ENPM691 Homework 04: Automating Memory Layout Analysis with pwndbg

Kalpesh Bharat Parmar
M.Eng, Cybersecurity
University of Maryland, College Park
kalpesh@umd.edu
UMD Directory ID – kalpesh
Course and section - ENPM691 0101

***Abstract*—This paper extends Homework 03 by introducing automated debugging with GDB and the pwndbg plugin. Using a scripted workflow, we examine where C variables of different types are placed in memory and validate the results through program execution, debugger inspection, and assembly disassembly. The automation confirms the memory segmentation model in 32-bit Linux and demonstrates the efficiency of using pwndbg for reproducible debugging.**

I. System Configuration

All experiments were conducted on the following system:

- Operating System: Kali-Linux-2025.2
- Compiler: gcc 14.3.0 (Debian 14.3.0-5) with multilib support
- Debugger: GDB 16.3 with pwndbg extension
- Mode: Compiled and executed in 32-bit mode

II. Purpose

The goal of this assignment is to automate memory layout analysis using pwndbg. Building on Homework 03, the program `address_layout.c` is examined through automated debugger scripting to determine the placement of local, global, static, and heap variables in memory.

III. Method

A. Compilation

The program was compiled with:

```
gcc -m32 -g address_layout.c -o address_layout
```

Listing 1. Compilation Command

Explanation:

- `-m32`: Produces a 32-bit executable.
- `-g`: Includes debugging symbols for GDB [1].
- `-o address_layout`: Specifies the output binary name.

B. Automation with pwndbg

A GDB script (`hw4_pwndbg.gdb`) was used to:

1) Log program output.
2) Restart at `main` and inspect variables.
3) Disassemble `main()`.
4) Show memory maps using `vmmap` [2].

The script was executed with the command:

```
gdb -q -x hw4_pwndbg.gdb
```

Here:

- `-q` launches GDB in *quiet mode*, suppressing the startup banner and extra messages.
- `-x hw4_pwndbg.gdb` tells GDB to execute all commands from the given script file on startup, automating the process instead of requiring manual input.

This ensured clean, reproducible output suitable for logging into `hw4_log.txt`.

IV. Results

A. Program Execution

The program output is shown in Fig. 1.

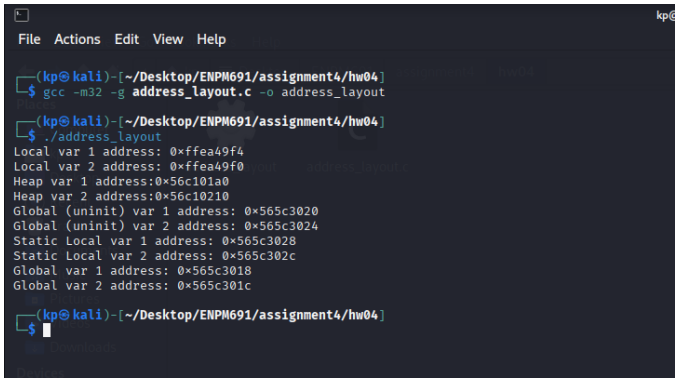

Fig. 1. Execution output showing variable addresses.

B. Debugger Inspection

Variable addresses printed in GDB are shown in Fig. 2.

C. Summary of Variable Segments

Table I summarizes representative addresses and their segments. Although numeric values vary due to ASLR(Address Space Layout Randomization), the pattern remains consistent.

D. Assembly Inspection

The disassembly of `main()` is shown in Fig. 3.

```
File  Actions  Edit  View  Help
GNU gdb (Debian 16.3-1) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./address_layout...
(gdb) break main
Breakpoint 1 at 0x11ba: file address_layout.c, line 12.
(gdb) run
Starting program: /home/kp/Desktop/ENPM691/assignment3/address_layout
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at address_layout.c:12
12          int local_var_1 = 0;
(gdb) print &local_var_1
$1 = (int *) 0xffffcee4
(gdb) print &local_var_2
$2 = (int *) 0xffffcee0
(gdb) print &global_var_1
$3 = (int *) 0x56559018 <global_var_1>
(gdb) print &global_var_2
$4 = (int *) 0x5655901c <global_var_2>
(gdb) print &global_uninit_var_1
$5 = (int *) 0x56559020 <global_uninit_var_1>
(gdb) print &global_uninit_var_2
$6 = (int *) 0x56559024 <global_uninit_var_2>
(gdb) print &static_var_1
$7 = (int *) 0x56559028 <static_var_1>
(gdb) print &static_var_2
$8 = (int *) 0x5655902c <static_var_2>
(gdb) print ptr_1
$9 = (int *) 0x0
(gdb) print ptr_2
$10 = (int *) 0x0
(gdb)
```

Fig. 2. GDB output inspecting variables at function entry.

TABLE I
OBSERVED VARIABLE ADDRESSES AND MEMORY SEGMENTS

Variable	Sample Address	Segment
`local_var_1`	0xffffcee4	Stack
`local_var_2`	0xffffcee0	Stack
`ptr_1` (malloc)	0x57a481a0	Heap
`ptr_2` (malloc)	0x57a48210	Heap
`global_var_1`	0x56559018	.data
`global_var_2`	0x5655901c	.data
`global_uninit_var_1`	0x56559020	.bss
`global_uninit_var_2`	0x56559024	.bss
`static_var_1`	0x56559028	.data
`static_var_2`	0x5655902c	.data

E. Automated Script Output

Figure 4 shows output from the pwndbg script execution.

V. Discussion

The analysis confirms that:

- Locals are allocated on the stack and accessed via offsets from %ebp.
- Heap variables are created through `malloc` and reside in dynamically allocated memory.
- Global initialized variables and static locals are placed in `.data`, while uninitialized globals are in `.bss`.

The scripting approach provided three main advantages:

1) **Consistency:** Automated logging avoids manual mistakes.
2) **Efficiency:** Multiple inspections (variables, disassembly, memory map) are performed in one run.
3) **Clarity:** pwndbg formatting and `vmmap` improve readability [2].

```
File  Actions  Edit  View  Help
(gdb) disassemble main
Dump of assembler code for function main:
   0x5655619d <+0>:     lea    0x4(%esp),%ecx
   0x565561a1 <+4>:     and    $0xfffffff0,%esp
   0x565561a4 <+7>:     push   -0x4(%ecx)
   0x565561a7 <+10>:    push   %ebp
   0x565561a8 <+11>:    mov    %esp,%ebp
   0x565561aa <+13>:    push   %ebx
   0x565561ab <+14>:    push   %ecx
   0x565561ac <+15>:    sub    $0x10,%esp
   0x565561af <+18>:    call   0x565560a0 <__x86.get_pc_thunk.bx>
   0x565561b4 <+23>:    add    $0x2e40,%ebx
=> 0x565561ba <+29>:    movl   $0x0,-0x14(%ebp)
   0x565561c1 <+36>:    movl   $0x0,-0x18(%ebp)
   0x565561c8 <+43>:    sub    $0xc,%esp
   0x565561cb <+46>:    push   $0x64
   0x565561cd <+48>:    call   0x56556050 <malloc@plt>
   0x565561d2 <+53>:    add    $0x10,%esp
   0x565561d5 <+56>:    mov    %eax,-0xc(%ebp)
   0x565561d8 <+59>:    sub    $0xc,%esp
   0x565561db <+62>:    push   $0x64
   0x565561dd <+64>:    call   0x56556050 <malloc@plt>
   0x565561e2 <+69>:    add    $0x10,%esp
   0x565561e5 <+72>:    mov    %eax,-0x10(%ebp)
   0x565561e8 <+75>:    sub    $0x8,%esp
   0x565561eb <+78>:    lea    -0x14(%ebp),%eax
   0x565561ee <+81>:    push   %eax
   0x565561ef <+82>:    lea    -0x1fec(%ebx),%eax
   0x565561f5 <+88>:    push   %eax
   0x565561f6 <+89>:    call   0x56556040 <printf@plt>
   0x565561fb <+94>:    add    $0x10,%esp
   0x565561fe <+97>:    sub    $0x8,%esp
   0x56556201 <+100>:   lea    -0x18(%ebp),%eax
   0x56556204 <+103>:   push   %eax
   0x56556205 <+104>:   lea    -0x1fd3(%ebx),%eax
   0x5655620b <+110>:   push   %eax
   0x5655620c <+111>:   call   0x56556040 <printf@plt>
   0x56556211 <+116>:   add    $0x10,%esp
   0x56556214 <+119>:   sub    $0x8,%esp
   0x56556217 <+122>:   push   -0xc(%ebp)
   0x5655621a <+125>:   lea    -0x1fba(%ebx),%eax
   0x56556220 <+131>:   push   %eax
   0x56556221 <+132>:   call   0x56556040 <printf@plt>
   0x56556226 <+137>:   add    $0x10,%esp
   0x56556229 <+140>:   sub    $0x8,%esp
   0x5655622c <+143>:   push   -0x10(%ebp)
```

Fig. 3. Disassembly excerpt of `main()` showing stack offsets and malloc calls.

```
File  Actions  Edit  View  Help
┌──(kp㉿kali)-[~/Desktop/ENPM691/assignment4/hw04]
└─$ gdb -q -x hw4_pwndbg.gdb
pwndbg: loaded 211 pwndbg commands. Type pwndbg [filter] for a list.
pwndbg: created 13 GDB functions (can be used with print/break). Type help function to see them.
Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.


═══ RUN 1: program output (self-printed addresses) ═══
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Local var 1 address: 0xffffced4
Local var 2 address: 0xffffced0
Heap var 1 address:0x5655a1a0
Heap var 2 address:0x5655a210
Global (uninit) var 1 address: 0x56559020
Global (uninit) var 2 address: 0x56559024
Static Local var 1 address: 0x56559028
Static Local var 2 address: 0x5655902c
Global var 1 address: 0x56559018
Global var 2 address: 0x5655901c
[Inferior 1 (process 21883) exited normally]

(Above: addresses printed by the program itself.)

═══ RUN 2: debugger inspection at main ═══
Temporary breakpoint 1 at 0x565561ba: file address_layout.c, line 12.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, main () at address_layout.c:12
12          int local_var_1 = 0;

-- &locals / &globals / &statics at function entry --
$1 = (int *) 0xffffced4
$2 = (int *) 0xffffced0
$3 = (int *) 0x56559018 <global_var_1>
$4 = (int *) 0x5655901c <global_var_2>
$5 = (int *) 0x56559020 <global_uninit_var_1>
$6 = (int *) 0x56559024 <global_uninit_var_2>
$7 = (int *) 0x56559028 <static_var_1>
$8 = (int *) 0x5655902c <static_var_2>

(Note: ptr_1/ptr_2 are 0x0 before malloc.)
$9 = (int *) 0x0
```

Fig. 4. Automated run using pwndbg script.

Although exact addresses differ across runs due to ASLR, the memory placement pattern is stable and consistent with the expected segmentation model [3].

VI. Appendix

A. Source Code: address_layout.c

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

int global_var_1 = 0;
int global_var_2 = 0;

int global_uninit_var_1;
int global_uninit_var_2;

int main()
{
  int local_var_1 = 0;
  int local_var_2 = 0;

  static int static_var_1 = 0;
  static int static_var_2 = 0;

  int *ptr_1 = malloc(100);
  int *ptr_2 = malloc(100);

  printf("Local var 1 address: %p\n", &local_var_1);
  printf("Local var 2 address: %p\n", &local_var_2);

  printf("Heap var 1 address:%p\n", ptr_1);
  printf("Heap var 2 address:%p\n", ptr_2);

  printf("Global (uninit) var 1 address: %p\n", &
      global_uninit_var_1);
  printf("Global (uninit) var 2 address: %p\n", &
      global_uninit_var_2);

  printf("Static Local var 1 address: %p\n", &
      static_var_1);
  printf("Static Local var 2 address: %p\n", &
      static_var_2);

  printf("Global var 1 address: %p\n", &global_var_1
      );
  printf("Global var 2 address: %p\n", &global_var_2
      );

  return 0;
}
```

B. GDB Script: hw4_pwndbg.gdb

```
# ----- Settings & logging -----
set pagination off
set confirm off
set disassembly-flavor intel
set logging file hw4_log.txt
set logging overwrite on
set logging on

# ----- Target -----
file ./address_layout

echo \n===== RUN 1: program output (self-printed
    addresses) =====\n
run
echo \n(Above: addresses printed by the program
    itself.)\n

# ----- Restart for debugger-side inspection -----
echo \n===== RUN 2: debugger inspection at main
    =====\n
start       # temp-break at start of main

echo \n-- &locals / &globals / &statics at function
    entry --\n
print &local_var_1
print &local_var_2
print &global_var_1
print &global_var_2
print &global_uninit_var_1
print &global_uninit_var_2
print &static_var_1
print &static_var_2

echo \n(Note: ptr_1/ptr_2 are 0x0 before malloc.)\n
print ptr_1
print ptr_2

# ----- Capture ptr_1 after malloc -----
# Find the line number for "ptr_1 = malloc(100);"
    first:
#   (gdb) list
#   (gdb) list 1,50
# Replace <L1> and <L2> below with those actual line
     numbers.
# Break on the lines *after* the assignments so the
    pointers are set.

echo \n-- break after ptr_1 assignment --\n

# CHANGE THIS:
next
next
next

print ptr_1

echo \n-- break after ptr_2 assignment --\n
next
next
print ptr_2

echo \n-- disassembly of main() --\n
disassemble /s main

echo \n-- process memory map (pwndbg vmmap) --\n
vmmap
quit
```

C. Log Output: hw4_log.txt

```

===== RUN 1: program output (self-printed addresses)
     =====
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
    gnu/libthread_db.so.1".
[Inferior 1 (process 21883) exited normally]

(Above: addresses printed by the program itself.)

===== RUN 2: debugger inspection at main =====
Temporary breakpoint 1 at 0x565561ba: file
    address_layout.c, line 12.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
    gnu/libthread_db.so.1".

Temporary breakpoint 1, main () at address_layout.c
    :12
12        int local_var_1 = 0;

-- &locals / &globals / &statics at function entry
    --
$1 = (int *) 0xffffced4
$2 = (int *) 0xffffced0
$3 = (int *) 0x56559018 <global_var_1>
$4 = (int *) 0x5655901c <global_var_2>
$5 = (int *) 0x56559020 <global_uninit_var_1>
$6 = (int *) 0x56559024 <global_uninit_var_2>
$7 = (int *) 0x56559028 <static_var_1>
$8 = (int *) 0x5655902c <static_var_2>

(Note: ptr_1/ptr_2 are 0x0 before malloc.)
```

```
$9 = (int *) 0x0
$10 = (int *) 0x0

-- break after ptr_1 assignment --
13        int local_var_2 = 0;
18        int *ptr_1 = malloc(100);
19        int *ptr_2 = malloc(100);
$11 = (int *) 0x5655a1a0

-- break after ptr_2 assignment --
21        printf("Local var 1 address: %p\n", &
    local_var_1);
22        printf("Local var 2 address: %p\n", &
    local_var_2);
$12 = (int *) 0x5655a210

-- disassembly of main() --
Dump of assembler code for function main:
address_layout.c:
11      {
   0x5655619d <+0>:     lea    ecx,[esp+0x4]
   0x565561a1 <+4>:     and    esp,0xfffffff0
   0x565561a4 <+7>:     push   DWORD PTR [ecx-0x4]
   0x565561a7 <+10>:    push   ebp
   0x565561a8 <+11>:    mov    ebp,esp
   0x565561aa <+13>:    push   ebx
   0x565561ab <+14>:    push   ecx
   0x565561ac <+15>:    sub    esp,0x10
   0x565561af <+18>:    call   0x565560a0 <__x86.
       get_pc_thunk.bx>
   0x565561b4 <+23>:    add    ebx,0x2e40

12        int local_var_1 = 0;
   0x565561ba <+29>:    mov    DWORD PTR [ebp-0x14
       ],0x0

13        int local_var_2 = 0;
   0x565561c1 <+36>:    mov    DWORD PTR [ebp-0x18
       ],0x0

14
15        static int static_var_1 = 0;
16        static int static_var_2 = 0;
17
18        int *ptr_1 = malloc(100);
   0x565561c8 <+43>:    sub    esp,0xc
   0x565561cb <+46>:    push   0x64
   0x565561cd <+48>:    call   0x56556050 <
       malloc@plt>
   0x565561d2 <+53>:    add    esp,0x10
   0x565561d5 <+56>:    mov    DWORD PTR [ebp-0xc],
       eax

19        int *ptr_2 = malloc(100);
   0x565561d8 <+59>:    sub    esp,0xc
   0x565561db <+62>:    push   0x64
   0x565561dd <+64>:    call   0x56556050 <
       malloc@plt>
   0x565561e2 <+69>:    add    esp,0x10
   0x565561e5 <+72>:    mov    DWORD PTR [ebp-0x10],
       eax

20
21        printf("Local var 1 address: %p\n", &
    local_var_1);
   0x565561e8 <+75>:    sub    esp,0x8
   0x565561eb <+78>:    lea    eax,[ebp-0x14]
   0x565561ee <+81>:    push   eax
   0x565561ef <+82>:    lea    eax,[ebx-0x1fec]
   0x565561f5 <+88>:    push   eax
   0x565561f6 <+89>:    call   0x56556040 <
       printf@plt>
   0x565561fb <+94>:    add    esp,0x10

22        printf("Local var 2 address: %p\n", &
    local_var_2);
=> 0x565561fe <+97>:    sub    esp,0x8
   0x56556201 <+100>:   lea    eax,[ebp-0x18]
   0x56556204 <+103>:   push   eax
   0x56556205 <+104>:   lea    eax,[ebx-0x1fd3]
   0x5655620b <+110>:   push   eax
   0x5655620c <+111>:   call   0x56556040 <
       printf@plt>
   0x56556211 <+116>:   add    esp,0x10

23
24        printf("Heap var 1 address:%p\n", ptr_1);
   0x56556214 <+119>:   sub    esp,0x8
   0x56556217 <+122>:   push   DWORD PTR [ebp-0xc]
   0x5655621a <+125>:   lea    eax,[ebx-0x1fba]
   0x56556220 <+131>:   push   eax
   0x56556221 <+132>:   call   0x56556040 <
       printf@plt>
   0x56556226 <+137>:   add    esp,0x10

25        printf("Heap var 2 address:%p\n", ptr_2);
   0x56556229 <+140>:   sub    esp,0x8
   0x5655622c <+143>:   push   DWORD PTR [ebp-0x10]
   0x5655622f <+146>:   lea    eax,[ebx-0x1fa3]
   0x56556235 <+152>:   push   eax
   0x56556236 <+153>:   call   0x56556040 <
       printf@plt>
   0x5655623b <+158>:   add    esp,0x10

26
27        printf("Global (uninit) var 1 address: %p\
    n", &global_uninit_var_1);
   0x5655623e <+161>:   sub    esp,0x8
   0x56556241 <+164>:   lea    eax,[ebx+0x2c]
   0x56556247 <+170>:   push   eax
   0x56556248 <+171>:   lea    eax,[ebx-0x1f8c]
   0x5655624e <+177>:   push   eax
   0x5655624f <+178>:   call   0x56556040 <
       printf@plt>
   0x56556254 <+183>:   add    esp,0x10

28        printf("Global (uninit) var 2 address: %p\
    n", &global_uninit_var_2);
   0x56556257 <+186>:   sub    esp,0x8
   0x5655625a <+189>:   lea    eax,[ebx+0x30]
   0x56556260 <+195>:   push   eax
   0x56556261 <+196>:   lea    eax,[ebx-0x1f68]
   0x56556267 <+202>:   push   eax
   0x56556268 <+203>:   call   0x56556040 <
       printf@plt>
   0x5655626d <+208>:   add    esp,0x10

29
30        printf("Static Local var 1 address: %p\n",
     &static_var_1);
   0x56556270 <+211>:   sub    esp,0x8
   0x56556273 <+214>:   lea    eax,[ebx+0x34]
   0x56556279 <+220>:   push   eax
   0x5655627a <+221>:   lea    eax,[ebx-0x1f44]
   0x56556280 <+227>:   push   eax
   0x56556281 <+228>:   call   0x56556040 <
       printf@plt>
   0x56556286 <+233>:   add    esp,0x10

31        printf("Static Local var 2 address: %p\n",
     &static_var_2);
   0x56556289 <+236>:   sub    esp,0x8
   0x5655628c <+239>:   lea    eax,[ebx+0x38]
   0x56556292 <+245>:   push   eax
   0x56556293 <+246>:   lea    eax,[ebx-0x1f24]
   0x56556299 <+252>:   push   eax
   0x5655629a <+253>:   call   0x56556040 <
       printf@plt>
   0x5655629f <+258>:   add    esp,0x10

32
33        printf("Global var 1 address: %p\n", &
    global_var_1);
   0x565562a2 <+261>:   sub    esp,0x8
```

```
   0x565562a5 <+264>:   lea    eax,[ebx+0x24]
   0x565562ab <+270>:   push   eax
   0x565562ac <+271>:   lea    eax,[ebx-0x1f04]
   0x565562b2 <+277>:   push   eax
   0x565562b3 <+278>:   call   0x56556040 <
       printf@plt>
   0x565562b8 <+283>:   add    esp,0x10

34        printf("Global var 2 address: %p\n", &
    global_var_2);
   0x565562bb <+286>:   sub    esp,0x8
   0x565562be <+289>:   lea    eax,[ebx+0x28]
   0x565562c4 <+295>:   push   eax
   0x565562c5 <+296>:   lea    eax,[ebx-0x1eea]
   0x565562cb <+302>:   push   eax
   0x565562cc <+303>:   call   0x56556040 <
       printf@plt>
   0x565562d1 <+308>:   add    esp,0x10

35
36        return 0;
   0x565562d4 <+311>:   mov    eax,0x0

37      }
   0x565562d9 <+316>:   lea    esp,[ebp-0x8]
   0x565562dc <+319>:   pop    ecx
   0x565562dd <+320>:   pop    ebx
   0x565562de <+321>:   pop    ebp
   0x565562df <+322>:   lea    esp,[ecx-0x4]
   0x565562e2 <+325>:   ret
End of assembler dump.

-- process memory map (pwndbg vmmap) --
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
     Start        End Perm     Size  Offset File (
         set vmmap-prefer-relpaths on)
0x56555000 0x56556000 r--p     1000       0
    address_layout
0x56556000 0x56557000 r-xp     1000    1000
    address_layout
0x56557000 0x56558000 r--p     1000    2000
    address_layout
0x56558000 0x56559000 r--p     1000    2000
    address_layout
0x56559000 0x5655a000 rw-p     1000    3000
    address_layout
0x5655a000 0x5657c000 rw-p    22000       0 [heap]
0xf7d68000 0xf7d8b000 r--p    23000       0 /usr/lib
    /i386-linux-gnu/libc.so.6
0xf7d8b000 0xf7f14000 r-xp   189000   23000 /usr/lib
    /i386-linux-gnu/libc.so.6
0xf7f14000 0xf7f99000 r--p    85000  1ac000 /usr/lib
    /i386-linux-gnu/libc.so.6
0xf7f99000 0xf7f9b000 r--p     2000  231000 /usr/lib
    /i386-linux-gnu/libc.so.6
0xf7f9b000 0xf7f9c000 rw-p     1000  233000 /usr/lib
    /i386-linux-gnu/libc.so.6
0xf7f9c000 0xf7fa6000 rw-p     a000       0 [
    anon_f7f9c]
0xf7fbf000 0xf7fc1000 rw-p     2000       0 [
    anon_f7fbf]
0xf7fc1000 0xf7fc5000 r--p     4000       0 [vvar]
0xf7fc5000 0xf7fc7000 r-xp     2000       0 [vdso]
0xf7fc7000 0xf7fc8000 r--p     1000       0 /usr/lib
    /i386-linux-gnu/ld-linux.so.2
0xf7fc8000 0xf7fec000 r-xp    24000    1000 /usr/lib
    /i386-linux-gnu/ld-linux.so.2
0xf7fec000 0xf7ffb000 r--p     f000   25000 /usr/lib
    /i386-linux-gnu/ld-linux.so.2
0xf7ffb000 0xf7ffd000 r--p     2000   33000 /usr/lib
    /i386-linux-gnu/ld-linux.so.2
0xf7ffd000 0xf7ffe000 rw-p     1000   35000 /usr/lib
    /i386-linux-gnu/ld-linux.so.2
0xfffdd000 0xffffe000 rw-p    21000       0 [stack]
```

REFERENCES

[1] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB*, Free Software Foundation, 2002.

[2] "pwndbg: A GDB plugin for exploit development," GitHub repository. [Online]. Available: https://github.com/pwndbg/pwndbg

[3] A. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed., Pearson, 2015.