GDB Labs and You

Black Lodge Research Edition

@Crypto_Monkey

https://github.com/cryptomonkey

Contents

Introduction – Start Here

Assumption 1: You are running Linux

Assumption 2: You have gdb installed with python compiled in

    ldd `which gdb` and look for the python library

Assumption 3: You know the basics of gdb

    https://web.stanford.edu/class/cs107/gdb_refcard.pdf

    http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Assumption 4: You have the rest of the github repository this came from

Assumption 5: You will ask questions if anything is unclear or I cease making sense

Extending GDB Labs

Lab 01: Run GDB

Check that your system is ready for the labs

1. From a terminal run “gdb”
2. Check for python by entering “py” or “python”
3. Run the following

   (gdb) py

   >import sys

   >print(sys.version)

   >end

Note what version of python you will be using

Lab 02: Basic Python In GDB:

Executing basic python

```
python
print(“Running Python!”)
end
```

Loading in a library

```
py
import os
print(“my pid is %d” % os.getpid())
end
```

Lab 03: Loading An External Script:

Create a file containing the commands you want to execute -

```
root@kali:/mnt/hgfs/code/gdb work# cat my_pid.py
import os
print("My pid is %d" %os.getpid())
```

In gdb run your script via the source command

```
(gdb) source ./my_pid.py
My pid is 3521
(gdb)
```

Note this is the pid of gdb, the program that is running, called the inferior, has a different pid

EXAMPLE

```
root@kali:/mnt/hgfs/code/gdb work# gdb stacker
GNU gdb (GDB) 8.0.50.20170511-git
Copyright (C) 2017 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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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 stacker...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x775
(gdb) r
Starting program: /mnt/hgfs/code/gdb work/stacker

Breakpoint 1, 0x0000555555554775 in main ()
(gdb) source ./my_pid.py
```

```
My pid is 3537
(gdb) call getpid()
$1 = 3539
```

Why do these two differ (remember your pids will be different than the ones listed)?

Lab 04: Python Exercises :

Prepare an executable

```
root@kali:/mnt/hgfs/code/gdb work# cat stacker.c
#include <stdio.h>

void void_func(void) {
   unsigned int blah = 4059231220;  /* 0xf1f2f3f4 */
   char* manifesto="Stacks are for wimps";

   blah++;
   printf("%s \n", manifesto);
   printf("but I'm at %p \n", *void_func);
}

int function_127(int first, int second, char* bluster){
   int internal_int = 16909060; /* 0x01020304 */
   char* internal_string = "0x444F524B21";

   printf("%s: %i plus %i iz %i \n", bluster, first, second, first+second);
   printf("but my int is %i and the string is at %p \n", internal_int, &internal_string);

   void_func();
}
```

```
int main(int argc, char *argv[]) {
   int main_int = 1903326068;  /* 0x71727374 */

   void_func();
   main_int = function_127(65535, 1, "Yo Dog" );

}
```

```
root@kali:/mnt/hgfs/code/gdb work# gcc -o stacker -ggdb stacker.c
```

Explore using python in gdb by starting our sample process, it is not stripped or obfuscated yet so it is wide open to us. Also, since we are only executing one command and our session is persistent let's switch to entering it all on one line. We will the setup all via python

```
root@kali:/mnt/hgfs/code/gdb work# gdb -quiet stacker
Reading symbols from stacker...done.
(gdb) python gdb.Breakpoint('main')
Breakpoint 1 at 0x780: file stacker.c, line 23.
(gdb) python gdb.execute('run')

Breakpoint 1, main (argc=1, argv=0x7fffffffe2a8) at stacker.c:23
23      int main_int = 1903326068;  /* 0x71727374 */
```

Is the python instance persistent between python sessions? Does it remember library loads?

```
(gdb) python
>import os
>end
(gdb) python
>print(os.getpid())
>end
3712
```

Yes, does it remember variables?

```
(gdb) python
```

```
>test = "blah"
>end
(gdb) python
>print(test)
>end
blah
(gdb)
```

Since we left stacker running and hit the main breakpoint is the variable main_int defined? It should be since we are in main so how can we access it via python?

Without python –

```
 (gdb) print main_int
$4 = 0
```

With python -

```
(gdb) python test = gdb.parse_and_eval("main_int")
(gdb) py print(test)
```

OK something is wrong it clearly says in stacker.c that

```
int main(int argc, char *argv[]) {
   int main_int = 1903326068;  /* 0x71727374 */
```

Ooops, our breakpoint (b main) stopped us before the assignment happened, we will try that again after moving forward one source line

```
(gdb) n
25      void_func();
(gdb) print main_int
$5 = 1903326068
(gdb) python test = gdb.parse_and_eval("main_int")
(gdb) py print(test)
1903326068
```

CAUTION: What if we move forward in the program and the value of main_int changes, does the value of the python variable "test" change?   NOOOOOOOO! You must update it via

python test = gdb.parse_and_eval("main_int")

Other things to try

(gdb) python print (gdb.breakpoints())

(<gdb.Breakpoint object at 0x7f26a95648a0>,)

(gdb) python print (gdb.breakpoints()[0].location)

main

(gdb) python print (gdb.breakpoints()[0].enabled)

True

(gdb) python print (gdb.breakpoints()[0].delete())

None

(gdb) python print (gdb.breakpoints()[0].location)

Traceback (most recent call last):

File "<string>", line 1, in <module>

IndexError: tuple index out of range

Error while executing Python code.

Ooops again, there is no breakpoint zero anymore, we deleted it

Lab 05: gdb-dashboard :

If you have installed gdb-dashboard (https://github.com/cyrus-and/gdb-dashboard) you can try this lab.

Using the executable from Lab 4 we will make the enhanced display show up in a second terminal while we work in an uncluttered terminal

1. Open two terminal windows and determine their tty

root@kali:~# tty

/dev/pts/1

2. Start running gdb in the terminal you want to use for input (you will use the other one for the display).

root@kali:~# cd /mnt/hgfs/code/gdb\ work/

root@kali:/mnt/hgfs/code/gdb work# gdb -silent stacker

Reading symbols from stacker...done.

3. If it is not set to autoload, load gdb-dashboard into your gdb session and set the breakpoint

(gdb) source ~/.gdbinit_dash

>>> b main

Breakpoint 1 at 0x780: file stacker.c, line 23.

4. Transfer the enhanced display to the second terminal by setting the output to the value returned by the tty command in that terminal

>>> dashboard -output /dev/pts/0

You can also transfer parts of the display to individual terminals but for this lab we will keep it simple

5. Step through the code via ‘n’ or ‘si’

Does the display update on each step? Can you think of any advantages of such a setup?

NOTE – Below Here We Are Using pwndbg Because It Has The Most Commands And We Are Lazy

Lab 06: What Address Belongs Where?

If you have installed pwndbg (https://github.com/pwndbg/pwndbg) you can try this lab.

Using the executable from Lab 4 list the memory map to determine how the displays know the use of each part of memory.

1. Start the process we used in Lab 4

root@kali:/mnt/hgfs/code/gdb work# gdb -silent ./stacker

Reading symbols from ./stacker...done.

2. Insert a breakpoint for main

(gdb) b main

Breakpoint 1 at 0x780: file stacker.c, line 23.

3. Load in pwndbg and run the process

(gdb) source ~/.gdbinit_pwndbg

Loaded 108 commands. Type pwndbg [filter] for a list.

pwndbg> r

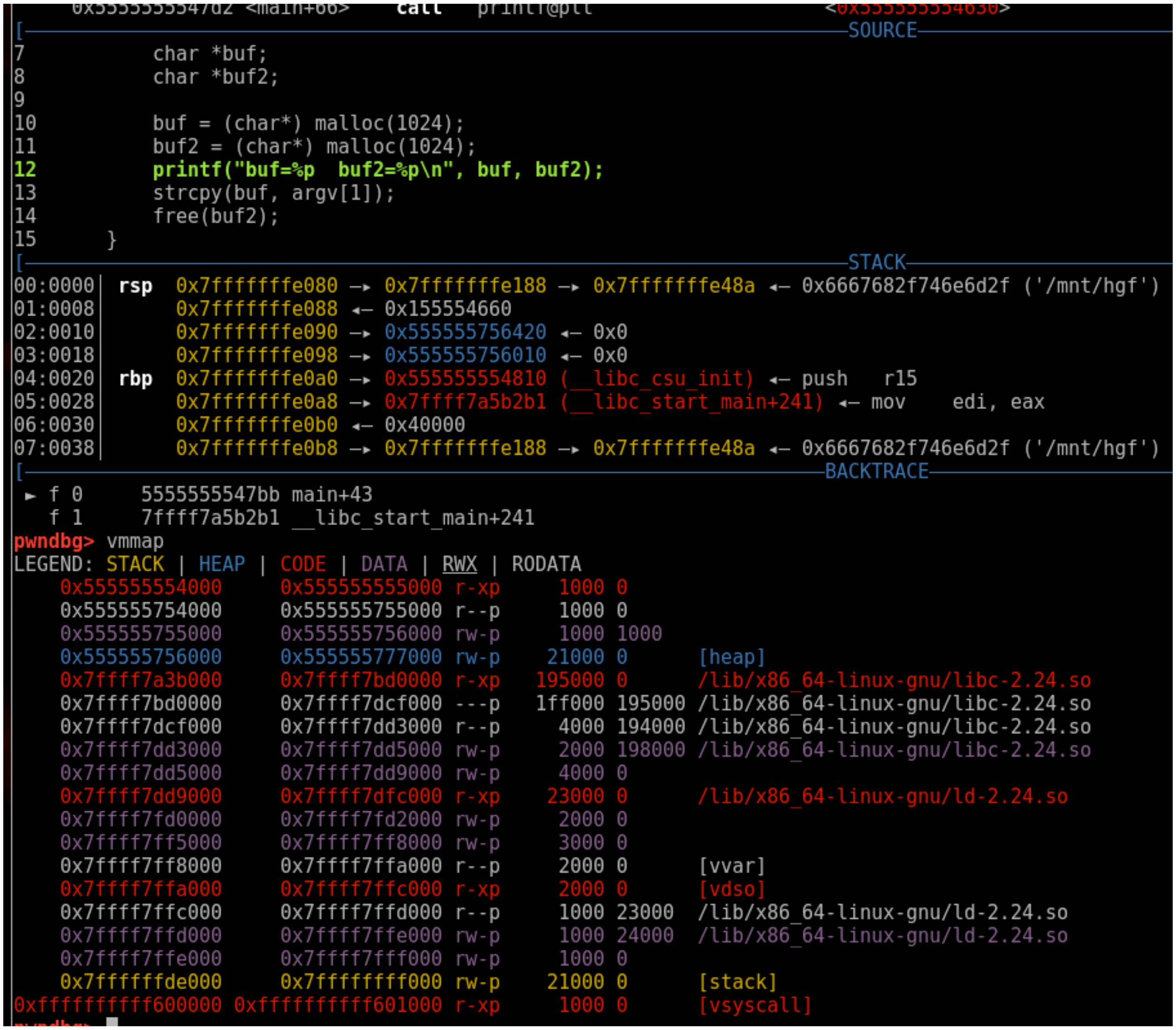
4. Run the vmmap command to dump the virtual memory address mappings

pwndbg> vmmap

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x555555554000     0x555555555000 r-xp     1000 0
    0x555555754000     0x555555755000 r--p     1000 0
    0x555555755000     0x555555756000 rw-p     1000 1000
    0x7ffff7a3b000     0x7ffff7bd0000 r-xp   195000 0      /lib/x86_64-linux-gnu/libc-2.24.so
    0x7ffff7bd0000     0x7ffff7dcf000 ---p   1ff000 195000 /lib/x86_64-linux-gnu/libc-2.24.so
    0x7ffff7dcf000     0x7ffff7dd3000 r--p     4000 194000 /lib/x86_64-linux-gnu/libc-2.24.so
    0x7ffff7dd3000     0x7ffff7dd5000 rw-p     2000 198000 /lib/x86_64-linux-gnu/libc-2.24.so
    0x7ffff7dd5000     0x7ffff7dd9000 rw-p     4000 0
    0x7ffff7dd9000     0x7ffff7dfc000 r-xp    23000 0      /lib/x86_64-linux-gnu/ld-2.24.so
    0x7ffff7fd0000     0x7ffff7fd2000 rw-p     2000 0
    0x7ffff7ff5000     0x7ffff7ff8000 rw-p     3000 0
    0x7ffff7ff8000     0x7ffff7ffa000 r--p     2000 0      [vvar]
    0x7ffff7ffa000     0x7ffff7ffc000 r-xp     2000 0      [vdso]
    0x7ffff7ffc000     0x7ffff7ffd000 r--p     1000 23000  /lib/x86_64-linux-gnu/ld-2.24.so
    0x7ffff7ffd000     0x7ffff7ffe000 rw-p     1000 24000  /lib/x86_64-linux-gnu/ld-2.24.so
    0x7ffff7ffe000     0x7ffff7fff000 rw-p     1000 0
    0x7ffffffde000     0x7ffffffff000 rw-p    21000 0      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp     1000 0      [vsyscall]
```

If I put shellcode on the stack will it be executable?

Has this process dynamically allocated any memory? Compare it to the picture below



Is this process statically linked?

Can a ROP gadget from libc execute?

- If you say yes then why are three of the four memory sections not marked executable?

Lab 07: What Is The Inferior pid Using pwndbg?:

If you have installed pwndbg ([https://github.com/pwndbg/pwndbg)](https://github.com/pwndbg/pwndbg) you can try this lab.

Using the executable from Lab 4 list the memory map to determine how the displays know the use of each part of memory.

1. Start the process we used in Lab 4

root@kali:/mnt/hgfs/code/gdb work# gdb -silent ./stacker

Reading symbols from ./stacker...done.

2. Insert a breakpoint for main

(gdb) b main

Breakpoint 1 at 0x780: file stacker.c, line 23.

3. Load in pwndbg and run the process

(gdb) source ~/.gdbinit_pwndbg

Loaded 108 commands. Type pwndbg [filter] for a list.

pwndbg> r

4. To get information about the running inferior process you can use the procinfo command

What is the pid of gdb? (source my_pid.py from Lab 3 to fetch it via python)

- Did procinfo also give us this information?
- If we were in a fork of the process running in gdb would this still be true? (to be pedantic assume that follow-fork-mode is set to child and we are in the child after the fork)

Has the process opened any files?

- What are the three file descriptors and why are they pointed at a terminal?
- What file does this process have open?

```
pwndbg> procinfo
exe       '/mnt/hgfs/code/gdb work/open_file'
pid       3481
tid       3481
selinux
ppid      3478
uid       [0, 0, 0, 0]
gid       [0, 0, 0, 0]
groups    [0]
fd[0]     /dev/pts/0
fd[1]     /dev/pts/0
fd[2]     /dev/pts/0
fd[3]     /tmp/test.txt
pwndbg>
```

- Could this procinfo been executed after a breakpoint on main()?

Lab 08: Goes Here