

Programmazione Sicura



Reverse
Engineering



Barbara Masucci
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Punto della situazione

- Nelle lezioni scorse abbiamo visto diverse tecniche per
 - Iniezione locale
 - Iniezione remota
 - Corruzione della memoria
- Scopo della lezione di oggi:
 - Scoprire come **estrarre informazioni** da file binari
 - Introdurre i principi di base della **Reverse Engineering**
 - Descrivere dei **tool** che possono essere utilizzati per la scoperta di vulnerabilità all'interno di programmi
 - Risolvere due **sfide Capture The Flag** presentate nell'ambito del programma **Cyberchallenge.IT**



Ottenere informazioni da un file binario

- Dato un **file binario**, possiamo
 - Controllare se è **eseguibile** oppure no
 - Scoprire per quale **architettura** è stato compilato
 - Collezionare **simboli e stringhe** usate dal programma
 - Controllare se c'è un **processo in esecuzione** associato al programma
 - Identificare i **nomi delle funzioni e delle librerie** usate dal programma



Ottenere informazioni da un file binario

- Esistono diversi comandi per estrarre informazioni da un **file binario**:
 - **file**
 - **strings**
 - **objdump**
 - **readelf**
- L'estrazione di tali informazioni non richiede l'esecuzione del file (**analisi statica**)



Ottenere informazioni da un file binario

- Il comando **file** determina il tipo del file binario e l'architettura per cui è stato compilato
- Uso: **file nomefile**

```
barbara@barbara-VirtualBox:~/Scaricati/SS2$ file /bin/bash
/bin/bash: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically link
ed, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]
=6f072e70e3e49380ff4d43cdde8178c24cf73daa, stripped
```



Ottenere informazioni da un file binario

- Il comando **strings** consente di collezionare tutte le stringhe presenti in un file binario
 - Stampa tutte le sequenze di caratteri stampabili con almeno 4 caratteri
- Le stringhe collezionate possono fornire informazioni su dati segreti
- Uso: **strings nomefile**



Ottenerne informazioni da un file binario

- Ad esempio si consideri il seguente programma

```
CC> ./guessmyname
Guess my name: Paulo
I am sorry, this is not my name!
Please, try again.
CC> █
```

- Usando il comando **strings** si scopre facilmente il valore del nome da indovinare

```
CC> strings guessmyname
/lib64/ld-linux-x86-64.so.2
libc.so.6
__isoc99_scanf
puts
printf
malloc
__cxa_finalize
strcmp
__libc_start_main
GLIBC_2.7
GLIBC_2.2.5
_ITM_deregisterTMClockTable
__gmon_start__
_ITM_registerTMClockTable
u+UH
[]A\A\A^A_
Calef
Guess my name:
%10s
Well done! You have guessed my name!
I am sorry, this is not my name!
Please, try again.
_:*3$"
GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.8060
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
```



Ottenere informazioni da un file binario

- Possiamo usare l'informazione ottenuta per "indovinare" il nome....

```
CC> ./guessmyname
Guess my name: Calef
Well done! You have guessed my name!
CC> █
```



Ottenere informazioni da un file binario

- Non bisogna mai inserire **dati segreti** nel codice sorgente!

```
int main(int argc, char** argv) {
    char* name = "Calef";
    char* input = malloc(10);

    printf("Guess my name: ");
    scanf("%10s",input);
    if (strcmp(name,input)==0) {
        printf("Well done! You have guessed my name!\n");
    } else {
        printf("I am sorry, this is not my name!\n");
        printf("Please, try again.\n");
    }
}
```



Ottenere informazioni da un file binario

- Il comando **objdump** fornisce diverse informazioni presenti in un file binario:
 - Informazioni presenti nell'header del file

```
CC> objdump -f /bin/bash

/bin/bash:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x000000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000000000030430
```

- Contenuto di specifiche sezioni del file

```
CC> objdump -s -j .rodata /bin/bash | more

/bin/bash:      file format elf64-x86-64

Contents of section .rodata:
de000 01000200 474e5520 62617368 2c207665 ....GNU bash, ve
de010 7273696f 6e202573 2d282573 290a0078 rsion %s-(%s)..x
de020 38365f36 342d7063 2d6c696e 75782d67 86_64-pc-linux-g
de030 6e750047 4e55206c 6f6e6720 6f707469 nu.GNU long opti
de040 6f6e733a 0a00092d 2d25730a 00536865 ons:....-%s..She
de050 6c6c206f 7074696f 6e733a0a 00092d25 ll options:....-%
de060 73206f72 202d6f20 6f707469 6f6e0a00 s or -o option..
de070 72756e5f 6f6e655f 636f6d6d 616e6400 run_one_command.
de080 2d630072 62617368 00492068 61766520 -c.rbash.I have
de090 6e6f206e 616d6521 003f3f68 6f73743f no name!..?host?
de0a0 6e6f206e 616d6521 003f3f68 6f73743f no name!..?host?
```



Ottenere informazioni da un file binario

- Il comando **readelf** fornisce diverse informazioni presenti in un file binario:
 - Informazioni presenti nell'header del file

```
CC> readelf -h /bin/bash
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: DYN (Shared object file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x30430
  Start of program headers: 64 (bytes into file)
  Start of section headers: 1181528 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 13
  Size of section headers: 64 (bytes)
  Number of section headers: 30
  Section header string table index: 29
```



Ottenere informazioni da un file binario

- Il comando **readelf** fornisce diverse informazioni presenti in un file binario:
 - Informazioni presenti in particolari sezioni del file

```
CC> readelf -x .rodata guessmyname

Hex dump of section '.rodata':
 0x00002000 01000200 00000000 43616c65 66004775 .....Calef.Gu
 0x00002010 65737320 6d79206e 616d653a 20002531 ess my name: .%1
 0x00002020 30730000 00000000 57656c6c 20646f6e 0s.....Well don
 0x00002030 65212059 6f752068 61766520 67756573 e! You have gues
 0x00002040 73656420 6d79206e 616d6521 00000000 sed my name!....
 0x00002050 4920616d 20736f72 72792c20 74686973 I am sorry, this
 0x00002060 20697320 6e6f7420 6d79206e 616d6521 is not my name!
 0x00002070 00506c65 6173652c 20747279 20616761 .Please, try aga
in..
```

CC> █



Analisi statica e dinamica

- A volte però le informazioni ottenute con l'analisi statica non bastano per scoprire il **punto debole** di un programma
- In tali casi, è necessario esaminare il programma mentre è in esecuzione (**analisi dinamica**)
 - Un tool che consente tale esame è **gdb**



Reverse engineering

- Per analizzare a fondo il comportamento di un file eseguibile è utile dare uno sguardo al suo
 - codice assembly
 - codice sorgente
- Ciò può essere fatto mediante **disassemblatori** e **decompilatori**



Disassemblatori

- I **disassemblatori** sono spesso utilizzati per analizzare programmi eseguibili alla ricerca di vulnerabilità
 - Esempi: **gdb**, **objdump**
- Abbiamo visto un esempio di utilizzo nelle sfide CTF su stack-based buffer overflow
 - Abbiamo analizzato la memoria allocata al programma
 - Abbiamo investigato sulle relazioni spaziali delle variabili in uso
 - Abbiamo determinato il layout dello stack



Disasembatori

(gdb) **disassemble main**

Dump of assembler code for function main:

```
0x08048408 <main+0>: push    %ebp
0x08048409 <main+1>: mov     %esp,%ebp
0x0804840b <main+3>: and    $0xfffffffff0,%esp
0x0804840e <main+6>: sub    $0x50,%esp
0x08048411 <main+9>: lea     0x10(%esp),%eax%
0x08048415 <main+13>: mov    %eax,(%esp)
0x08048418 <main+16>: call   0x804830c <gets@plt>
0x0804841d <main+21>: leave
0x0804841e <main+22>: ret
End of assembler dump.
```



Decompiler

- I **decompiler** rendono ancora più semplice la ricerca di vulnerabilità, poichè consentono l'ispezione del codice sorgente
- Il codice ottenuto talvolta necessita di essere reso più leggibile mediante
 - Modifica della **signature delle funzioni** (valore di ritorno e argomenti)
 - Modifica di nomi e tipi delle **variabili**



Ghidra

- Tra i tool disponibili per la decompilazione, useremo **Ghidra**
 - Si tratta di un tool rilasciato nel 2019 dall'NSA
 - E' un prodotto open-source, disponibile al link <https://ghidra-sre.org/>
 - Useremo la versione 9.2.2
 - Per usarlo è necessario innanzitutto installare **Java Development Kit 11**
 - NOTA: JDK 20 crea problemi!



Ghidra

- Ghidra non necessita di alcuna installazione
 - E' sufficiente scaricare il pacchetto ed estrarlo



- Dopo l'estrazione, basta avviare il programma
 - **ghidraRun.bat** su Windows
 - **ghidraRun** su Mac o Linux

Viene caricata l'interfaccia grafica di Ghidra



CTF: Slow Printer

- Descrizione della sfida:
 - Ci viene fornito un file binario che, mandato in esecuzione, procede alla stampa della bandierina da catturare
 - Problema: ci vuole tempo, **molto tempo...**
 - Riusciamo a catturare la bandierina in tempi brevi?



CTF: Slow Printer

- Innanzitutto, scarichiamo il file binario `slow_printer` relativo alla sfida sulla nostra macchina Linux
- Poi, visualizziamo i suoi metadati con `ls -la`

```
-rwx-rw-r-- 1 barbara barbara 14560 mar 16 12:07 slow_printer
```

- Modifichiamo i permessi in modo che il file possa essere eseguito

```
chmod u+x slow_printer
```

```
-rwxrwx-r-- 1 barbara barbara 14560 mar 16 12:07 slow_printer
```



CTF: Slow Printer

- Ora possiamo mandare il file in esecuzione

```
barbara@barbara-VirtualBox:~/Scaricati$ ./slow_printer
Your flag is:
CCIT{tim3}
```

- Il programma inizia a stampare la flag, ma la stampa sembra **rallentata**
- Probabilmente, c'è la chiamata a qualche funzione che provoca un **ritardo**



CTF: Slow Printer

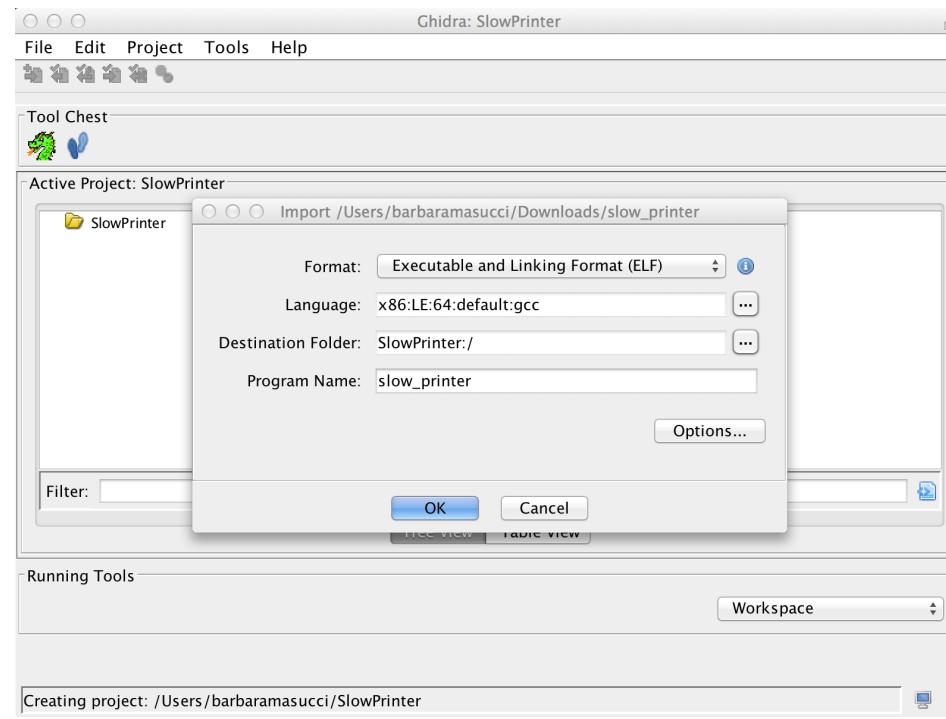
- Idea: “eliminare” la chiamata che ritarda l'esecuzione della stampa della flag
- Tra i tool disponibili, decidiamo di usare **Ghidra**



CTF: Slow Printer

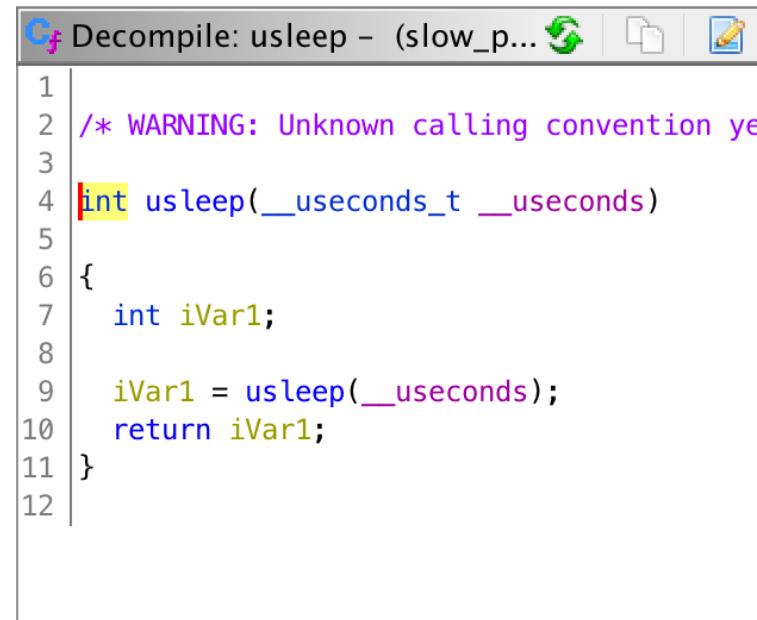
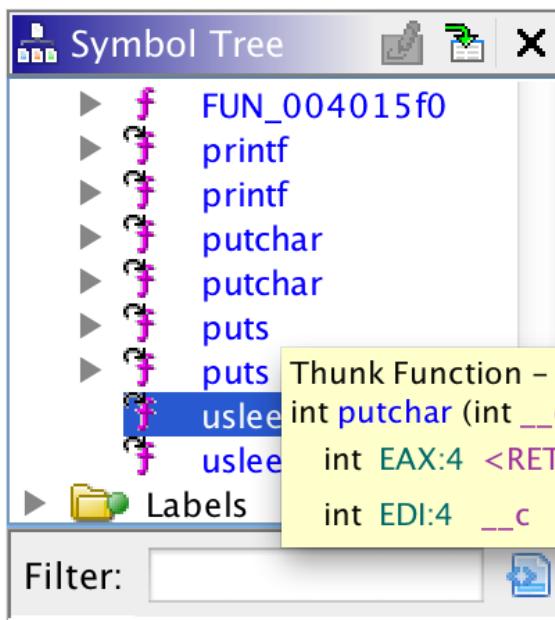
➤ Dopo aver lanciato Ghidra:

- New Project -> Non Shared e diamo un nome al progetto
- Trasciniamo il file slow_printer nel progetto



CTF: Slow Printer

- Da **Symbol Tree**, scegliamo **Functions** e vediamo la lista delle funzioni
 - Notiamo la funzione **usleep**



Decompile: usleep - (slow_p...)

```
1 /* WARNING: Unknown calling convention yet!
2
3
4 int usleep(__useconds_t __useconds)
5 {
6     int iVar1;
7
8     iVar1 = usleep(__useconds);
9     return iVar1;
10}
11
12
```



CTF: Slow Printer

- Idea: Modificare il comportamento della funzione `usleep` 
- Come fare?
 - Possiamo usare la variabile `LD_PRELOAD` per **ridefinire dinamicamente** la funzione `usleep` (function overriding) senza dover ricompilare il sorgente !



CTF: Slow Printer

- Il file usleep.c contiene una nuova implementazione delle funzione `usleep`

```
#include <unistd.h>
#include <sys/types.h>

int usleep(useconds_t usec) {
    ;
}
```

- In pratica, la funzione non fa nulla



CTF: Slow Printer

- Creiamo il file usleep.c e scriviamo la nuova implementazione
`nano usleep.c`
- Ora generiamo la libreria condivisa usando `gcc`
`gcc -shared -fPIC -o usleep.so usleep.c`
- Opzioni usate
 - `-shared`: genera un oggetto linkabile a tempo di esecuzione e condivisibile con altri oggetti
 - `-fPIC`: genera codice indipendente dalla posizione (Position Independent Code), rilocabile ad un indirizzo di memoria arbitrario



CTF: Slow Printer

- Per caricare anticipatamente la **libreria condivisa** usleep.so, modifichiamo la variabile LD_PRELOAD:
`export LD_PRELOAD=./usleep.so`
- Ora mandiamo in esecuzione il programma
`./slow_printer`
- Il programma stampa la flag: sfida vinta!



CTF: Slow Printer

➤ Conclusioni:

- L'uso di un **tool di reversing** ci ha consentito di risalire al sorgente, a partire dal binario eseguibile
- Mediante ispezione del sorgente abbiamo compreso che la causa del ritardo della stampa era la funzione **usleep**
- Abbiamo quindi modificato il comportamento della funzione **usleep** vincendo la sfida



CTF: Pacman

- Descrizione della sfida:
 - Giochiamo a Pacman!
- Materiale fornito:
 - Binario eseguibile: **pacman**



CTF: Pacman

- Innanzitutto, scarichiamo il file binario **pacman** relativo alla sfida sulla nostra macchina Linux
- Poi, visualizziamo i suoi metadati con **ls -la**

```
-rwx-rw-r-- 1 barbara barbara 14504 mar 23 10:57 pacman
```

- Modifichiamo i permessi in modo che il file possa essere eseguito

chmod u+x pacman

```
-rwxrwx-r-- 1 barbara barbara 14504 mar 23 10:57 pacman
```



CTF: Pacman

- Mandiamo il file in esecuzione

```
barbara@barbara-VirtualBox:~/Scaricati$ ./pacman
barbara@barbara-VirtualBox:~/Scaricati$ █
```

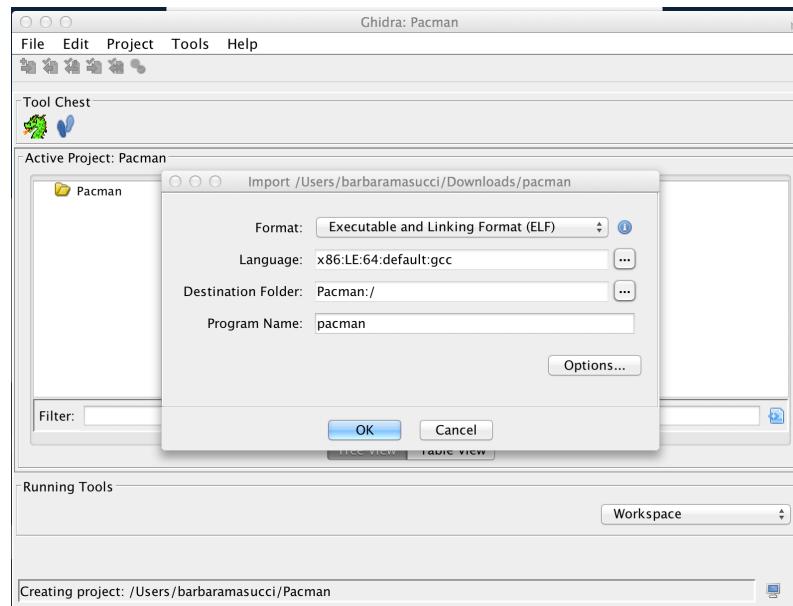
- Il programma termina anche se proviamo a passargli un input

```
barbara@barbara-VirtualBox:~/Scaricati$ ./pacman input
Game over.
barbara@barbara-VirtualBox:~/Scaricati$ █
```



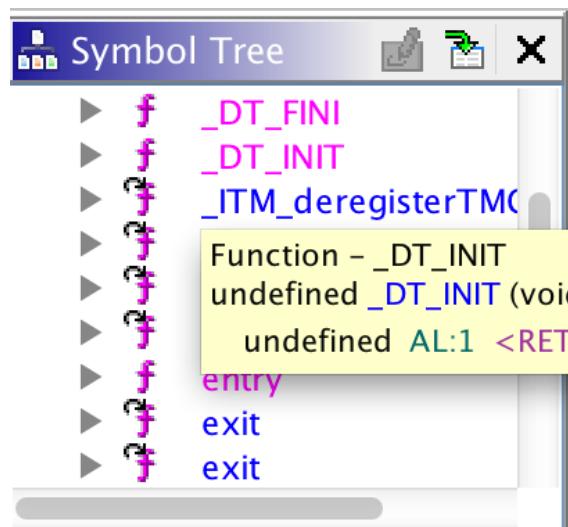
CTF: Pacman

- Analizzeremo il binario usando **Ghidra**
 - New Project -> Non Shared e diamo un nome al progetto
 - Trasciniamo il file **pacman** nel progetto



CTF: Pacman

- Da **Symbol Tree**, scegliamo **Functions** e vediamo la lista delle funzioni
- Poichè non è presente la funzione **main**, cerchiamo **entry**



CTF: Pacman

➤ Decompliamo **entry**

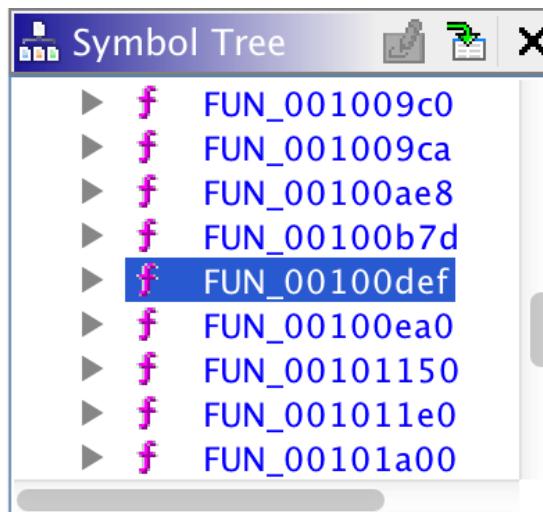
C# Decompile: entry - (pacman)

```
1 void entry(undefined8 param_1,undefined8 param_2,undefined8 param_3)
2 {
3     undefined8 in_stack_00000000;
4     undefined auStack8 [8];
5
6     __libc_start_main(FUN_00100def,in_stack_00000000,&stack0x00000008,FUN_00101a00,FUN_00101a70,
7                      param_3,auStack8);
8     do {
9         /* WARNING: Do nothing block with infinite loop */
10    } while( true );
11 }
12 }
```



CTF: Pacman

- Guardiamo quale funzione è invocata da
`_libc_start_main()`
- Troviamo la funzione `FUN_00100def`
 - Cerchiamola nel Symbol Tree e decompiliamola



CTF: Pacman

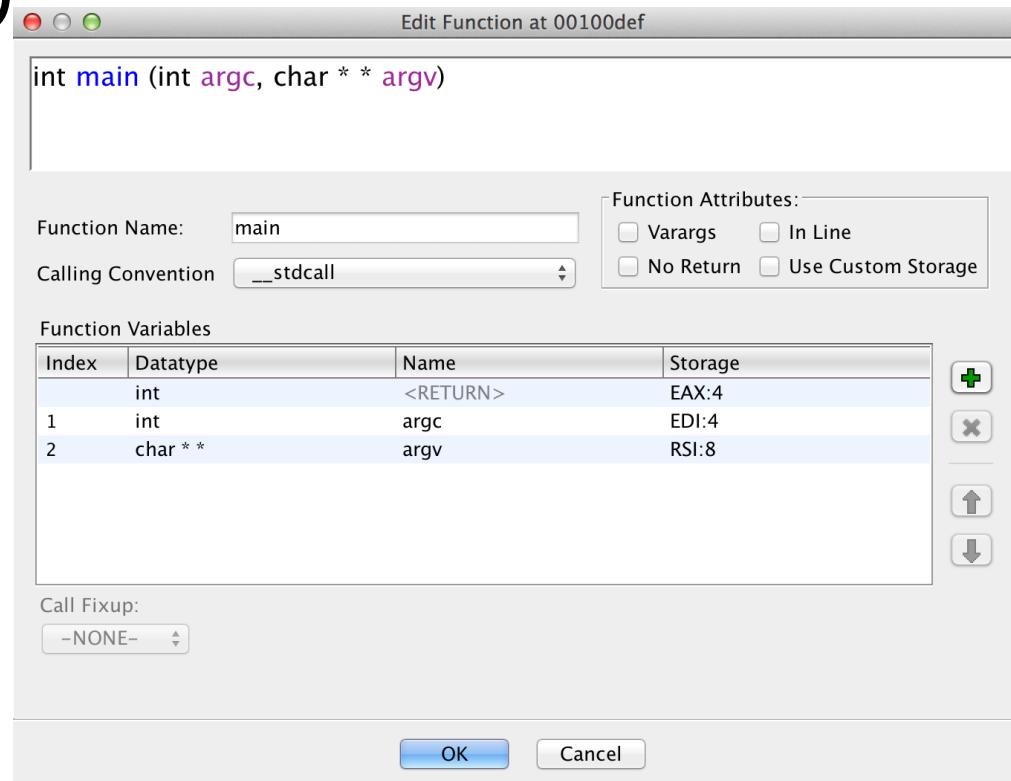
Il codice non è molto leggibile

```
C# Decompile: FUN_00100def - (pacman)
1
2 undefined8 FUN_00100def(int param_1, long param_2)
3
4 {
5     long lVar1;
6     char *pcVar2;
7     undefined **local_10;
8
9     if ((param_1 == 2) && (lVar1 = ptrace(PTRACE_TRACEME,0,1,0), lVar1 != -1)) {
10         clock_gettime(0,(timespec *)&DAT_003030e0);
11         local_10 = &PTR_s_pacmanpacman_00303060;
12         while (*local_10 != (char *)0x0) {
13             pcVar2 = strdup(*local_10);
14             *local_10 = pcVar2;
15             local_10 = local_10 + 1;
16         }
17         FUN_00100b7d(*(undefined8 *)(&param_2 + 8));
18         return 0;
19     }
20     return 0xffffffff;
21 }
22 }
```



CTF: Pacman

➤ Rendiamo il codice più leggibile, modificando la signature della funzione (facendo right-click su essa)



CTF: Pacman

Ora il codice del `main` appare così

```
Ghidra: Decompile: main - (pacman)
1
2 int main(int argc,char **argv)
3
4 {
5     long lVar1;
6     char *pcVar2;
7     undefined **local_10;
8
9     if ((argc == 2) && (lVar1 = ptrace(PTRACE_TRACEME,0,1,0), lVar1 != -1)) {
10        clock_gettime(0,(timespec *)&DAT_003030e0);
11        local_10 = &PTR_s_pacmanpacman_00303060;
12        while (*local_10 != (char *)0x0) {
13            pcVar2 = strdup(*local_10);
14            *local_10 = pcVar2;
15            local_10 = local_10 + 1;
16        }
17        FUN_00100b7d(argv[1]);
18        return 0;
19    }
20    return -1;
21 }
22 }
```



CTF: Pacman

- All'inizio del `main` notiamo un controllo sul numero di parametri in input al programma

```
C:\ Decompile: main - (pacman)
1 int main(int argc,char **argv)
2 {
3     long lVar1;
4     char *pcVar2;
5     undefined **local_10;
6
7     if ((argc == 2) && (lVar1 = ptrace(PTRACE_TRACEME,0,1,0), lVar1 != -1)) {
8         clock_gettime(0,(timespec *)&DAT_003030e0);
9         local_10 = &PTR_s_pacmanpacman_00303060;
10        while (*local_10 != (char *)0x0) {
11            pcVar2 = strdup(*local_10);
12            *local_10 = pcVar2;
13            local_10 = local_10 + 1;
14        }
15        FUN_00100b7d(argv[1]);
16        return 0;
17    }
18    return -1;
19 }
```

- Se il numero di parametri è diverso da 2, il programma termina
- Quindi il programma **si aspetta un input**



CTF: Pacman

➤ Poi, notiamo l'invocazione della chiamata di sistema **ptrace**

```
C Decompile: main - (pacman)
1
2 int main(int argc,char **argv)
3
4 {
5     long lVar1;
6     char *pcVar2;
7     undefined **local_10;
8
9     if ((argc == 2) && (lVar1 = ptrace(PTRACE_TRACEME,0,1,0), lVar1 != -1)) {
10        clock_gettime(0,(timespec *)&DAT_003030e0);
11        local_10 = &PTR_s_pacmanpacman_00303060;
12        while (*local_10 != (char *)0x0) {
13            pcVar2 = strdup(*local_10);
14            *local_10 = pcVar2;
15            local_10 = local_10 + 1;
16        }
}
```

➤ Cosa fa la chiamata **ptrace**?

- Leggiamo il manuale



CTF: Pacman

- La chiamata di sistema **ptrace** consente a un processo (**tracer**) di osservare l'esecuzione di un altro processo (**tracee**)

```
lVar1 = ptrace(PTRACE_TRACEME, 0, 1, 0),
```

- Nel nostro caso, il processo cerca di fare il **ptrace** di sè stesso

- Quindi, indirettamente, controlla se c'è un altro processo (ad es. un **debugger**) attaccato ad esso
- Se questo è il caso, **ptrace** dà in output -1
- Si tratta di un **anti-debugging trick**



CTF: Pacman

- Se il programma ha ricevuto un input e non c'è un debugger collegato al processo, si procede
 - Tralasciamo le altre istruzioni e passiamo all'invocazione della funzione **FUN_00100b7d**

C# Decompile: main – (pacman)

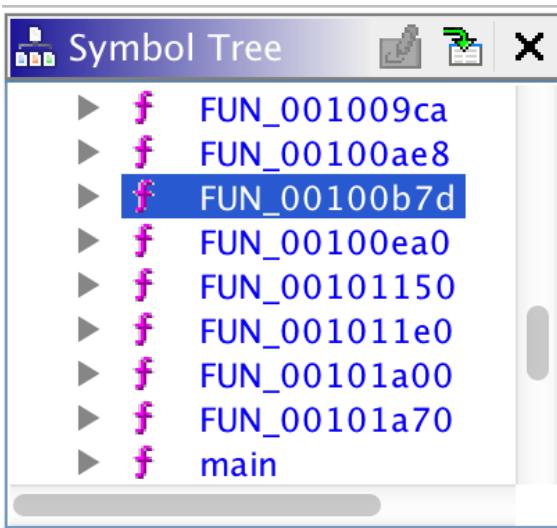
```
1 int main(int argc,char **argv)
2 {
3     long lVar1;
4     char *pcVar2;
5     undefined **local_10;
6
7     if ((argc == 2) && (lVar1 = ptrace(PTRACE_TRACEME,0,1,0), lVar1 != -1)) {
8         clock_gettime(0,(timespec *)&DAT_003030e0);
9         local_10 = &PTR_s_pacmanpacman_00303060;
10        while (*local_10 != (char *)0x0) {
11            pcVar2 = strdup(*local_10);
12            *local_10 = pcVar2;
13            local_10 = local_10 + 1;
14        }
15        FUN_00100b7d(argv[1]);
16        return 0;
17    }
18    return -1;
19 }
```



CTF: Pacman

Troviamo la funzione **FUN_00100b7d**

- Cerchiamola nel Symbol Tree e decompiliamola



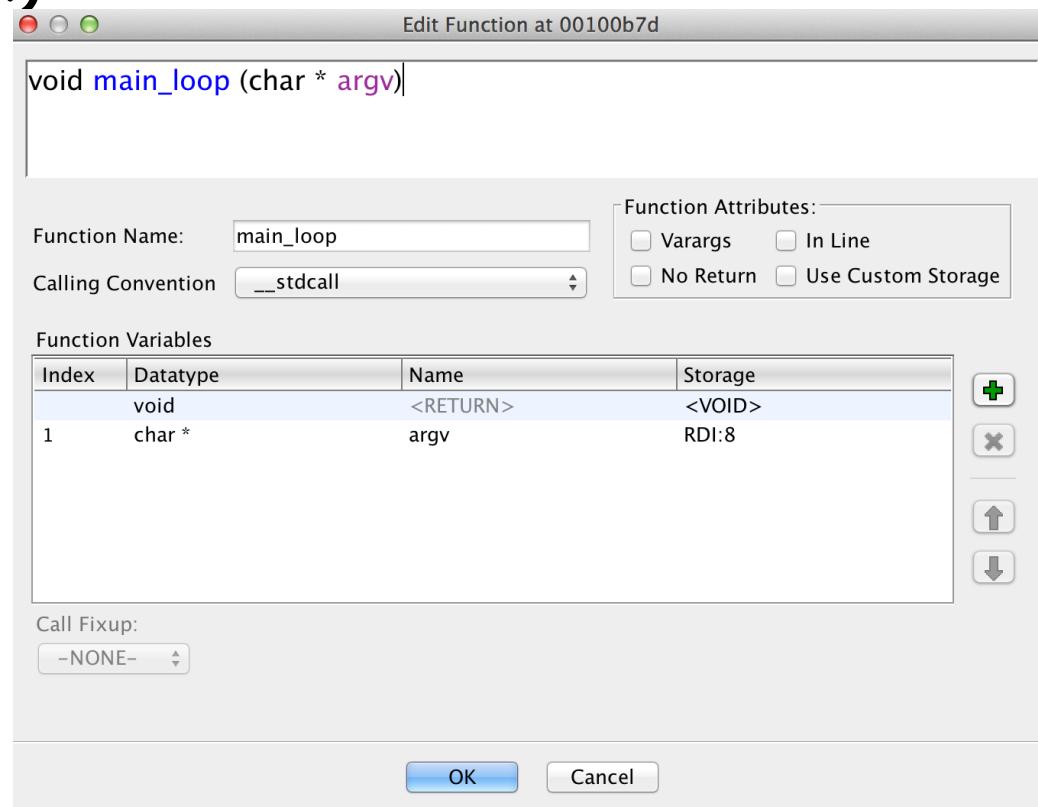
The image shows a decompiler window titled 'Decompile: FUN_00100b7d - (pacman)'. The code is as follows:

```
1 void FUN_00100b7d(char *param_1)
2 {
3     char *pcVar1;
4     char cVar2;
5     char cVar3;
6     ushort **ppuVar4;
7     long in_FS_OFFSET;
8     char *local_150;
9     int local_144;
10    int local_140;
11    int local_13c;
12    timespec local_138;
13    char local_128 [4];
14    undefined local_124;
15    undefined local_123;
16    undefined local_122;
17    long local_20;
18
19    local_20 = *(long *)(&in_FS_OFFSET + 0x28);
20    memset(local_128, 0, 0x100);
21    local_144 = 0;
22    local_140 = 1;
23    local_13c = 1;
24    local_150 = param_1;
25    do {
26        clock_gettime(0, &local_138);
27        cVar2 = (&PTR_s_pacmanpacman_00303060)[local_13c][local_140];
28        cVar3 = FUN_00100ae8(start_time, DAT_003030e8, local_138.tv_sec, local_138.tv_nsec);
29        cVar3 = cVar3 + cVar2;
30        switch(cVar3) {
31            case 'a':
32            case 'c':
33            case 'm':
34            case 'n':
35            case 'o':
```



CTF: Pacman

➤ Rendiamo il codice più leggibile, modificando la signature della funzione (facendo right-click su essa)



CTF: Pacman

➤ Notiamo il blocco di codice finale

- I caratteri 'j', 'k', 'h', 'l' incrementano e decrementano due variabili (`local_13c` e `local_140`)
- Questi caratteri sono usati nell'editor `vi` per muovere il cursore

```
63 if (cVar2 == 'j') {  
64     local_13c = local_13c + 1;  
65 }  
66 else {  
67     if (cVar2 < 'k') {  
68         if (cVar2 == 'h') {  
69             local_140 = local_140 + -1;  
70         }  
71     }  
72     else {  
73         if (cVar2 == 'k') {  
74             local_13c = local_13c + -1;  
75         }  
76         else {  
77             if (cVar2 == 'l') {  
78                 local_140 = local_140 + 1;  
79             }  
80         }  
81     }  
82 }  
83 } while( true );  
84 }  
85 }
```



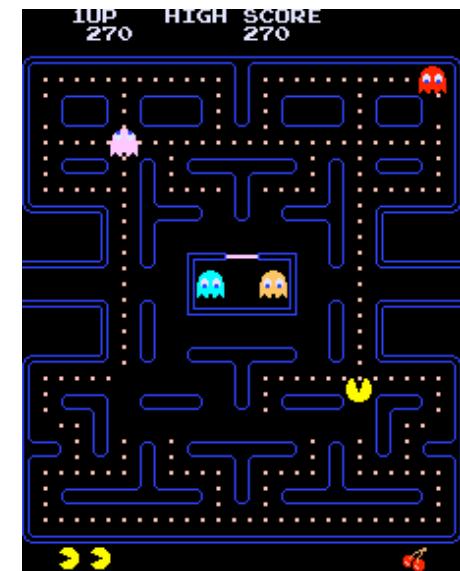
CTF: Pacman

➤ Poichè il nome della sfida è **Pacman**...
può darsi che questi caratteri servano
a **muovere il cursore** in una sorta di labirinto?



CTF: Pacman

- Prima abbiamo visto che **il programma si aspetta un input**
- Deduciamo che l'input debba essere una sequenza di caratteri **'j', 'k', 'h', 'l'**
- Quale sequenza ci fa vincere la sfida?



CTF: Pacman

➤ Tornando in alto nel codice, rinominiamo le variabili

- local_13c → y
- local_140 → x
- PTR_s_pacmanpacman_003030360 → maze

```
local_144 = 0;
local_140 = 1;|
local_13c = 1;
local_150 = param_1;
do {
    clock_gettime(0,&local_138);
    cVar2 = (&PTR_s_pacmanpacman_00303060)[local_13c][local_140];
```



CTF: Pacman

➤ Il codice ora appare così

```
x = 1;  
y = 1;  
local_150 = param_1;  
do {  
    clock_gettime(0,&local_138);  
    cVar2 = (&maze)[y][x];
```

- Quindi, il gioco parte dalla casella (1,1) e la posizione cambia in base all'input dato al programma (`argv[1]`)
- Più in basso, notiamo un controllo sul valore della casella su cui si finisce in base all'input fornito



CTF: Pacman

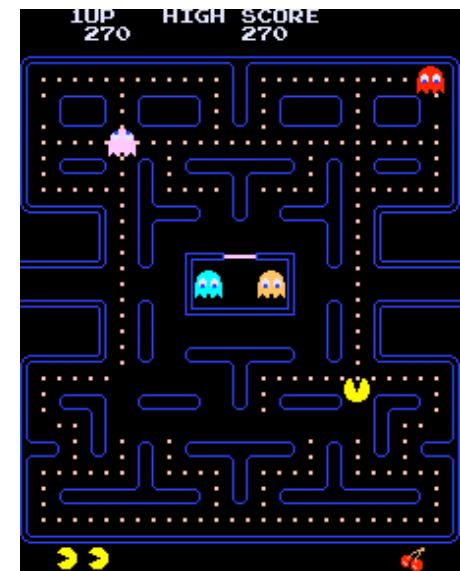
- Se si finisce su una delle caselle contenenti uno dei caratteri della parola “pacman”, il programma termina e stampa
Game over

```
switch(cVar3) {  
    case 'a':  
    case 'c':  
    case 'm':  
    case 'n':  
    case 'p':  
        switchD_00100c74_caseD_61:  
            puts("Game over.");  
            /* WARNING: Subroutine does not return */  
            exit(-1);
```



CTF: Pacman

- Quindi **dobbiamo spostarci nel labirinto senza toccare le caselle contenenti i caratteri 'a', 'c', 'm', 'n', 'p'**
- Per capire come muoverci, dobbiamo esaminare il contenuto della variabile **maze**

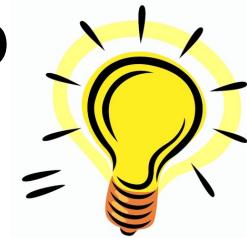


CTF: Pacman

- Per esaminare il contenuto di `maze` c'è necessità che il programma sia in esecuzione
- Potremmo usare `gdb` ma...come aggirare l'anti-debugging trick?
- Se non troviamo una soluzione, `ptrace` individuerà la presenza del debugger e il programma terminerà



CTF: Pacman

- Idea: Modificare il comportamento della funzione `ptrace` 
- Come fare?
 - Possiamo usare la variabile `LD_PRELOAD` per ridefinire dinamicamente la funzione `ptrace` (function overriding) senza dover ricompilare il sorgente



CTF: Pacman

- Il file ptrace.c contiene una nuova implementazione delle funzione `ptrace`

```
#include <unistd.h>
#include <sys/types.h>

int ptrace(int i, int j, int k, int l) {
    return 0;
}
```

- In pratica, la funzione non fa nulla



CTF: Pacman

- Creiamo il file ptrace.c e scriviamo la nuova implementazione
`nano ptrace.c`
- Ora generiamo la libreria condivisa usando `gcc`
`gcc -shared -fPIC -o ptrace.so ptrace.c`
- Opzioni usate
 - `shared`: genera un oggetto linkabile a tempo di esecuzione e condivisibile con altri oggetti
 - `fPIC`: genera codice indipendente dalla posizione (Position Independent Code), rilocabile ad un indirizzo di memoria arbitrario



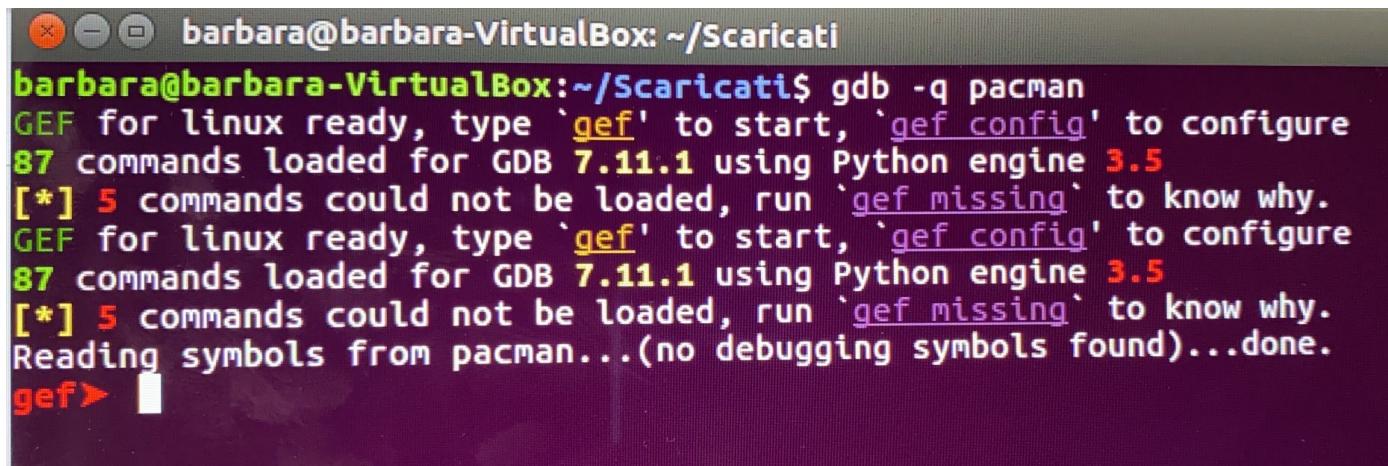
CTF: Pacman

- Prima di avviare **gdb**, assicuriamoci di aver installato **GEF** (**GDB Enhanced Feature**)
 - <https://gef.readthedocs.io/en/master/>
- **GEF** arricchisce i comandi di **gdb** e semplifica i processi di
 - Analisi dinamica
 - Sviluppo di exploit



CTF: Pacman

- Avviamo `gdb` per analizzare `pacman`
`gdb -q pacman`



```
barbara@barbara-VirtualBox:~/Scaricati$ gdb -q pacman
GEF for linux ready, type `gef' to start, `gef config' to configure
87 commands loaded for GDB 7.11.1 using Python engine 3.5
[*] 5 commands could not be loaded, run `gef missing' to know why.
GEF for linux ready, type `gef' to start, `gef config' to configure
87 commands loaded for GDB 7.11.1 using Python engine 3.5
[*] 5 commands could not be loaded, run `gef missing' to know why.
Reading symbols from pacman... (no debugging symbols found)...done.
gef>
```

- Il programma si arresta in attesa di un comando



CTF: Pacman

- Inseriamo un breakpoint alla funzione invocata subito dopo `ptrace` nel `main`

```
int main(int argc,char **argv)
{
    long lVar1;
    char *pcVar2;
    undefined **local_10;

    if ((argc == 2) && (lVar1 = ptrace(PTRACE_TRACEME,0,1,0), lVar1 != -1)) {
        clock_gettime(0,(timespec *)&DAT_003030e0);
```

```
gef> b clock_gettime
Breakpoint 1 at 0x840
gef> █
```

- Quando `pacman` sarà in esecuzione, si arresterà al breakpoint



CTF: Pacman

- Prima di mandare `pacman` in esecuzione,
dobbiamo aggirare il controllo anti-debugging
di `ptrace`
- Settiamo la variabile di ambiente `LD_PRELOAD`
`set environment LD_PRELOAD ./ptrace.so`
gef ➤ set environment LD_PRELOAD ./ptrace.so
- In tal modo provocheremo l'esecuzione della
`ptrace` modificata (che non fa nulla)



CTF: Pacman

- Ora mandiamo `pacman` in esecuzione con un input qualsiasi

run input

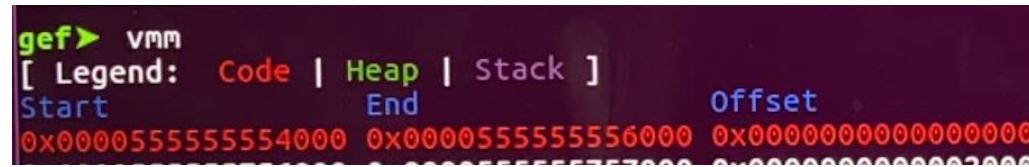
```
[#0] Id 1, Name: "pacman", stopped 0x7fffff7920910 in __GI_clock_gettime (), reason: BREAKPOINT
[ #0]                                     trace
[ #0] 0x7fffff7920910 → __GI_clock_gettime(clock_id=0x0, tp=0x5555557570e0)
[ #1] 0x55555554e40 → lea rax, [rip+0x202219]          # 0x555555757060
[ #2] 0x7fffff782b840 → __libc_start_main(main=0x55555554def, argc=0x2, argv=0x7fffffffde98, init=<optimized
    out>, fini=<optimized out>, rtld_fini=<optimized out>, stack_end=0x7fffffffde88)
[ #3] 0x555555548ea → hlt
gef> █
```

- Il programma, dopo l'esecuzione della `ptrace` modificata, si ferma sul breakpoint (`clock_gettime`)



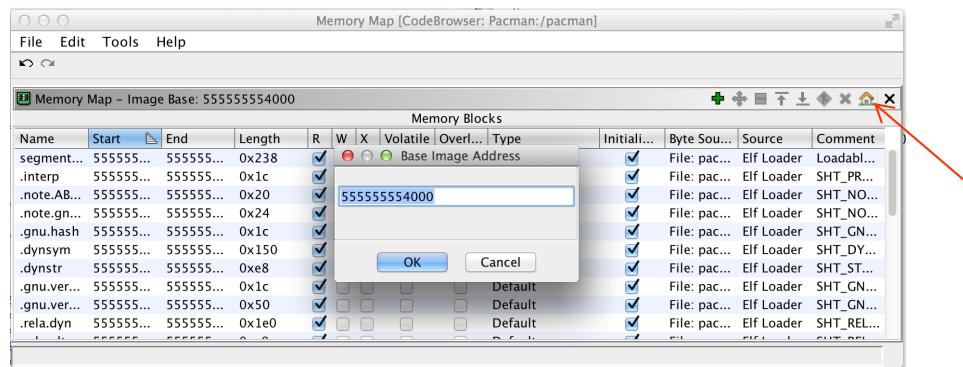
CTF: Pacman

- Recuperiamo l'indirizzo di partenza del processo in memoria con **vmm**



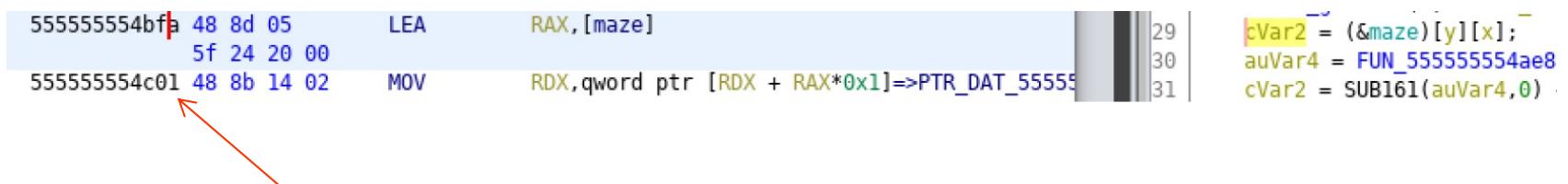
```
gef> vmm
[ Legend: Code | Heap | Stack ]
Start           End           Offset
0x000055555554000 0x000055555556000 0x0000000000000000
```

- Sincronizziamo **Ghidra** con **gdb**
- Selezioniamo Window → Memory
 - Clicchiamo sulla casetta e inseriamo l'indirizzo



CTF: Pacman

- In **Ghidra** vediamo qual è l'indirizzo dell'istruzione successiva all'accesso alla variabile **maze**



```
5555555554bf 48 8d 05      LEA      RAX, [maze]          29
                5f 24 20 00
5555555554c01 48 8b 14 02    MOV      RDX,qword ptr [RDX + RAX*0x1] => PTR_DAT_55555554c01 30
                                31
cVar2 = (&maze)[y][x];
auVar4 = FUN_5555555554ae8
cVar2 = SUB16l(auVar4, 0)
```

- In **gdb**, inseriamo un breakpoint a tale indirizzo

b *0x00005555555554c01

```
gef> b *0x00005555555554c01
Breakpoint 2 at 0x5555555554c01
gef>
```



CTF: Pacman

- Proseguiamo con l'esecuzione del programma finchè non giungiamo al **breakpoint 2**
 - In tal modo ci troveremo al punto in cui il programma accede alla variabile `maze`
- E' necessario digitare il comando **c** per 4 volte
 - Il programma si ferma tre volte sul **breakpoint 1** e la quarta volta sul **breakpoint 2**



CTF: Pacman

- Ora visualizziamo il contenuto del registro **rax**
telescope \$rax

```
gef> telescope $rax
0x0000555555757060 +0x0000: 0x0000555555758010 → "pacmanpacman"
0x0000555555757068 +0x0008: 0x0000555555758030 → "cho4$aaagioc"
0x0000555555757070 +0x0010: 0x0000555555758050 → "caaargmmmmmc"
0x0000555555757078 +0x0018: 0x0000555555758070 → "cz1pia66600c"
0x0000555555757080 +0x0020: 0x0000555555758090 → "cx4p2c00666c"
0x0000555555757088 +0x0028: 0x00005555557580b0 → "cg8a2pacmanc"
0x0000555555757090 +0x0030: 0x00005555557580d0 → "ci_cz737373c"
0x0000555555757098 +0x0038: 0x00005555557580f0 → "co1pacpacconc"
0x00005555557570a0 +0x0040: 0x0000555555758110 → "c4_____pac0ac"
0x00005555557570a8 +0x0048: 0x0000555555758130 → "czxgioN1234c"
gef>
```

- Abbiamo ottenuto il contenuto della variabile **maze**



Casella (1,1)
di maze
(labirinto)

CTF: Pacman

➤ Guardando il contenuto di `maze`, capiamo che la sequenza giusta da fornire in input è

111jjjjj11111jjjh

```
gef> telescope $rax
0x0000555555757060 +0x0000: 0x0000555555758010 → "pacmanpacman"
0x0000555555757068 +0x0008: 0x0000555555758030 → "chc4$aaagioc"
0x0000555555757070 +0x0010: 0x0000555555758050 → "caaarmmmmmmc"
0x0000555555757078 +0x0018: 0x0000555555758070 → "cz1piia66600c"
0x0000555555757080 +0x0020: 0x0000555555758090 → "cx4p2c00666c"
0x0000555555757088 +0x0028: 0x00005555557580b0 → "cg8a2pacmanc"
0x0000555555757090 +0x0030: 0x00005555557580d0 → "ci_cz737373c"
0x0000555555757098 +0x0038: 0x00005555557580f0 → "co1pacpacconc"
0x00005555557570a0 +0x0040: 0x0000555555758110 → "c4____pac0ac"
0x00005555557570a8 +0x0048: 0x0000555555758130 → "czxgioM1234c"
gef>
```



CTF: Pacman

- Usciamo da `gdb` (comando `q`) e digitiamo
`./pacman llljjjjjllllljjjh`
- Il programma stampa la flag: Sfida vinta!



CTF: Pacman

➤ Conclusioni:

- L'uso di un **tool di reversing** ci ha consentito di risalire al sorgente, a partire dall'eseguibile
- Mediante ispezione del sorgente abbiamo notato la presenza di un anti-debugging trick (funzione **ptrace**)
- Abbiamo quindi modificato il comportamento della funzione **ptrace** per poter esaminare il binario mediante un debugger
- L'analisi della memoria ci ha consentito di capire la struttura del labirinto (**maze**)
- Siamo stati in grado di fornire la sequenza di input corretta per vincere la sfida



Vi siete divertiti?



“That's all Folks!”

Adesso tocca a voi



Seminari: 12 Maggio

- Nebula: Level 03
- Nebula: Level 05
- Nebula: Level 06
- Nebula: Level 08
- Nebula: Level 09
- Nebula: Level 11
- Nebula: Level 12
- Nebula: Level 14
- Nebula: Level 15



Seminari: 13 Maggio

- Nebula: Level 16
- Nebula: Level 17
- Nebula: Level 18
- Nebula: Level 19



Seminari: 19 Maggio

- DVWA: File Upload
- DWVA: Javascript
- Protostar: Stack Six
- Protostar: Stack Seven
- Protostar: Format One
- Protostar: Format Two
- Protostar: Format Three
- Protostar: Format Four
- CVE-2019-0708



Seminari: 20 Maggio

- Protostar: Heap Zero
- Protostar: Heap One
- Protostar: Heap Two
- Protostar: Heap Three



Seminari: 26 Maggio

- Protostar: Final Zero
- Protostar: Final One
- Protostar: Final Two
- CVE-2020-7247



Seminari: 27 Maggio

- Heist_ML_CTF_Challenge
- Morph su Cyberchallenge.IT
- Wordpress CVE su TryHackMe
- CVE-2023-4863



Seminari: 3 Giugno

- Fusion Level 03
- Fusion Level 04
- Fusion Level 05
- Fusion Level 06
- Fusion Level 07
- PARTY DI FINE CORSO

