Seguridad del software

PEC 1

Debugging y reversing de una aplicación

UOC - MISTIC

Pablo Riutort Grande

18 de octubre de 2020

Índice

1. Introducción **3**

2. Reversing **4**

3. Análisis de la función principal **6**
3.1. Vista gráfica 7

4. Alterar el flujo del código **10**
4.1. Patching 10

Índice de figuras

1. Comandos de r2 3
2. Análisis del binario 4
3. Comando 'a' 4
4. Flags del binario 5
5. Listado de strings del binario 5
6. Comando para ver el contenido de la pila en Radare 5
7. entry0 tiene la misma dirección de memoria que main 6
8. Print Disassembled Function 7
9. PDF: Sección de la primera función puts 7
10. PDF: Sección de la función fgets 7
11. PDF: Sección de la función strcmp 7
12. PDF: Sección de la segunda función puts 7
13. Primer bloque del modo gráfico 8
14. Segundo bloque del modo gráfico 8
15. Variables de las funciones 9
16. Hexadecimal del binario guess_my_name 10
17. Cambio del código hexadecimal del binario 11

1. Introducción

Para este ejercicio se ha instalado y utilizado la herramienta de análisis de binarios e ingeniería inversa radare2 [1]. El binario analizado ha sido proporcionado por crackmes.one que es un servicio que proporciona binarios para analizar [2], se ha seleccionado un binario de nivel 1 compilado en C para sistemas operativos Unix/Linux [3].
En la sección de reversing se analiza el binario mencionado ... [completar una vez terminado el ejercicio].

El reverse engineering es un proceso complicado así que Radare es un software basado en interfaz de línea de comandos donde se puede especificar en cada momento qué se desea hacer sobre el binario. Para interactuar con él hay que introducir distintos comandos con distintos parámetros [Fig. 1].

```
[0x00400450]> ?
Usage: [.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
| %var=value              alias for 'env' command
| *[?] off[=[0x]value]    pointer read/write data/values (see ?v, wx, wv)
| (macro arg0 arg1)       manage scripting macros
| .[?] [-|(m)|f|!sh|cmd]  Define macro or load r2, cparse or rlang file
| _[?]                    Print last output
| =[?] [cmd]              send/listen for remote commands (rap://, raps://
, udp://, http://, <fd>)
| <[...]                  push escaped string into the RCons.readChar buff
er
| /[?]                    search for bytes, regexps, patterns, ..
| ![?] [cmd]              run given command as in system(3)
| #[?] !lang [..]         Hashbang to run an rlang script
| a[?]                    analysis commands
| b[?]                    display or change the block size
| c[?] [arg]              compare block with given data
| C[?]                    code metadata (comments, format, hints, ..)
| d[?]                    debugger commands
| e[?] [a[=b]]            list/get/set config evaluable vars
| f[?] [name][sz][at]     add flag at current address
| g[?] [arg]              generate shellcodes with r_egg
| i[?] [file]             get info about opened file from r_bin
| k[?] [sdb-query]        run sdb-query. see k? for help, 'k *', 'k **' ..
.
| l [filepattern]         list files and directories
| L[?] [-] [plugin]       list, unload load r2 plugins
| m[?]                    mountpoints commands
| o[?] [file] ([offset])  open file at optional address
| p[?] [len]              print current block with format and length
| P[?]                    project management utilities
| q[?] [ret]              quit program with a return value
| r[?] [len]              resize file
| s[?] [addr]             seek to address (also for '0x', '0x1' == 's 0x1'
)
| t[?]                    types, noreturn, signatures, C parser and more
| T[?] [-] [num|msg]      Text log utility (used to chat, sync, log, ...)
| u[?]                    uname/undo seek/write
| v                       visual mode (v! = panels, vv = fcnview, vV = fcn
graph, vVV = callgraph)
| w[?] [str]              multiple write operations
| x[?] [len]              alias for 'px' (print hexadecimal)
| y[?] [len] [[[@]addr    Yank/paste bytes from/to memory
| z[?]                    zignatures management
| ?[??][expr]             Help or evaluate math expression
| ?$?                     show available '$' variables and aliases
| ?@?                     misc help for '@' (seek), '~' (grep) (see ~??)
| ?>?                     output redirection
| ?|?                     help for '|' (pipe)
```

Figura 1: Comandos de r2

2. Reversing

El reversing se ha realizado con radare2 sobre el binario llamado “guess_my_name”, el primer paso es importar el binario con radare y hacer un análisis del mismo.

```
pablo@fossa:~/UOC/software/PEC_1$ r2 guess_my_name
 -- ---8<--------------------8<-----------------8<-----------------8<------
[0x100000ef5]> a
fcns    0
xrefs   0
calls   0
strings 3
symbols 7
imports 3
covrage 64
codesz  4096
percent 1%
[0x100000ef5]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x100000ef5]>
```

Figura 2: Análisis del binario

El comando ’a’ realiza un análisis del binario. Existen varios niveles de análisis con radare, se ha utilizado el más exhaustivo [Fig. 3]. También existen distintas variantes de ’a’, más adelante se utilizará para listar funciones y variables [1].

```
[0x100000ef5]> a?
Usage: a  [abdefFghoprxstc] [...]
| a                alias for aai - analysis information
| a*               same as afl*;ah*;ax*
| aa[?]            analyze all (fcns + bbs) (aa0 to avoid sub renaming)
```

Figura 3: Comando ’a’

Después de este análisis ya tendremos suficiente información como para consultar los flags del binario. Los flags nos aportan información muy interesante ya que asocian un nombre con un offset del archivo y además se agrupan en namespaces, algunos de estos namespaces (flag spaces) son secciones, funciones, símbolos y strings [Fig. 4] [4].

Para listar strings podemos seleccionar el flag space de strings 5. Vemos que se listan 3 strings: “What’s my name”, “mrmtg” y “You got it”.

Si listamos las funciones veremos que radare encuentra 4, 3 de sistema y 1 propia llamada “entry0”. Las funciones de sistema son “puts”, “gets” y “strcmp”, estas funciones son vulnerables al buffer overflow que ocurre cuando los datos se escriben sobrepasando los límites de memoria reservada para una estructura de datos en particlar. En C se definen los strings como un array de caracteres terminados por el carácter “null” y no se realiza una comprobación

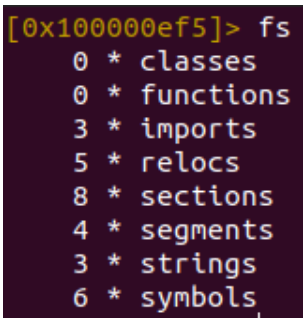

Figura 4: Flags del binario

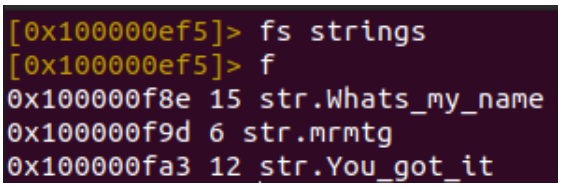

Figura 5: Listado de strings del binario

de límites de estos arrays incluso en las librerías estándard del lenguaje; si el tamaño del string es demasiado grande se produce el desbordamiento y puede provocar un comportamiento no deseado del programa [5].

Para ver el contenido de la pila en Radare podemos utilzar el comando “px” que se utiliza para mostrar el contenido en hexadecimal de la memoria, en este caso, para apuntar al contenido de la pila podemos usar “rsp”: “pxa @ rsp” [7] [Fig. 6].

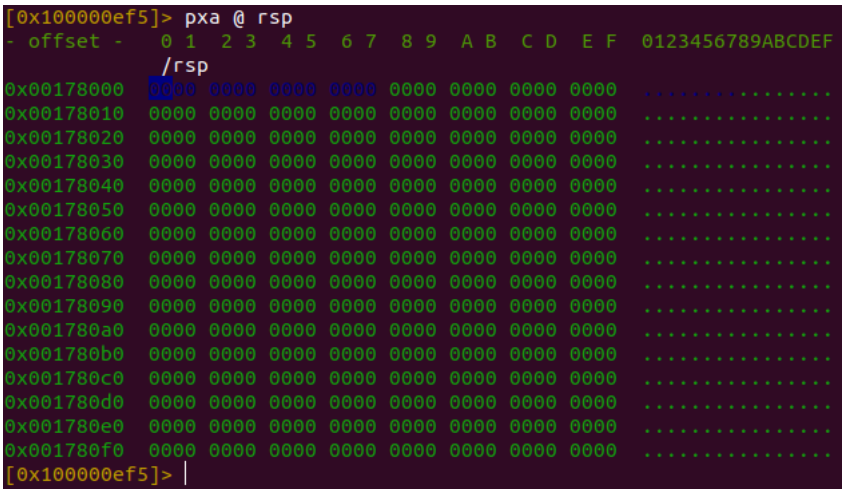

Figura 6: Comando para ver el contenido de la pila en Radare

3. Análisis de la función principal

La función “entry0” hace la función de main tal y como podemos ver en la dirección de memoria 0x100000ef5 que coincide con el nombre de “main”.

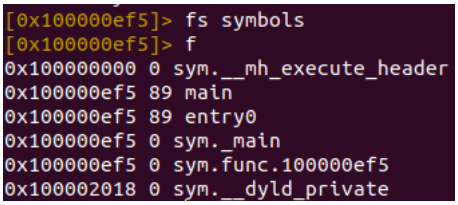

Figura 7: entry0 tiene la misma dirección de memoria que main

Para ver el cuerpo de la función podemos utilizar el comando seek sobre el nombre de la función para buscar el la función “entry0”: ’s entry0’. Posteriormente podemos ejecutar el comando “pdf” para ver la función seleccionada desensamblada. Tanto en el cuerpo de la función como con el comando “afv” podemos ver la declaración de variables, en nuestro binario encontramos “char *s1” [4] [Fig. 8].

Si analizamos detenidamente la función desensamblada podemos ver que primero se muestra por pantalla el string “What’s my name” [Fig. 9], seguido de la función fgets que lee una línea de un stream especificado y lo guarda en una variable [Fig. 10]. Esta variable es luego comparada con otro string asignado a “mrmtg” mediante la función strcmp [Fig. 11]. Si el resultado de esta comparación no es positivo, entonces saltamos a la dirección de memoria 0x100800f09, en caso contrario se muestra el string “You got it!” [Fig. 12].

De este comportamiento, podemos deducir que el programa hace una comparación con el string “mrmtg”, en caso de no coincidir saltamos a otra dirección de memoria en bucle.

A continuación tenemos una aproximación de lo que podría ser el código fuente del programa, un bucle continuo mediante “do while” que recibe el input estándard y lo compara con el string ”mrmtg”, en caso de acierto muestra el string “You got it!” y finaliza el programa.

```c
#include <stdio.h>
#include <string.h>

char s[6];

int main() {
  puts("Whats my name?");

  do
    fgets(s, sizeof(s), stdin);
  while(strcmp(s, "mrmtg"));

  puts("You got it!");

  return 0;
}
```

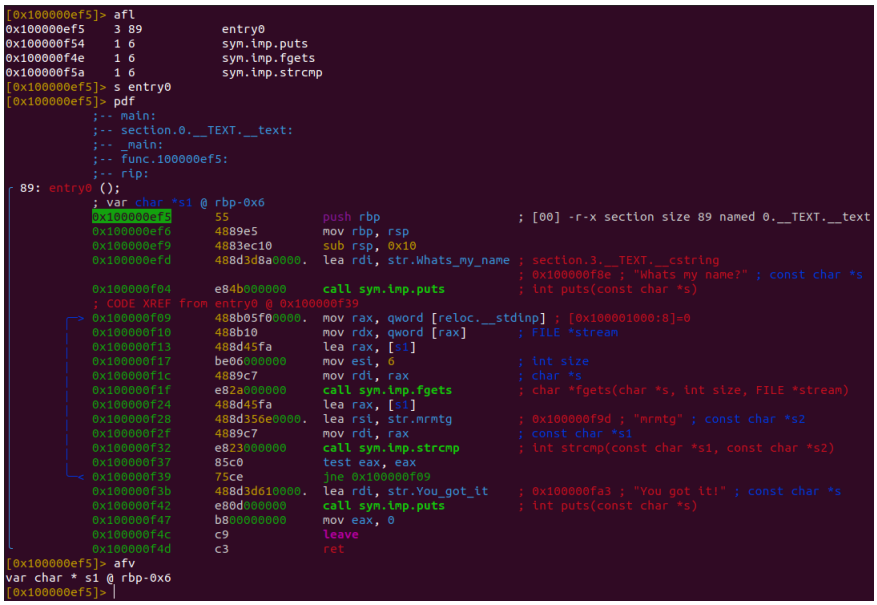

Figura 8: Print Disassembled Function

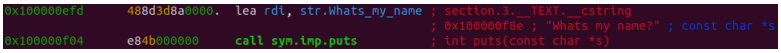

Figura 9: PDF: Sección de la primera función puts

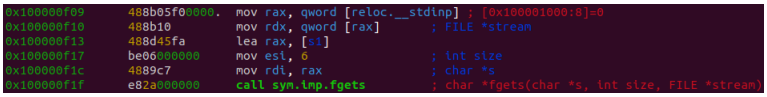

Figura 10: PDF: Sección de la función fgets

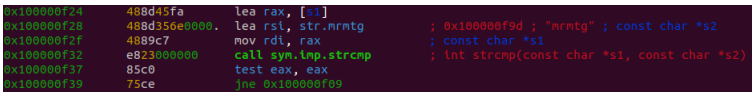

Figura 11: PDF: Sección de la función strcmp

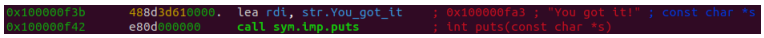

Figura 12: PDF: Sección de la segunda función puts

3.1. Vista gráfica

En radare existe el modo gráfico que nos permite ver el flujo del binario, podemos acceder a este modo mediante el comando “VV” [Fig. 14].

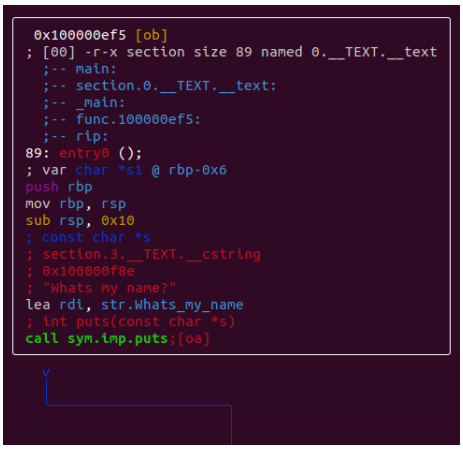

Figura 13: Primer bloque del modo gráfico

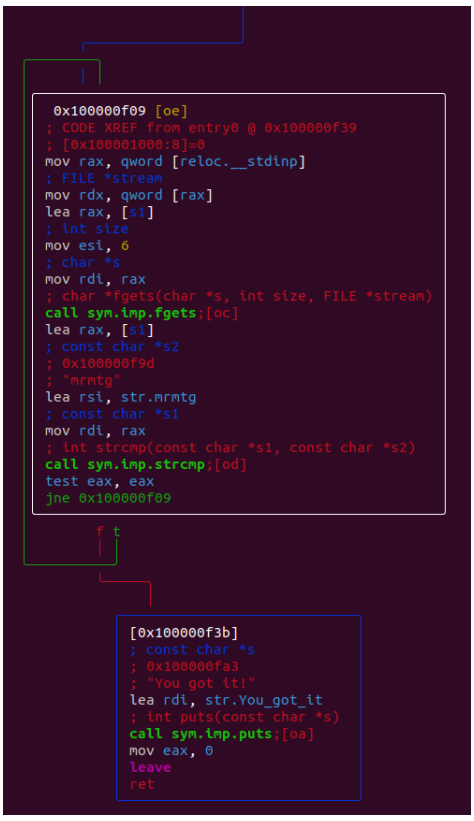

Figura 14: Segundo bloque del modo gráfico

En este modo también se puede ver con más detalle las variables declaradas y otras constantes que se utilizan en las funciones llamadas en el binario [Fig. 15].

```
const char *s;
int puts(const char *s);
```

```
FILE *stream;
int size = 6;
char *s;
char *fgets(char *s, int size, FILE *stream);
```

```
const char *s2;
const char *s1;
int strcmp(const char *s1, const char *s2);
```

```
const char *s;
int puts(const char *s)
```

En la asignación de strings a variables se ejecuta el mnemónico "lea rdi, str.<String>" donde "lea" es el opcode de *Load Effective Address* que carga la dirección efectiva del string en el registro *rdi*, usado como puntero destino.
En la declaración de arrays de tipo char está el mnemónico "mov rdi, rax". El registro *rdi* contiene la dirección de memoria del array y la instrucción copia el contenido del registro *rax* (Registro acumulador o de propósito general) al primer elemento del array.
La declaración de la variable de tipo entero "size" consiste en mover el valor 6 al registro *esi*, que es un registro utilizado como puntero.

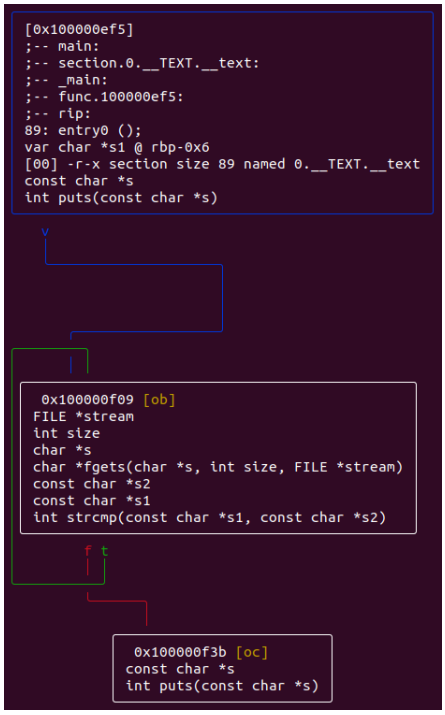

Figura 15: Variables de las funciones

4. Alterar el flujo del código

Como ya se ha comentado anteriormente, las funciones “puts”, “gets” y “strcmp” son vulnerables al buffer overflow y estas pueden ser atacadas si la entrada se prepara de forma especial para corromper la ejecución del stack del programa. En el siguiente ejemplo el código utiliza la función vulnerable gets() para leer datos en un buffer. Como no hay limitación de tamaño para la lectura de estos datos en la función depende del usuario el introducir los datos del tamaño adecuado al buffer [6].

```
...
char buf[BUFSIZE];
gets(buf);
...
```

Si accede a la pila de ejecución con la técnica de buffer overwlor se podrían introducir instrucciones de alto nivel como sacar una shell.

4.1. Patching

Otra alternativa para cambiar el comportamiento del binario es cambiando el binario en sí. Para eso se puede ver el código decompilado y decidir qué bytes se tienen que cambiar para modificar el comportamiento como, por ejemplo, cambiando alguna condición de salto [8].
En Linux tenemos el comando “xxd” que crea un dump en hexadecimal de un archivo o del input estándar, podemos ver el contenido de un binario con este comando [Fig. 17].

```
pablo@fossa:~/UOC/software/PEC_1$ xxd guess_my_name > guess_my_name.hex; head guess_my_name.hex
00000000: cffa edfe 0700 0001 0300 0000 0200 0000  ................
00000010: 1100 0000 9805 0000 8500 2000 0000 0000  .......... .....
00000020: 1900 0000 4800 0000 5f5f 5041 4745 5a45  ....H...__PAGEZE
00000030: 524f 0000 0000 0000 0000 0000 0000 0000  RO..............
00000040: 0000 0000 0100 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 1900 0000 d801 0000  ................
00000070: 5f5f 5445 5854 0000 0000 0000 0000 0000  __TEXT..........
00000080: 0000 0000 0100 0000 0010 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0010 0000 0000 0000  ................
```

Figura 16: Hexadecimal del binario guess_my_name

En nuestro binario tenemos una única condición de salto (jne) y es la candidata idónea para ser modificada de tal forma que cambie el flujo del programa. Esta condición de salto se encuentra en la dirección 0x000011cd la cual podemos encontrar en el hexadecimal para buscar mejor nuestra instrucción de salto.

El opcode del mnemónico según la documentación del procesador empieza por 75c, si buscamos ese número en el dump en hexadecimal lo podremos modificar por 74c0 que es el opcode del mnemónico JE (Jump short if equal) [9]. Una vez modificado podemos restaurar un binario nuevo a partir del nuevo hexadecimal y ejecutarlo para ver que el comportamiento se ha modificado [Fig. 17].

```
pablo@fossa:~/UOC/software/PEC_1$ xxd guess_my_name > guess_my_name.hex
pablo@fossa:~/UOC/software/PEC_1$ grep "000011c" guess_my_name.hex
000011c0: 8d3d 532e 0000 e8c5 feff ff85 c075 ce48  .=S.........u.H
pablo@fossa:~/UOC/software/PEC_1$ vim guess_my_name.hex
pablo@fossa:~/UOC/software/PEC_1$ grep "000011c" guess_my_name.hex
000011c0: 8d3d 532e 0000 e8c5 feff ff85 c074 ce48  .=S.........u.H
pablo@fossa:~/UOC/software/PEC_1$ xxd -r guess_my_name.hex > guess_my_name.cracked
pablo@fossa:~/UOC/software/PEC_1$ chmod u+x guess_my_name.cracked
pablo@fossa:~/UOC/software/PEC_1$ ./guess_my_name.cracked
Whats my name?
pablo
You got it!
```

Figura 17: Cambio del código hexadecimal del binario

Referencias

[1] **radare2**
Radare2 is a complete framework for reverse-engineering and analyzing binaries
`https://rada.re/n/radare2.html`

[2] **crackmes.one**
This is a simple place where you can download crackmes to improve your reverse engineering skills.
`https://crackmes.one/`

[3] **crackmes.one**
mrmtg's Guess_my_name
`https://crackmes.one/crackme/5ed0584b33c5d449d91ae67b`

[4] **Radare2 Book**
Flags `https://radare.gitbooks.io/radare2book/content/basic_commands/flags.html`
Code Analysis `https://radare.gitbooks.io/radare2book/content/analysis/code_analysis.html`
Hexadecimal View `https://radare.gitbooks.io/radare2book/content/basic_commands/print_modes.html`
Variables `https://radare.gitbooks.io/radare2book/content/analysis/variables.html`
Vidual Mode `https://radare.gitbooks.io/radare2book/content/visual_mode/intro.html`

[5] **Secure Coding in C and C++: Strings and Buffer Overflows**
Robert C. Seacord, Apr 24, 2013
`https://www.informit.com/articles/article.aspx?p=2036582&seqNum=3`

[6] **OWASP**
Buffer Overflow
`https://owasp.org/www-community/vulnerabilities/Buffer_Overflow`

[7] **Stack Exchange - Reverse Engineering**
How to get a nice stack view in radare2?
https://reverseengineering.stackexchange.com/questions/16844/how-to-get-a-nice-stack-view-in-radare2

[8] **Using Radare2 to patch a binary**
Derik Ramirez, Dec 28 2019.
https://rderik.com/blog/using-radare2-to-patch-a-binary/

[9] **Intel Pentium Instruction Set Reference (Basic Architecture Overview)**
JNE - Jump if Condition Is Met
http://faydoc.tripod.com/cpu/jne.htm