena en un slice y retorna el valor obtenido desde `src`. En caso de error tendríamos un panic en la aplicación. Para mas detalles, aquí tenemos la descripción de esta función.

Ahora que sabemos que esta función únicamente se encarga de recibir un string en formato hex y devolverlo en formato byte array, podemos utilizar directamente este resultado como parte de la formula en `main.part3`.

1.2.6 De regreso a `main.part3`

Como conocemos la primera parte de la flag (AHAU{) podemos validar si podemos obtener este valor por el proceso inverso:

Como los primeros valores de hexstring son 36, 25, 38, 21, podemos calcular la desproteccion:

```
1 f = int((0x36-0)/2)^90
2 print(chr(f))
```

```
xbytemx@holi:~$ python3
Python 3.13.7 (main, Aug 20 2025, 22:17:40) [GCC 14.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print(chr(int((0x36-0)/2)^90))
A
>>> print(chr(int((0x25-0)/2)^90))
H
>>> print(chr(int((0x38-1)/2)^90))
A
>>> print(chr(int((0x21-2)/2)^90))
U
>>> □
```

Figura 1.18: Ya salio AHAU

Nota: Recordemos que el XOR de 90 es necesario porque fue el primer paso que recibieron todos los caracteres.

Podemos replicar este proceso hasta tener los 25 primeros caracteres de la flag. Recordemos que este paso es necesario debido al `if-statement` que nos requiere cumplir con `main.part3` para pasar a `main.part4`. Si estuviéramos haciendo análisis dinámico, necesitamos esta primera mitad, pero como estamos haciendo el análisis de manera estática sin parchar el binario, podemos analizar ahora `main.part4` antes de generar un solver.

1.2.7 Función `main.part4`

El pseudocódigo de `main.part4` es:

```
1  /* DWARF original prototype: void main.part4([]uint8 flag, bool ~r0)
2     Golang function info: Flags: []
3     Golang source: /app/main.go:52
4     Golang stacktrace signature: func main.part4(struct? {8, 8, 8}) ??? */
5
6  bool main::main.part4([]uint8 flag)
7
8  {
9     byte bVar1;
10    uint8 *puVar2;
11    uint x;
12    int iVar3;
13    uint y;
14    []uint8 [Var4;
15    []uint8 flag_spill;
```

```
16  uint y_00;
17
18  y = flag.len;
19  while (&stack0x00000000 <= CURRENT_G.stackguard0) {
20      runtime::runtime.morestack_noctxt();
21  }
22  if (y < 0x19) {
23      /* WARNING : Subroutine does not return */
24      runtime::runtime.panicSliceB(0x19,y);
25  }
26  [Var4.len = 0x32;
27  [Var4.array = (uint8 *)"4071510a7150584a4e0d5c0c577150095d63491d4e4b414f44";
28  [Var4.cap = 0x32;
29  [Var4 = main.decodeHex([Var4];
30  y_00 = [Var4.len;
31  puVar2 = [Var4.array;
32  x = 0;
33  do {
34      if ((int)(y - 0x19) <= (int)x) {
35          return true;
36      }
37      iVar3 = x + ((int)(SUB168(SEXT816(-0x5555555555555555) * SEXT816((int)x),8) + x) >> 1) *
38      -3;
39      bVar1 = flag.array[x + ((dword)(-(flag.cap + -0x19) >> 0x3f) & 0x19)];
40      if (iVar3 == 0) {
41          if (y_00 <= x) {
42              /* WARNING : Subroutine does not return */
43              runtime::runtime.panicIndex(x,y_00);
44          }
45          if ((bVar1 ^ puVar2[x]) != 99) {
46              return false;
47          }
48      }
49      else if (iVar3 == 1) {
50          if (y_00 <= x) {
51              /* WARNING : Subroutine does not return */
52              runtime::runtime.panicIndex(x,y_00);
53          }
54          if ((bVar1 ^ puVar2[x]) != 0x74) {
55              return false;
56          }
57      }
58      else if (iVar3 == 2) {
59          if (y_00 <= x) {
60              /* WARNING : Subroutine does not return */
61              runtime::runtime.panicIndex(x,y_00);
62          }
63          if ((bVar1 ^ puVar2[x]) != 0x66) {
64              return false;
65          }
66      }
67      x = x + 1;
68  } while( true );
69 }
```

Esta función es un poco más grande, pero si observamos, nuevamente tenemos un ciclo `do-while` eterno, donde hay cuatro `return`'s, tres en `false` y uno en `true`. El caso en `true` se da cuando el `index uVar3` llega a 25 (`0x19`), que como ya sabemos es la longitud restante de `main.part4`.

```
1  if ((int)(y - 0x19) <= (int)x)
```

Como podemos ver, `x` se incrementa en cada ciclo (`x = x+1`), por lo que mientras `y - 25` mayor a `x`, no terminaremos este ciclo.

Nuevamente tenemos un texto protegido `[Var4` que se convierte de hexstring a byte array. Este array es de tamaño 50 (`0x32`), y que representa el valor de `y_00`, por lo que hora resolviendo la incógnita de `y=flag.len=50; y-0x19=25`, sabemos que este `do-while` tendrá 25 ciclos sino alcanza algún `return false`.

La estructura de los 3 siguientes `if-statements` lucen muy similares:

```
1  if (iVar3 == AAA) {
2      if (y_00 <= x) {
3          /* WARNING : Subroutine does not return */
4          runtime::runtime.panicIndex(x,y_00);
5      }
6      if ((bVar1 ^ puVar2[x]) != BBB) {
7          return false;
8      }
9  }
```

Donde `AAA` puede ser 0,1,2 y donde `BBB` toma valores estáticos (99, 116, 102). Después de ver otras representaciones de estos valores, tenemos que:

```
1  >>> chr(99)
2  'c'
3  >>> chr(116)
4  't'
5  >>> chr(102)
6  'f'
```

Dentro de cada `if-statement` se compara `iVar3` con un entero, y dentro existe otro `if-statement` que válida si la operación XOR entre `bVar1` y `puVar2[x]` es igual a los valores `ctf`. La condición de panic protege los buffer overflows para no salirnos del scope.

Ahora solo debemos identificar que valores de `bVar1` y `iVar3`, como `puVar2[x]` es el array del texto protegido.

```

1  iVar3 = x + ((int)(SUB168(SEXT816(-0x5555555555555555) * SEXT816((int)x),8) + x) >> 1) *
2  -3;
   bVar1 = flag.array[x + ((dword)(-(flag.cap + -0x19) >> 0x3f) & 0x19)];

```

Para el valor de iVar3, la única dependencia que tenemos es con x, valor que conocemos (contador o index). Ahora las funciones SEXT816 y SUB168 son funciones internas, al leer la documentación encontramos la siguiente descripción:

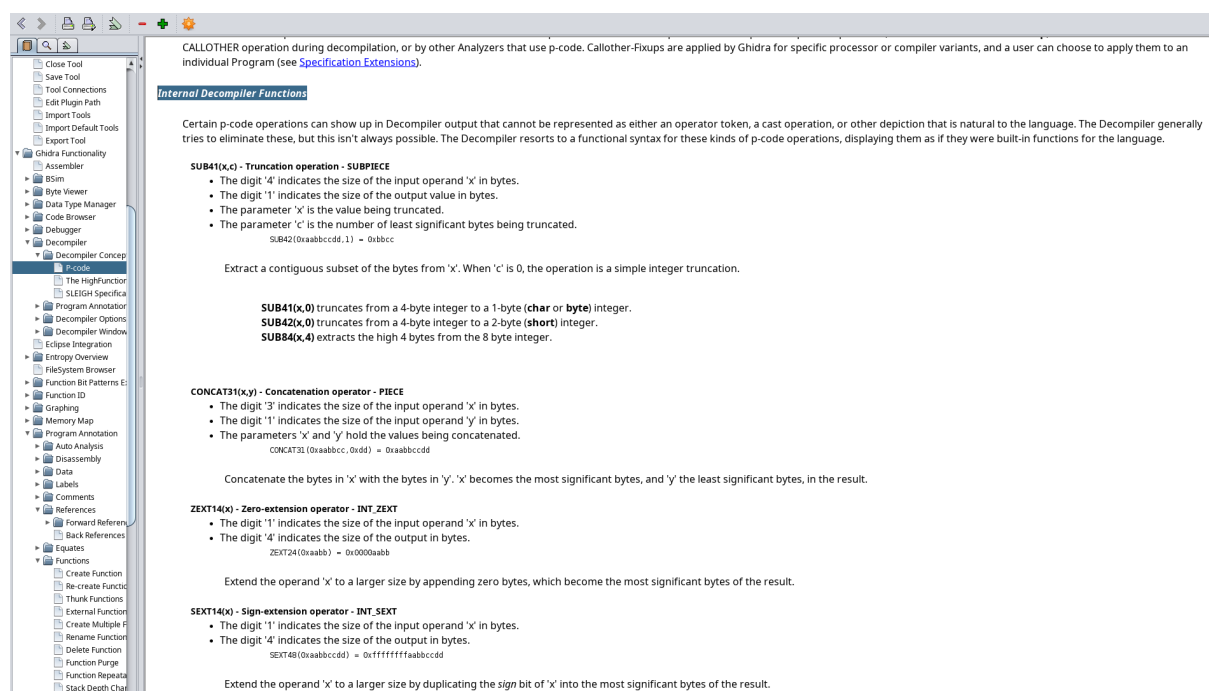


Figura 1.19: Documentación del plugin de decompilación

- SEXT816 (x) convierte el input x en un signed extended de 8 bytes a 16 bytes.
- SUB168 (x , 8) extrae del input x los 8 bytes más altos de un int de 16 bytes.

Si observamos el código en ensamblador:

```

1  0048d0a9 48 89 c7      MOV     RDI,flag.array
2  0048d0ac 48 b8 ab          MOV     flag.array,-0x5555555555555555
3          aa aa aa
4          aa aa aa aa
5  0048d0b6 49 89 d0      MOV     R8,RDX
6  0048d0b9 48 f7 e9      IMUL    flag.array
7  0048d0bc 48 01 ca      ADD     RDX,flag.array
8  0048d0bf 48 d1 fa      SAR     RDX,0x1
9  0048d0c2 48 8d 14 52    LEA     RDX,[RDX + RDX*0x2]
10 0048d0c6 48 89 c8      MOV     flag.array,flag.array

```



```
11      0048d0c9 48 29 d1      SUB      flag.array,RDX
12      0048d0c9 48 29 d1      SUB      iVar4,RDX
```

Sí validamos esto en un debugger, cuando el resultado es 0, cuando es 1 y cuando es 2, encontraremos que la operación realizada es módulo a 3 de x.

Para el caso de bVar1, flag.cap es constante y vale 50 (longitud de flag validada), por lo que podemos calcular el resultado ejecutando código o preguntándole a un ollamita (DeepSeek-R1):

```
1  xbytemx@holi:~$ podman exec -it ollama ollama run deepseek-r1:latest
2  >>> Hey dude could you please explain me (summary please) what is this c code line doing, x is
   ↳ a counter of a for loop? res = array[x +
3  ... ((-25 >> 63) & 25)]
4  Thinking...
5  We are given a C code line: `res = array[x + ((-25 >> 63) & 25)];`
6  We know that `x` is the counter in a for loop.
7
8  Let's break it down:
9
10 The expression inside the brackets has two parts:
11   Part 1: (-25 >> 63)
12   Part 2: (Part 1 & 25)
13
14 Step 1: Understand what `-25 >> 63` does.
15 In C, shifting a negative number is implementation-defined in some aspects, but let's assume
   ↳ we are on a system where integers
16 are represented with two's complement and the leftmost bit (the sign bit) is the 64th bit for
   ↳ `long long`? However, note that
17 `-25 >> 63` uses an integer of at least 32 bits because 63 is mentioned. But in C, shifting a
   ↳ negative number by more than or
18 equal to its width (if it's promoted to unsigned) might be undefined behavior unless the
   ↳ compiler defines it.
19
20 However, let me assume we are dealing with `int` for simplicity? The expression `-25 >> 63`
   ↳ suggests that the integer is at
21 least 64 bits wide because shifting by 63 and then masking with 25 (which requires only a few
   ↳ bits) might be done to get a
22 specific value.
23
24 But note: in C, if we shift an `int` by more than its width (for negative numbers), it's
   ↳ undefined behavior. However, let's look
25 at the expression:
26
27   (-25 >> 63)
28
29 Since -25 is negative and we are shifting right by 63 bits, this might be intended to work on
   ↳ a type that can hold such shifts.
30
31 Alternatively, note that `-25` in two's complement representation has all high bits set if we
   ↳ consider an unsigned integer? But
32 for signed integers, shifting right of a negative number (with the sign bit) is
   ↳ implementation-defined. However, let me explain
```

with two common approaches:

Approach 1: If we are on a system where `int` is at least 32 bits and the shift by 63 is defined as arithmetic shift (which sets the high bits to the sign bit if it's signed integer), then shifting `-25` right by 63 might set all those bits appropriately.

Approach 2: Alternatively, in many compilers, shifting a negative number with `>>` will treat it as an unsigned number? But that is not standard for arithmetic shifts. Actually, let me correct:

In C, the left operand of a shift operator must be an integer type and the right operand must
 → be non-negative and less than or
 equal to the width of the promoted left operand (which usually means converting `int` to
 → `unsigned int`). So if we have -25 as an
`int`, then shifting by 63 is defined for that `int` only if it's at least 64 bits? Actually,
 → no: in C, integer promotions are
 applied. If the type of `-25` is `int`, and assuming `int` is 32 bits, then we cannot shift a
 → negative number (which has been
 promoted to `unsigned int`) by more than its width minus one because shifting an unsigned
 → integer by n where $n \geq \text{width}$
 results in undefined behavior? Actually, no: for unsigned integers, shifting by k when
 → $k \geq \text{width}$ is defined as shifting modulo the
 width. So:

Let's say we have a 32-bit signed `int` (which after promotion becomes 32-bit unsigned).
 → Then -25 would be represented as an unsigned integer with enough bits to represent it? But note: in two's complement, negative numbers are stored by flipping all bits and adding one. However, the expression `-25 >> 63` is applied on a signed `int`.

Since the behavior of shifting a negative number (with more than 0) for `>>` is implementation-defined, it's risky to rely on.

But let's look at what the code might be trying to do:

The idea seems to be: we want to get an index that wraps around or cycles in some way.
 → However, note the mask with 25 (which is a small number).

Let me rewrite without assuming signedness and consider two common ways:

```
Option A: If we assume that 'int' is at least 64 bits? Then -25 has more than 32 bits set if
→ it's negative. But shifting right
by 63 in a 64-bit integer would move the sign bit (which for negative numbers in two's
→ complement is the highest bit) to the
lowest positions, and then we mask with 25.
```

However, note that -25 in two's complement on an n-bit system has all bits set except the first few. But let me consider a specific example:

Suppose `int` is 64 bits (common these days). Then -25 can be represented as:

```
...11111111101111111111111111111111111111
```

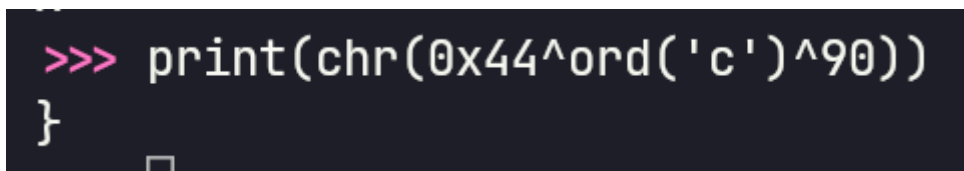
El valor de `bVar1` es entonces `flag.array[25 + x]`, lo que significa que sera los siguientes 25 caracteres de `flag`.

Asociando el concepto anterior, el resultado del módulo 3 de `x`, tendrá una operación XOR contra la letra correspondiente de `ctf`. Aquí es donde empezamos con cada letra de `flag` y cada caso de los `if-statements` descritos.

Si el resultado es 0, el XOR es contra 'c' Si el resultado es 1, el XOR es contra 't' Si el resultado es 2, el XOR es contra 'f'

Como conocemos el último valor `}`, cuyo valor de `x` es 24 (inicia en 0), por lo que su modulo es 0, y su letra es 'c', si tomamos el valor de la sección protegida y le aplicamos el algoritmo de protección para verificar si encontramos el carácter:

```
1 print(chr(0x44^ord('c')^90))
```



```
>>> print(chr(0x44^ord('c')^90))
}
```

Figura 1.20: Validando el final de la segunda parte

Nota: No hay que olvidar la primera protección, el XOR de 90.

Ahora solo toca crear un gran script que decodifique la flag.

1.2.8 Construcción del solver

Como pudimos ver en la función `main.part3`, necesitamos tomar el valor protegido, restarle el index actual (de 0 a 25), dividirlo entre 2 y finalmente hacer un XOR con 90.

El código se implementó de la siguiente manera:

```
1 flag = b""
2 xor = 90
3 hexblock1 = binascii.unhexlify("3625382146e14c8558131e815edf18531a6d701d46f1688d22")
4
5 for idx, c in enumerate(hexblock1):
6     flag += bytes(chr(int((int(c) - idx) / 2) ^ xor), "utf-8")
7
8 print(str(flag))
```

Para la segunda parte (`main.part4`), es similar a la primera, se toma el valor protegido, se evalúa de acuerdo al index, cuál es la letra correspondiente y se ejecuta un XOR con la letra y el valor 90 para desproteger la flag.

```
1 flag = b""
2 xor = 90
3 hexblock2 = binascii.unhexlify("4071510a7150584a4e0d5c0c577150095d63491d4e4b414f44")
4
5 for idx, c in enumerate(hexblock2):
6     match idx % 3:
7         case 0:
8             flag += bytes(chr(int(c) ^ ord("c") ^ xor), "utf-8")
9         case 1:
10            flag += bytes(chr(int(c) ^ ord("t") ^ xor), "utf-8")
11        case 2:
12            flag += bytes(chr(int(c) ^ ord("f") ^ xor), "utf-8")
13
14 print(str(flag))
```

1.2.9 Final script

El script final queda de la siguiente manera:

```
1 #!/usr/bin/env python
2 # coding: utf-8
3 import binascii
4
5 def main():
6     flag = b""
7
8     xor = 90
9     hexblock1 = binascii.unhexlify("3625382146e14c8558131e815edf18531a6d701d46f1688d22")
10    hexblock2 = binascii.unhexlify("4071510a7150584a4e0d5c0c577150095d63491d4e4b414f44")
11
12    for idx, c in enumerate(hexblock1):
13        flag += bytes(chr(int((int(c) - idx) / 2) ^ xor), "utf-8")
14
15    for idx, c in enumerate(hexblock2):
16        match idx % 3:
17            case 0:
18                flag += bytes(chr(int(c) ^ ord("c") ^ xor), "utf-8")
19            case 1:
20                flag += bytes(chr(int(c) ^ ord("t") ^ xor), "utf-8")
21            case 2:
22                flag += bytes(chr(int(c) ^ ord("f") ^ xor), "utf-8")
23    return str(flag)
24
25 if __name__ == "__main__":
26    print(main())
```

Después de ejecutarlo:

```
xbytemx@holi:~$ cat solver.py
#!/usr/bin/env python
# coding: utf-8
import binascii

def main():
    flag = b""

    xor = 90
    hexblock1 = binascii.unhexlify("3625382146e14c8558131e815edf18531a6d701d46f1688d22")
    hexblock2 = binascii.unhexlify("4071510a7150584a4e0d5c0c577150095d63491d4e4b414f44")

    for idx, c in enumerate(hexblock1):
        flag += bytes(chr(int((int(c) - idx) / 2) ^ xor), "utf-8")

    for idx, c in enumerate(hexblock2):
        match idx % 3:
            case 0:
                flag += bytes(chr(int(c) ^ ord("c") ^ xor), "utf-8")
            case 1:
                flag += bytes(chr(int(c) ^ ord("t") ^ xor), "utf-8")
            case 2:
                flag += bytes(chr(int(c) ^ ord("f") ^ xor), "utf-8")
    return str(flag)

if __name__ == "__main__":
    print(main())
xbytemx@holi:~$ python3 solver.py
b' AHAU{4yer_Pas3_x_tu_C4sa_y_m3_ladr4r0n_l0s_p3rros}'
xbytemx@holi:~$
```

Figura 1.21: flag

1.3 Flag

```
1 AHAU{4yer_Pas3_x_tu_C4sa_y_m3_ladr4r0n_l0s_p3rros}
```