Weaponizing GDB

Or How To Get The Most Out Of This

"*Fine Piece of Software*"

Matt DuHarte

@Crypto_Monkey

https://github.com/CryptoMonkey

DISCLAIMER

All Standard disclaimers apply. This talk does not represent the views of my employers, associates, colleagues, or my dog. The information is provided to foster discussion of only the most ethical use of this information.

Agenda

- Motivation
- Base GDB
- Levels of Enhancement
  - Does it use additional programs?
- Suggested Uses

Why This? Why Now?

- Are there better debuggers than GDB?
  - GDB is everywhere, free and lots of help is available
- GDB is powerful enough but it is not that easy to learn IMO
- But <enter l33t hacker> says I should learn it the hard way
  - Suffering doesn't bring enlightenment, it just drives good beginners away and feeds egos
- Accomplishing goals are more important filling our head with obscure knowledge

I'm The Wrong Audience For GDB

I do write code and debug it but

- I spend more time looking at malware, wargames and CTF files
  - No source code, stripped binaries
  - "set print pretty on" and named breakpoints are no good without debugging symbols and info
- I often do not know what they do (that is what I'm trying to figure out)
- At first I have no idea how they were built or what technologies they use

GDB History

"GDB is designed to be a symbolic debugger for programs written in compiled imperative languages such as C, C++, Ada, and Fortran."

- The Architecture of Open Source Applications – Volume II (http://aosabook.org/en/gdb.html)

- First Strike – it expects to run in the symbolic world!
- The symbolic side of the code is of little use in stripped binaries, we have minimal symbols, no line numbers, etc…
- This can result in detectible 'slow stepping' binaries even when executing at full speed

GDB History

- It is not designed for hostile code
- The reliance on ptrace() can be problem
  - Second Strike – it is easy to lock up ptrace() to prevent GDB from working
  - We need to fix that manually to get this to work

GDB History

- In Internet terms this code is ancient
  - Written in 1985!
  - Lots of updates, not always following an easy to see design
    - Needed a lot of abstraction to work on so may platforms
  - There is less of an issue in terms of user interface
    - Some people like text windows
      - Most of what we will discuss likes a big text window
    - Several of the 'better' UIs actually make thing worse (I'm talking to you TUI mode!)

GDB Terms

- Inferior – the program you are debugging
- .gdbinit – a configuration file read by gdb at start up
  - There are multiple layers, first the system one in /etc
  - then your local one in ~ (a.k.a your home directory)

Deficiencies

- When you are a beginner you often want more information on the screen in a clean coherent manner
- Commands are a tad cryptic
- Was not scriptable in a modern language
- Etc…

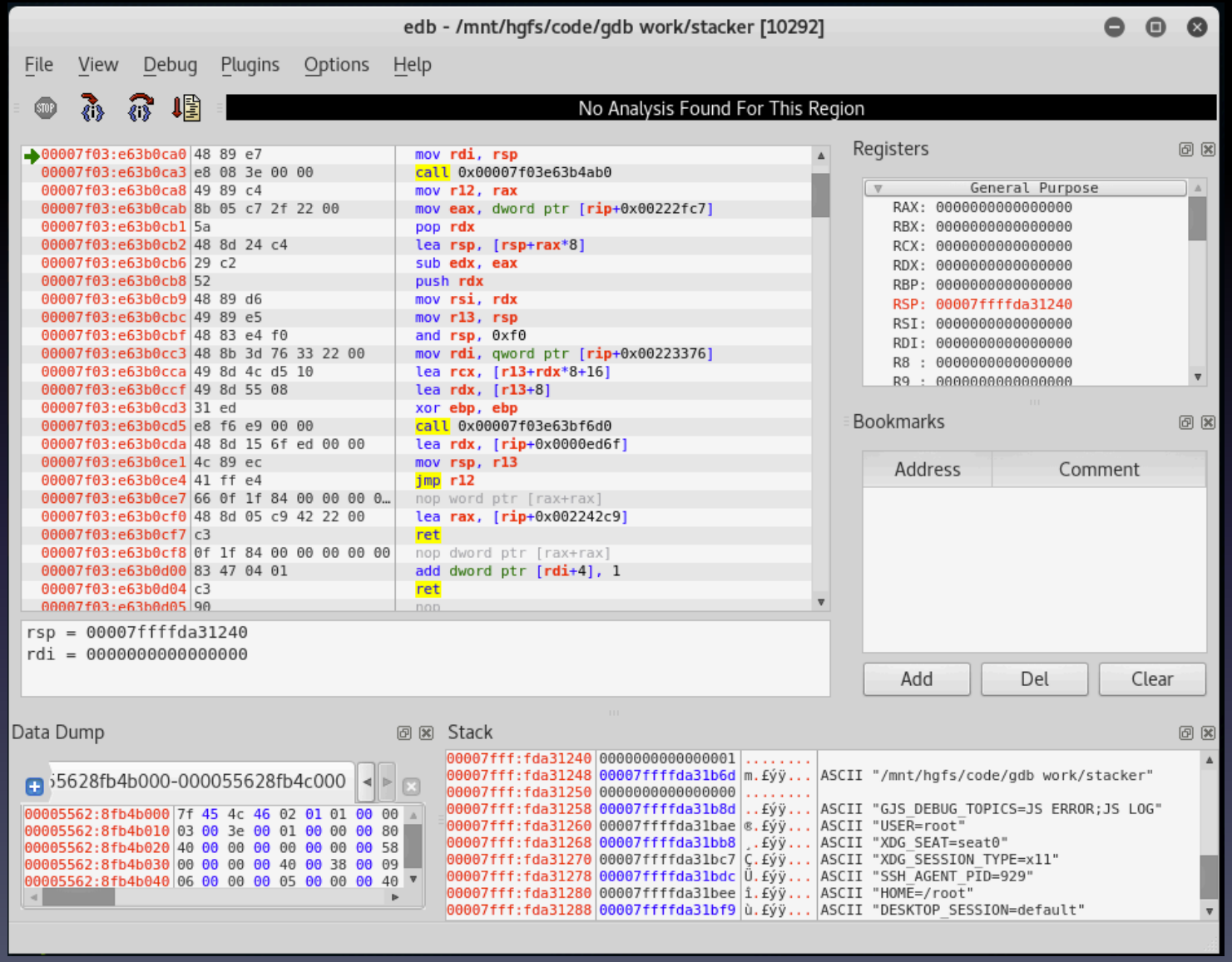
Remedies

- Put more info on the screen
- Give us better commands that get us to important info faster
  - As we look at code ask yourself why the author added the command to gdb

GDB Enhanced Viewing

TUI Mode – More Information On The Screen

Compare That To Available Info In edb



TUI Mode commands

- You can start gdb in TUI (Text UI) mode via the –tui command line switch, "tui enable/disable" or via "ctrl-x ctrl-a" once started
- You can change the layouts via the 'layout' command
  - next, prev, src, asm, regs, cmd
  - Or via ctrl-x 1,2

https://sourceware.org/gdb/onlinedocs/gdb/TUI.html#TUI

TUI Mode Issues

The display messes up all the time

TUI Mode Issues

- The screen seems to get wonky pretty regularly
  - Command "refresh", ctrl-L will refresh the screen or pop out and in tui mode clears it
- You can only turn on sets of predefined displays
  - Little ability to tweak the display
  - You have to use commands to switch between display portions (they scroll independently)
    - So up arrow for commands only works in the commands window
      - Use ctrl-n and ctrl-p for next and previous

There are other gdb veiwers

- cgdb
- DDD
- Insight (if you can get it to compile)

GDB Init and Scripting

- When it starts gdb will look for a .gdbinit file in your home directory
- You can set up all sorts of useful and sanity saving features like

```
set dissassembly-flavor intel
```

.gdbinit basics

- It is a series of commands that execute on each start of gdb
- If you have commands you run every time you can put them there
- You can define custom colors, shortcuts, basic if/else constructs

shortcuts != commands

Python Extension

- As of version 7 of gdb python can be used to extend gdb
  - Python versions (2 or 3) vary by distribution

```
(gdb) py
>import sys
>print (sys.version)
>end
2.7.13 (default, Jan 19 2017, 14:48:08)
[GCC 6.3.0 20170118]
(gdb) $$
```

https://sourceware.org/gdb/onlinedocs/gdb/Python.html#Python
https://sourceware.org/gdb/onlinedocs/gdb.pdf.gz

GDB lies

- What version and executable again?

```
(gdb) py
>import sys
>print(sys.version)
>print(sys.executable)
>end
3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609]
/usr/bin/python
(gdb) quit
user@mess:~$ ls -la /usr/bin/python
lrwxrwxrwx 1 root root 9 Dec  9  2015 /usr/bin/python -> python2.7
```

Python Is Compiled In

- Save yourself the headache, go by the library that is linked in

```
 :~$ ldd `which gdb`
linux-vdso.so.1 =>  (0x00007ffc95959000)
libreadline.so.6 => /lib/x86_64-linux-gnu/libreadline.so.6 (0x00007ff63016e000)
libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007ff62ff54000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ff62fd4f000)
libncurses.so.5 => /lib/x86_64-linux-gnu/libncurses.so.5 (0x00007ff62fb2d000)
libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007ff62f904000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007ff62f5fa000)
libpython3.5m.so.1.0 => /usr/lib/x86_64-linux-gnu/libpython3.5m.so.1.0 (0x00007ff62ef73000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007ff62ed56000)
libexpat.so.1 => /lib/x86_64-linux-gnu/libexpat.so.1 (0x00007ff62eb2c000)
liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5 (0x00007ff62e90a000)
libbabeltrace.so.1 => /usr/lib/x86_64-linux-gnu/libbabeltrace.so.1 (0x00007ff62e6fd000)
libbabeltrace-ctf.so.1 => /usr/lib/x86_64-linux-gnu/libbabeltrace-ctf.so.1 (0x00007ff62e4c2000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff62e0f9000)
/lib64/ld-linux-x86-64.so.2 (0x000055a7c4ccd000)
libutil.so.1 => /lib/x86_64-linux-gnu/libutil.so.1 (0x00007ff62def6000)
libglib-2.0.so.0 => /lib/x86_64-linux-gnu/libglib-2.0.so.0 (0x00007ff62dbe4000)
libuuid.so.1 => /lib/x86_64-linux-gnu/libuuid.so.1 (0x00007ff62d9df000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007ff62d76e000)
```

Labs 1, 2, and 3

```
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
(gdb) $
```

Inferior program

Start it running and pause it

pid of gdb

pid of inferior

```
gdb) shell ps
  PID TTY          TIME CMD
 2626 pts/4    00:00:00 bash
 3537 pts/4    00:00:00 gdb
 3539 pts/4    00:00:00 stacker
 3557 pts/4    00:00:00 ps
gdb)
```

Aside: Where it is running?

- "source" causes it to run within gdb
  - So our python script runs in the python library linked to gdb
    - You have access to the standard modules
  - But any other modules we import are the system ones
- "call" runs in the inferior process (not gdb)
- "shell" spans a new shell and runs there and outputs in gdb

Interpreter confusion

this is a really common issue

- Find the nmap module (installed only in python2)

```
user@mess:~$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import nmap
>>> print(nmap.__file__)
/usr/lib/python2.7/dist-packages/nmap/__init__.pyc
>>>
```

```
(gdb) python
>import nmap
>print(nmap.__file__)
>end
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'nmap'
Error while executing Python code.
(gdb) python
>import os
>print(os.__file__)
>end
/usr/lib/python3.5/os.py
(gdb)
```

- Try to import nmap into gdb (python3 compiled in)
- But the standard module is there

Accessing gdb From Python While In gdb

- There is a gdb python module
  - Help is available via “python help(‘gdb’)
  - “python gdb.execute()” runs gdb commands
  - “python gdb.parse_and_eval()” pulls data from the inferior process

Do Lab_4 for a quick demo of

gdb.parse_and_eval()

What Can You Do With Python

- Anything you can do with python
  - Now you can automate gdb with it
  - Now you can use it to parse things in gdb and inferiors like structures
    - Pretty printing!
  - Now you can add commands you write in python

See the links on the "Python Integration Basics" slide for the documentation links

Extending gdb

We want to

- Get more info on the display
- Add new commands
- Call external programs

The great part is that someone probably already did it for us

Three Flavors Of Stuff

When looking for a better gdb experience we find things that work using

1. only traditional pre-python gdb scripting
2. Using the linked python within gdb
3. Use the linked python and call external programs

What Flavor Of Stuff To Use

- What are the limitations of your display system?
  - Can you install a file to "source" into gdb?
  - Are you running via a MI interface?
  - Can you install packages on the display unit?
    - Are there space, permission, or architectural constraints?
  - Are you looking for a single monitor experience?
    - How big is that monitor?

They're All Going To Want To Take Over your .gdbinit!

- The solutions I'm discussing all want to take over your ~/.gdbinit
- I keep mine blank and source in the enhancement I want

```
root@kali:/mnt/hgfs/code/gdb work# ls -la ~/.gdbinit*
-rw-r--r-- 1 root root  54885 May 16 19:38 /root/.gdbinit_dash
-rw-r--r-- 1 root root     29 May 16 16:46 /root/.gdbinit_gef
-rw-r--r-- 1 root root 266633 May 16 16:46 /root/.gdbinit-gef.py
-rw-r--r-- 1 root root 111614 May 16 17:16 /root/.gdbinit_hack
-rw-r--r-- 1 root root     30 May 16 16:58 /root/.gdbinit.local
-rw-r--r-- 1 root root     22 May 18 17:32 /root/.gdbinit_peda
-rw-r--r-- 1 root root     43 Apr 24 11:42 /root/.gdbinit_pwndbg
```

A Better .gdbinit

https://github.com/gdbinit/Gdbinit

```
Public repo for gdbinit

x86/x86_64 and ARM are supported simultaneously since version 8.0.
To make ARM the default CPU set the $ARM var to 1 or use the "arm" command to switch.
```

- Often called the hacker .gdbinit
  - lots of people have put code into it
- A good basic place to start

On Every Step More Info Is Displayed

```
gdb$ r
Starting program: /mnt/hgfs/code/gdb work/stacker
----------------------------------------------------------------------------------------------------------------------------------[regs]
  RAX: 0x0000555555554771  RBX: 0x0000000000000000  RBP: 0x00007FFFFFFFE1C0  RSP: 0x00007FFFFFFFE1A0  o d I t s z a P c
  RDI: 0x0000000000000001  RSI: 0x00007FFFFFFFE2A8  RDX: 0x00007FFFFFFFE2B8  RCX: 0x0000000000000000  RIP: 0x0000555555554780
  R8 : 0x0000555555554820  R9 : 0x00007FFFF7DE8A50  R10: 0x0000000000000000  R11: 0x00007FFFF7B98100  R12: 0x0000555555554580
  R13: 0x00007FFFFFFFE2A0  R14: 0x0000000000000000  R15: 0x0000000000000000
  CS: 0033  DS: 0000  ES: 0000  FS: 0000  GS: 0000  SS: 002B
[0x002B:0x00007FFFFFFFE1A0]-------------------------------------------------------------------------------------------------[stack]
0x00007FFFFFFFE1F0 : 00 00 00 00 00 00 00 00 - 16 65 77 1D FE 57 A1 E8 .........ew..W..
0x00007FFFFFFFE1E0 : 08 C1 B9 F7 01 00 00 00 - 71 47 55 55 55 55 00 00 ........qGUUUU..
0x00007FFFFFFFE1D0 : 00 00 04 00 00 00 00 00 - A8 E2 FF FF FF 7F 00 00 ................
0x00007FFFFFFFE1C0 : B0 47 55 55 55 55 00 00 - B1 B2 A5 F7 FF 7F 00 00 .GUUUU..........
0x00007FFFFFFFE1B0 : A0 E2 FF FF FF 7F 00 00 - 00 00 00 00 00 00 00 00 ................
0x00007FFFFFFFE1A0 : A8 E2 FF FF FF 7F 00 00 - 80 45 55 55 01 00 00 00 .........EUU....
----------------------------------------------------------------------------------------------------------------------------------[code]
=> 0x555555554780 <main+15>:    mov    DWORD PTR [rbp-0x4],0x71727374
   0x555555554787 <main+22>:    call   0x5555555546b0 <void_func>
   0x55555555478c <main+27>:    lea    rdx,[rip+0x120]        # 0x5555555548b3
   0x555555554793 <main+34>:    mov    esi,0x1
   0x555555554798 <main+39>:    mov    edi,0xffff
   0x55555555479d <main+44>:    call   0x555555554701 <function_127>
   0x5555555547a2 <main+49>:    mov    DWORD PTR [rbp-0x4],eax
   0x5555555547a5 <main+52>:    mov    eax,0x0
----------------------------------------------------------------------------------------------------------------------------------

Breakpoint 1, main (argc=0x1, argv=0x7fffffffe2a8) at stacker.c:23
23          int main_int = 1903326068;  /* 0x71727374 */
```

We see separate sections of the display for regs, the stack, assembly, etc.

Registers change color when written to

```
gdb$ r
Starting program: /mnt/hgfs/code/gdb work/stacker
------------------------------------------------------------------------------------------------------------------------[regs]
  RAX: 0x0000555555554771  RBX: 0x0000000000000000  RBP: 0x00007FFFFFFFE1C0  RSP: 0x00007FFFFFFFE1A0  o d I t s z a P c
  RDI: 0x0000000000000001  RSI: 0x00007FFFFFFFE2A8  RDX: 0x00007FFFFFFFE2B8  RCX: 0x0000000000000000  RIP: 0x0000555555554780
  R8 : 0x0000555555554820  R9 : 0x00007FFFF7DE8A50  R10: 0x0000000000000000  R11: 0x00007FFFF7B98100  R12: 0x0000555555554580
  R13: 0x00007FFFFFFFE2A0  R14: 0x0000000000000000  R15: 0x0000000000000000
  CS: 0033  DS: 0000  ES: 0000  FS: 0000  GS: 0000  SS: 002B
[0x002B:0x00007FFFFFFFE1A0]-------------------------------------------------------------------------------------------[stack]
0x00007FFFFFFFE1F0 : 00 00 00 00 00 00 00 00 - 16 65 77 1D FE 57 A1 E8 .........ew..W..
0x00007FFFFFFFE1E0 : 08 C1 B9 F7 01 00 00 00 - 71 47 55 55 55 55 00 00 ........qGUUUU..
0x00007FFFFFFFE1D0 : 00 00 04 00 00 00 00 00 - A8 E2 FF FF FF 7F 00 00 ................
0x00007FFFFFFFE1C0 : B0 47 55 55 55 55 00 00 - B1 B2 A5 F7 FF 7F 00 00 .GUUUU..........
0x00007FFFFFFFE1B0 : A0 E2 FF FF FF 7F 00 00 - 00 00 00 00 00 00 00 00 ................
0x00007FFFFFFFE1A0 : A8 E2 FF FF FF 7F 00 00 - 80 45 55 55 01 00 00 00 .........EUU....
------------------------------------------------------------------------------------------------------------------------[code]
=> 0x555555554780 <main+15>:    mov    DWORD PTR [rbp-0x4],0x71727374
   0x555555554787 <main+22>:    call   0x5555555546b0 <void_func>
   0x55555555478c <main+27>:    lea    rdx,[rip+0x120]        # 0x5555555548b3
   0x555555554793 <main+34>:    mov    esi,0x1
   0x555555554798 <main+39>:    mov    edi,0xffff
   0x55555555479d <main+44>:    call   0x555555554701 <function_127>
   0x5555555547a2 <main+49>:    mov    DWORD PTR [rbp-0x4],eax
   0x5555555547a5 <main+52>:    mov    eax,0x0
------------------------------------------------------------------------------------------------------------------------

Breakpoint 1, main (argc=0x1, argv=0x7fffffffe2a8) at stacker.c:23
23          int main_int = 1903326068;  /* 0x71727374 */
```

But only when written to

```
gdb$ si
-------------------------------------------------------------------------------------------------------------------------------[regs]
  RAX: 0x0000555555554771  RBX: 0x0000000000000000  RBP: 0x00007FFFFFFFE1C0  RSP: 0x00007FFFFFFFE1A0  o d I t s z a P c
  RDI: 0x0000000000000001  RSI: 0x00007FFFFFFFE2A8  RDX: 0x00007FFFFFFFE2B8  RCX: 0x0000000000000000  RIP: 0x0000555555554787
  R8 : 0x0000555555554820  R9 : 0x00007FFFF7DE8A50  R10: 0x0000000000000000  R11: 0x00007FFFF7B98100  R12: 0x0000555555554580
  R13: 0x00007FFFFFFFE2A0  R14: 0x0000000000000000  R15: 0x0000000000000000
  CS: 0033  DS: 0000  ES: 0000  FS: 0000  GS: 0000  SS: 002B
[0x002B:0x00007FFFFFFFE1A0]-------------------------------------------------------------------------------------------------[stack]
0x00007FFFFFFFE1F0 : 00 00 00 00 00 00 00 00 - 16 65 77 1D FE 57 A1 E8 .........ew..W..
0x00007FFFFFFFE1E0 : 08 C1 B9 F7 01 00 00 00 - 71 47 55 55 55 55 00 00 ........qGUUUU..
0x00007FFFFFFFE1D0 : 00 00 04 00 00 00 00 00 - A8 E2 FF FF FF 7F 00 00 ................
0x00007FFFFFFFE1C0 : B0 47 55 55 55 55 00 00 - B1 B2 A5 F7 FF 7F 00 00 .GUUUU..........
0x00007FFFFFFFE1B0 : A0 E2 FF FF FF 7F 00 00 - 00 00 00 00 74 73 72 71 ............tsrq
0x00007FFFFFFFE1A0 : A8 E2 FF FF FF 7F 00 00 - 80 45 55 55 01 00 00 00 .........EUU....
-------------------------------------------------------------------------------------------------------------------------------[code]
=> 0x555555554787 <main+22>:    call   0x5555555546b0 <void_func>
   0x55555555478c <main+27>:    lea    rdx,[rip+0x120]        # 0x5555555548b3
   0x555555554793 <main+34>:    mov    esi,0x1
   0x555555554798 <main+39>:    mov    edi,0xffff
   0x55555555479d <main+44>:    call   0x555555554701 <function_127>
   0x5555555547a2 <main+49>:    mov    DWORD PTR [rbp-0x4],eax
   0x5555555547a5 <main+52>:    mov    eax,0x0
   0x5555555547aa <main+57>:    leave
-------------------------------------------------------------------------------------------------------------------------------
25          void_func();
gdb$ $$
```

- This greatly improves your ability to see what has changed on each step
- Nice when you are learning assembly

What does it do?

- Well this has improved our display
  - With some of the advantages of TUI mode of still being a vanilla terminal not using curses
  - Just like TUI mode you can turn on and off sections of the display
    - Disablestack, disablecpuregisters…
  - It also added a bunch of shortcuts

grep -A1 Syntax ~/.gdbinit_hack

```
Syntax: contextsize-stack NUM
| Set stack dump window size to NUM lines.
--
Syntax: contextsize-data NUM
| Set data dump window size to NUM lines.
--
Syntax: contextsize-code NUM
| Set code window size to NUM lines.
--
Syntax: bpl
| List all breakpoints.
--
Syntax: bp LOCATION
| Set breakpoint.
--
Syntax: bpc LOCATION
| Clear breakpoint.
--
Syntax: bpe NUM
| Enable breakpoint with number NUM.
--
Syntax: bpd NUM
| Disable breakpoint with number NUM.
--
Syntax: bpt LOCATION
| Set a temporary breakpoint.
--
```

Many new shortcuts added but not a lot of new commands

gdb-dashboard

- https://github.com/cyrus-and/gdb-dashboard
- Better displays via python

GDB dashboard requires at least GDB 7.7 compiled with Python 2.7 in order to work properly

gdb-dashboard display

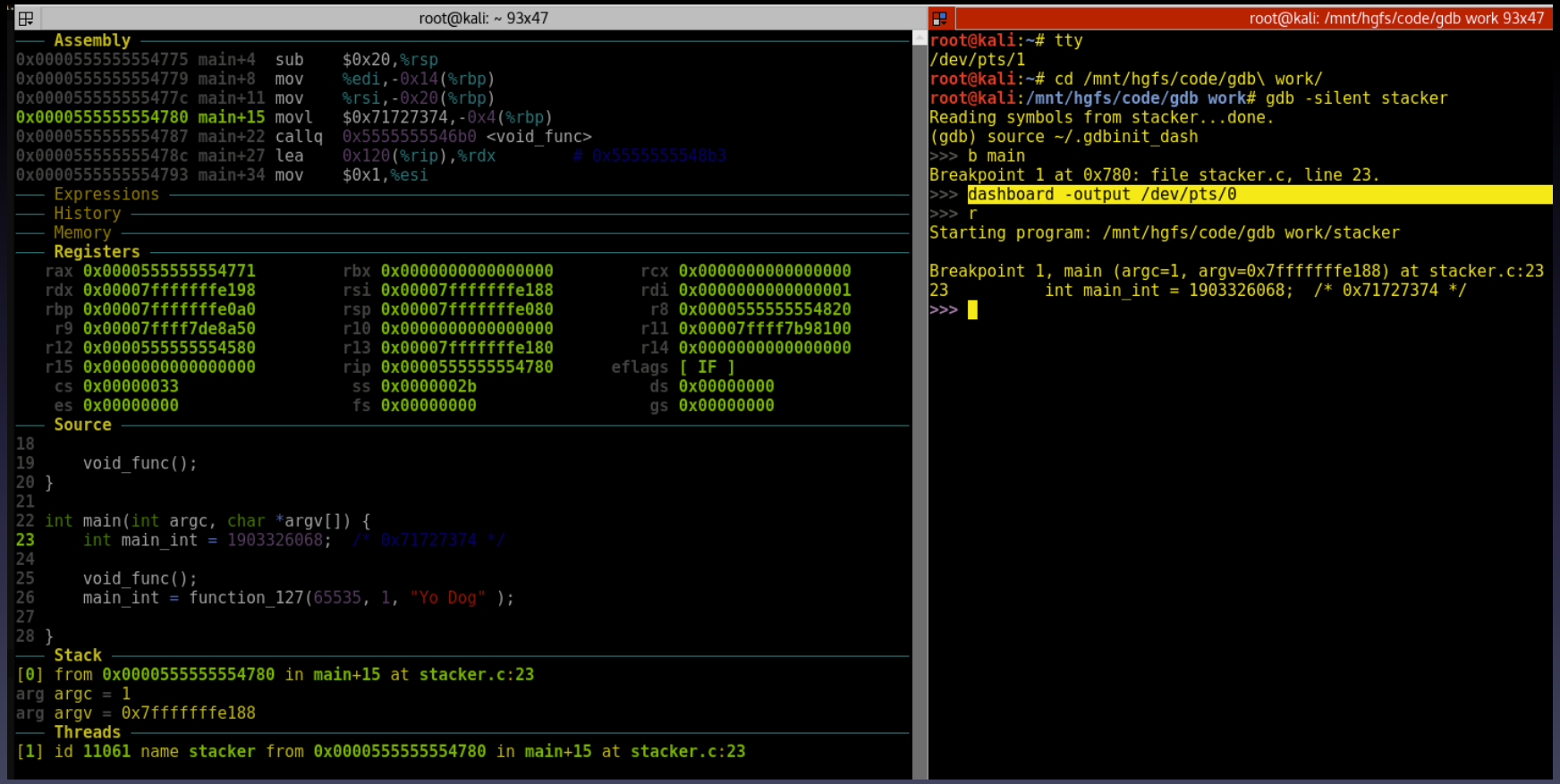
```
── Output/messages ──
── Assembly ──
0x0000555555554775 main+4  sub    $0x20,%rsp
0x0000555555554779 main+8  mov    %edi,-0x14(%rbp)
0x000055555555477c main+11 mov    %rsi,-0x20(%rbp)
0x0000555555554780 main+15 movl   $0x71727374,-0x4(%rbp)
0x0000555555554787 main+22 callq  0x5555555546b0 <void_func>
0x000055555555478c main+27 lea    0x120(%rip),%rdx        # 0x5555555548b3
0x0000555555554793 main+34 mov    $0x1,%esi
── Expressions ──
── History ──
── Memory ──
── Registers ──
   rax 0x0000555555554771         rbx 0x0000000000000000         rcx 0x0000000000000000         rdx 0x00007fffffffe198
   rsi 0x00007fffffffe188         rdi 0x0000000000000001         rbp 0x00007fffffffe0a0         rsp 0x00007fffffffe080
    r8 0x0000555555554820          r9 0x00007ffff7de8a50         r10 0x0000000000000000         r11 0x00007ffff7b98100
   r12 0x0000555555554580         r13 0x00007fffffffe180         r14 0x0000000000000000         r15 0x0000000000000000
   rip 0x0000555555554780      eflags [ IF ]                      cs 0x00000033                  ss 0x0000002b
    ds 0x00000000                  es 0x00000000                  fs 0x00000000                  gs 0x00000000
── Source ──
18
19      void_func();
20  }
21
22  int main(int argc, char *argv[]) {
23      int main_int = 1903326068;  /* 0x71727374 */
24
25      void_func();
26      main_int = function_127(65535, 1, "Yo Dog" );
27
28  }
── Stack ──
[0] from 0x0000555555554780 in main+15 at stacker.c:23
arg argc = 1
arg argv = 0x7fffffffe188
── Threads ──
[1] id 10974 name stacker from 0x0000555555554780 in main+15 at stacker.c:23

Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at stacker.c:23
23          int main_int = 1903326068;  /* 0x71727374 */
>>>
```

we get the value here

but lose registers changing color

But You Can Send Info To A Second Terminal



Each section of the display can be sent to a separate terminal just like voltron…or to a web browser if you are so inclined

Only Display Enhancements

- Both of these projects really only enhance the display of data in gdb
  - One is pure gdb, the other is gdb+built-in python
- Now lets add in some new commands

PEDA

PEDA

- Python Exploit Development Assistance for GDB
  - https://github.com/longld/peda
  - Released at BlackHat 2012 – see http://ropshell.com/peda/ for workshop files
- Enhance the display of gdb: colorize and display disassembly codes, registers, memory information during debugging
- Add commands to support debugging and exploit development

PEDA

- No real output on loading

```
root@kali:/mnt/hgfs/code/gdb_work# gdb -silent ./stacker
Reading symbols from ./stacker...done.
(gdb) b main
Breakpoint 1 at 0x780: file stacker.c, line 23.
(gdb) source ~/.gdbinit_peda
```

- Only python?
  - No, it also has calls out to readelf and other programs

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : Partial
```

clear the screen and display

```
[----------------------------------registers-----------------------------------]
RAX: 0x555555554771 (<main>:    push   rbp)
RBX: 0x0
RCX: 0x0
RDX: 0x7fffffffe198 --> 0x7fffffffe4ae ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:
01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc"...)
RSI: 0x7fffffffe188 --> 0x7fffffffe48e ("/mnt/hgfs/code/gdb work/stacker")
RDI: 0x1
RBP: 0x7fffffffe0a0 --> 0x5555555547b0 (<__libc_csu_init>:      push   r15)
RSP: 0x7fffffffe078 --> 0x55555555478c (<main+27>:      lea    rdx,[rip+0x120]        # 0x5555555548b3)
RIP: 0x5555555546b0 (<void_func>:       push   rbp)
R8 : 0x555555554820 (<__libc_csu_fini>: repz ret)
R9 : 0x7ffff7de8a50 (<_dl_fini>:        push   rbp)
R10: 0x2
R11: 0x1
R12: 0x555555554580 (<_start>:  xor    ebp,ebp)
R13: 0x7fffffffe180 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x5555555546a8 <frame_dummy+40>:     call   rax
   0x5555555546aa <frame_dummy+42>:     pop    rbp
   0x5555555546ab <frame_dummy+43>:     jmp    0x5555555545f0 <register_tm_clones>
=> 0x5555555546b0 <void_func>:  push   rbp
   0x5555555546b1 <void_func+1>:        mov    rbp,rsp
   0x5555555546b4 <void_func+4>:        sub    rsp,0x10
   0x5555555546b8 <void_func+8>:        mov    DWORD PTR [rbp-0x4],0xf1f2f3f4
   0x5555555546bf <void_func+15>:       lea    rax,[rip+0x172]        # 0x555555554838
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe078 --> 0x55555555478c (<main+27>:     lea    rdx,[rip+0x120]        # 0x5555555548b3)
0008| 0x7fffffffe080 --> 0x7fffffffe188 --> 0x7fffffffe48e ("/mnt/hgfs/code/gdb work/stacker")
0016| 0x7fffffffe088 --> 0x155554580
0024| 0x7fffffffe090 --> 0x7fffffffe180 --> 0x1
0032| 0x7fffffffe098 --> 0x7172737400000000 ('')
0040| 0x7fffffffe0a0 --> 0x5555555547b0 (<__libc_csu_init>:     push   r15)
0048| 0x7fffffffe0a8 --> 0x7ffff7a5b2b1 (<__libc_start_main+241>:       mov    edi,eax)
0056| 0x7fffffffe0b0 --> 0x40000
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
void_func () at stacker.c:3
3       void void_func(void) {
gdb-peda$
```

What PEDA offers

- The screen is blocked out into a section for
- Registers, code, and the stack
- There is more to the sections then basic displays

PEDA telescoping

Given:

```
Legend: code, data, rodata, value
```

The stack comes automatically comes alive with info

```
[-------------------------------------stack--------------------------------------]
0000| 0x7fffffffe060 --> 0x555555554838 ("Stacks are for wimps")
0008| 0x7fffffffe068 --> 0xf1f2f3f5555547fd
0016| 0x7fffffffe070 --> 0x7fffffffe0a0 --> 0x5555555547b0 (<__libc_csu_init>:  push   r15)
0024| 0x7fffffffe078 --> 0x55555555478c (<main+27>:     lea    rdx,[rip+0x120]        # 0x5555555548b3)
0032| 0x7fffffffe080 --> 0x7fffffffe188 --> 0x7fffffffe48e ("/mnt/hgfs/code/gdb work/stacker")
0040| 0x7fffffffe088 --> 0x155554580
0048| 0x7fffffffe090 --> 0x7fffffffe180 --> 0x1
0056| 0x7fffffffe098 --> 0x7172737400000000 ('')
```

- At the Stack pointer RSP: 0x7fffffffe060 we see the address is blue
- so it is in the data section
- We also see that it contains the value 0x555555554838
- that value is green so it is in the RODATA
- we see that the string found at that address is

"Stacks are for wimps"

PEDA telescoping

- The same display is also used in the registers section
- The telescope command can apply this effect to any address

```
gdb-peda$ telescope 0x7fffffffe070
0000| 0x7fffffffe070 --> 0x7fffffffe0a0 --> 0x5555555547b0 (<__libc_csu_init>:  push   r15)
0008| 0x7fffffffe078 --> 0x55555555478c (<main+27>:     lea    rdx,[rip+0x120]        # 0x5555555548b3)
```

Here we see the effect where the address we entered pointed to another stack address that pointed to a address in the code

In both cases (the entry points at a code address and the entry points to a stack value that points to a code address telescope is

showing us the name of the function
showing us the instruction that is pointed to
showing us the present value the the instruction is accessing
i.e. [rip+0x120] = 0x5555555548b3

Not Perfection

- While this is not perfect yet, it really makes our lives easier
- This is why, if you can spare the screen real estate, enhancing gdb will save you time and headaches
- You get a more complete picture of the code without jumping through hoops

PEDA Commands

- ‘peda help’ lists them
- We will not discuss them in depth
- The BlackHat walkthrough it a great place to start

Before we leave PEDA

- PEDA has some of the most accessible code for looking at python integration
- You see tons of the python we have discussed

```
gdb.execute('set logging off') # prevent nested call
gdbtarget = self.parse_and_eval(target)
```

PEDA Is Pretty Accessible Python Code

- Self contained
- Limited architecture support
- Good documentation on how to enhance it
  - http://ropshell.com/peda/Linux_Interactive_Exploit_Development_with_GDB_and_PEDA_Slides.pdf starting at slide 38

PEDA Commands

```
(gdb) source ~/.gdbinit_peda
gdb-peda$ help peda
PEDA - Python Exploit Development Assistance for GDB
For latest update, check peda project page: https://github.com/longld/peda/
List of "peda" subcommands, type the subcommand to invoke it:
aslr -- Show/set ASLR setting of GDB
asmsearch -- Search for ASM instructions in memory
assemble -- On the fly assemble and execute instructions using NASM
checksec -- Check for various security options of binary
cmpmem -- Compare content of a memory region with a file
context -- Display various information of current execution context
context_code -- Display nearby disassembly at $PC of current execution context
context_register -- Display register information of current execution context
context_stack -- Display stack of current execution context
crashdump -- Display crashdump info and save to file
deactive -- Bypass a function by ignoring its execution (eg sleep/alarm)
distance -- Calculate distance between two addresses
dumpargs -- Display arguments passed to a function when stopped at a call instruction
dumpmem -- Dump content of a memory region to raw binary file
dumprop -- Dump all ROP gadgets in specific memory range
eflags -- Display/set/clear/toggle value of eflags register
elfheader -- Get headers information from debugged ELF file
elfsymbol -- Get non-debugging symbol information from an ELF file
```

PEDA Commands

```
gennop -- Generate abitrary length NOP sled using given characters
getfile -- Get exec filename of current debugged process
getpid -- Get PID of current debugged process
goto -- Continue execution at an address
help -- Print the usage manual for PEDA commands
hexdump -- Display hex/ascii dump of data in memory
hexprint -- Display hexified of data in memory
jmpcall -- Search for JMP/CALL instructions in memory
loadmem -- Load contents of a raw binary file to memory
lookup -- Search for all addresses/references to addresses which belong to a memory range
nearpc -- Disassemble instructions nearby current PC or given address
nextcall -- Step until next 'call' instruction in specific memory range
nextjmp -- Step until next 'j*' instruction in specific memory range
nxtest -- Perform real NX test to see if it is enabled/supported by OS
patch -- Patch memory start at an address with string/hexstring/int
pattern -- Generate, search, or write a cyclic pattern to memory
pattern_arg -- Set argument list with cyclic pattern
pattern_create -- Generate a cyclic pattern
pattern_env -- Set environment variable with a cyclic pattern
pattern_offset -- Search for offset of a value in cyclic pattern
pattern_patch -- Write a cyclic pattern to memory
pattern_search -- Search a cyclic pattern in registers and memory
payload -- Generate various type of ROP payload using ret2plt
pdisass -- Format output of gdb disassemble command with colors
pltbreak -- Set breakpoint at PLT functions match name regex
procinfo -- Display various info from /proc/pid/
profile -- Simple profiling to count executed instructions in the program
pyhelp -- Wrapper for python built-in help
```

PEDA Commands

```
readelf -- Get headers information from an ELF file
refsearch -- Search for all references to a value in memory ranges
reload -- Reload PEDA sources, keep current options untouch
ropgadget -- Get common ROP gadgets of binary or library
ropsearch -- Search for ROP gadgets in memory
searchmem -- Search for a pattern in memory; support regex search
session -- Save/restore a working gdb session to file as a script
set -- Set various PEDA options and other settings
sgrep -- Search for full strings contain the given pattern
shellcode -- Generate or download common shellcodes.
show -- Show various PEDA options and other settings
skeleton -- Generate python exploit code template
skipi -- Skip execution of next count instructions
snapshot -- Save/restore process's snapshot to/from file
start -- Start debugged program and stop at most convenient entry
stepuntil -- Step until a desired instruction in specific memory range
strings -- Display printable strings in memory
substr -- Search for substrings of a given string/number in memory
telescope -- Display memory content at an address with smart dereferences
tracecall -- Trace function calls made by the program
traceinst -- Trace specific instructions executed by the program
unptrace -- Disable anti-ptrace detection
utils -- Miscelaneous utilities from utils module
vmmap -- Get virtual mapping address ranges of section(s) in debugged process
waitfor -- Try to attach to new forked process; mimic "attach -waitfor"
xinfo -- Display detail information of address/registers
xormem -- XOR a memory region with a key
xprint -- Extra support to GDB's print command
xrefs -- Search for all call/data access references to a function/variable
xuntil -- Continue execution until an address or function
```

GEF

GEF

`GEF` is a kick-ass set of commands for X86, ARM, MIPS, PowerPC and SPARC to make GDB cool again for exploit dev. It is aimed to be used mostly by exploiters and reverse-engineers, to provide additional features to GDB using the Python API to assist during the process of dynamic analysis and exploit development.

It has full support for both Python2 and Python3 indifferently (as more and more distros start pushing `gdb` compiled with Python3 support).

- https://github.com/hugsy/gef
- Like a bigger PEDA but multi-architecture

GEF Claims A Lot

- **One** single GDB script.
- Entirely **OS Agnostic**, **NO** dependencies: GEF is battery-included and is installable in 2 seconds (unlike PwnDBG).
- **Fast** limiting the number of dependencies and optimizing code to make the commands as fast as possible (unlike *PwnDBG*).
- Built around an architecture abstraction layer, so all commands work in any GDB-supported architecture such as x86-32/64, ARMv5/6/7, AARCH64, SPARC, MIPS, PowerPC, etc. (unlike PEDA)

GEF Claims A Lot

- Provides more than **50** commands to drastically change your experience in GDB.
- **Easily** extendable to create other commands by providing more comprehensible layout to GDB Python API.
- Works consistently on both Python2 and Python3.
- Suited for real-life apps debugging, exploit development, just as much as CTF (unlike *PEDA* or *PwnDBG*)

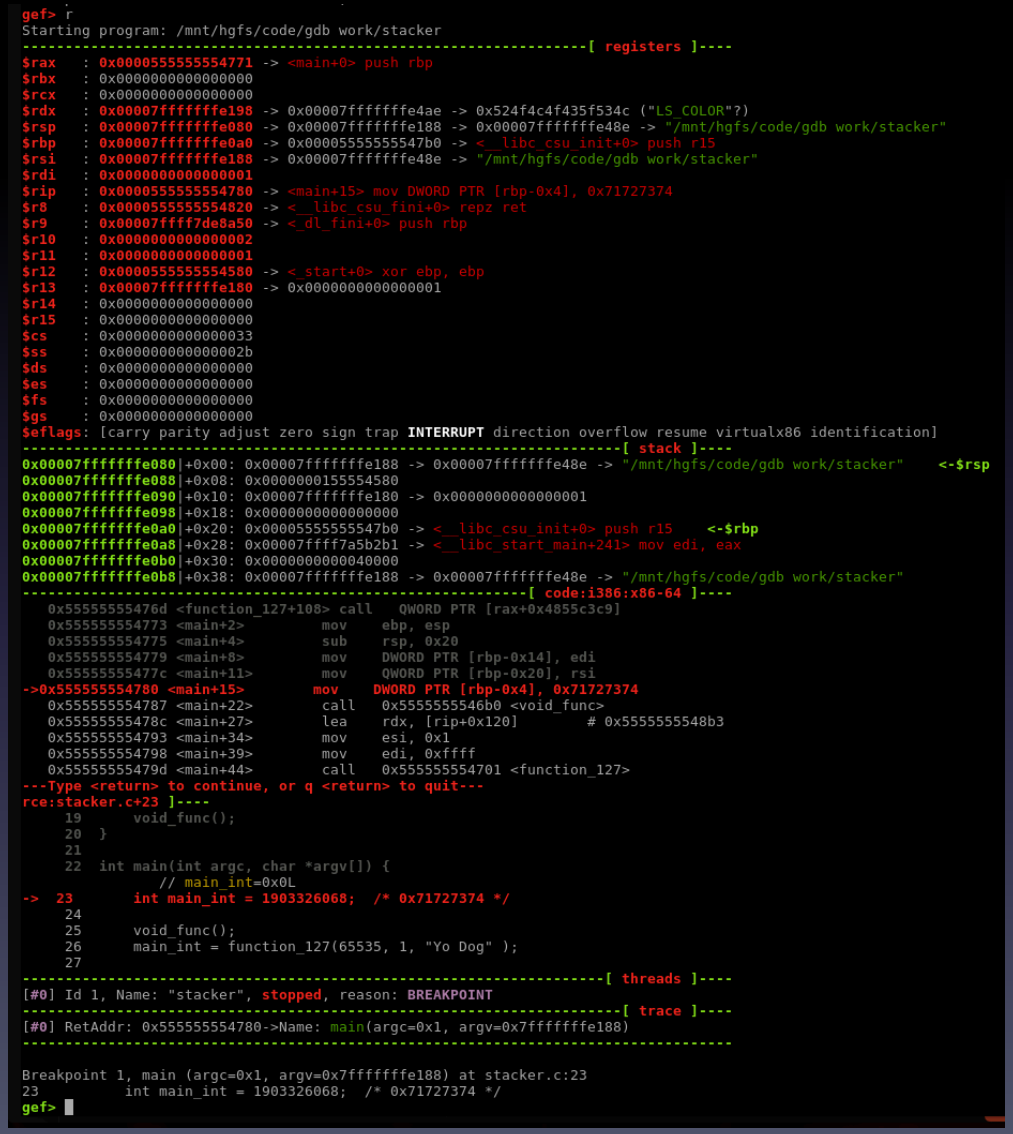
GEF

```
root@kali:/mnt/hgfs/code/gdb work# gdb -silent ./stacker
Reading symbols from ./stacker...done.
(gdb) b main
Breakpoint 1 at 0x780: file stacker.c, line 23.
(gdb) source ~/.gdbinit_gef
GEF for linux ready, type `gef' to start, `gef config' to configure
51 commands loaded for GDB 8.0.50.20170511-git using Python engine 2.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
```

- When started, it works but seems to want me to install additional python libraries
  - No big deal it even tells you what it needs and how to install it

```
gef> gef missing
[*] Command `retdec` is missing, reason -> Missing `retdec-python` package for Pyth
on2, install with: `pip2 install retdec-python`.
[*] Command `set-permission` is missing, reason -> Missing `keystone-engine` packag
e for Python2, install with: `pip2 install keystone-engine`.
```

GEF Displays



- Are Massive
- The sections are
  - Registers
  - Stack
  - Code
  - Threads
  - Trace

GEF

- We see Telescoping

```
$rbp   : 0x00007fffffffe0a0 -> 0x000055555555547b0 -> <__libc_csu_init+0> push r15
$rsi   : 0x00007fffffffe188 -> 0x00007fffffffe48e -> "/mnt/hgfs/code/gdb work/stacker"
$rdi   : 0x0000000000000001
$rip   : 0x0000555555554787 -> <main+22> call 0x5555555546b0 <void_func>
```

- Register Values

```
0x00007fffffffe080|+0x00: 0x00007fffffffe188 -> 0x00007fffffffe48e -> "/mnt/hgfs/code/gdb work/stacker" <-$rsp
0x00007fffffffe088|+0x08: 0x0000000155554580
```

- Backtraces without typing bt

```
[#0] RetAddr: 0x555555554560->Name: printf@plt()
[#1] RetAddr: 0x5555555546e6->Name: void_func()
[#2] RetAddr: 0x55555555478c->Name: main(argc=0x1, argv=0x7fffffffe188)
```

But Also Better Flow Displays

```
   0x555555554772 <main+1>          mov    rbp, rsp
   0x555555554775 <main+4>          sub    rsp, 0x20
   0x555555554779 <main+8>          mov    DWORD PTR [rbp-0x14], edi
   0x55555555477c <main+11>         mov    QWORD PTR [rbp-0x20], rsi
   0x555555554780 <main+15>         mov    DWORD PTR [rbp-0x4], 0x71727374
->0x555555554787 <main+22>        call   0x5555555546b0 <void_func>
   \->  0x5555555546b0 <void_func+0>    push   rbp
      0x5555555546b1 <void_func+1>    mov    rbp, rsp
      0x5555555546b4 <void_func+4>    sub    rsp, 0x10
      0x5555555546b8 <void_func+8>    mov    DWORD PTR [rbp-0x4], 0xf1f2f3f4
      0x5555555546bf <void_func+15>   lea    rax, [rip+0x172]        # 0x555555554838
      0x5555555546c6 <void_func+22>   mov    QWORD PTR [rbp-0x10], rax
```

Here you get a view of what is beyond a call and jump reasons

```
   0x7ffff7de79bb <_dl_fixup+75>    add    rbx, QWORD PTR [r8]
   0x7ffff7de79be <_dl_fixup+78>    cmp    ecx, 0x7
->0x7ffff7de79c1 <_dl_fixup+81>   jne    0x7ffff7de7b17 <_dl_fixup+423> NOT taken [Reason: !(!Z)]
   0x7ffff7de79c7 <_dl_fixup+87>    test   BYTE PTR [rsi+0x5], 0x3
   0x7ffff7de79cb <_dl_fixup+91>    jne    0x7ffff7de7a67 <_dl_fixup+247>
   0x7ffff7de79d1 <_dl_fixup+97>    mov    rax, QWORD PTR [r10+0x1c8]
   0x7ffff7de79d8 <_dl_fixup+104>   test   rax, rax
   0x7ffff7de79db <_dl_fixup+107>   je     0x7ffff7de7a90 <_dl_fixup+288>
```

Beginners Notes

- I believe that when you are learning this info is really helpful
  - All of these enhancements give good displays
  - You get a better feel for how the program flows
  - Register changes really jump out to your
- Provided you have a big display!

GEF Commands

```
gef> gef
----------------------------------------------------[ GEF - GDB Enhanced Features ]----------------------------------------
aslr                    -- View/modify GDB ASLR behavior.
canary                  -- Shows the canary value of the current process. Apply the techique detailed in
                           https://www.elttam.com.au/blog/playing-with-canaries/ to show the canary.
capstone-disassemble    -- Use capstone disassembly framework to disassemble code. (alias: cs-dis)
checksec                -- Checksec.sh (http://www.trapkit.de/tools/checksec.html) port.
context                 -- Display execution context. (alias: ctx)
dereference             -- Dereference recursively an address and display information (alias: telescope, dps)
edit-flags              -- Edit flags in a human friendly way (alias: flags)
elf-info                -- Display ELF header informations.
entry-break             -- Tries to find best entry point and sets a temporary breakpoint on it. (alias: start)
format-string-helper    -- Exploitable format-string helper: this command will set up specific breakpoints
                           at well-known dangerous functions (printf, snprintf, etc.), and check if the pointer
                           holding the format string is writable, and therefore susceptible to format string
                           attacks if an attacker can control its content. (alias: fmtstr-helper)
gef-remote              -- gef wrapper for the `target remote` command. This command will automatically
                           download the target binary in the local temporary directory (defaut /tmp) and then
                           source it. Additionally, it will fetch all the /proc/PID/maps and loads all its
                           information.
heap                    -- Base command to get information about the Glibc heap structure.
heap-analysis-helper    -- Heap vulnerability analysis helper: this command aims to track dynamic heap allocation
                           done through malloc()/free() to provide some insights on possible heap vulnerabilities. The
                           following vulnerabilities are checked:
                           - NULL free
                           - Use-after-Free
                           - Double Free
                           - Heap overlap
```

GEF Commands

```
- Heap Overlap
hexdump                 -- Display arranged hexdump (according to architecture endianness) of memory range.
hijack-fd               -- ChangeFdCommand: redirect file descriptor during runtime.
ida-interact            -- IDA Interact: set of commands to interact with IDA via a XML RPC service
                           deployed via the IDA script `ida_gef.py`. It should be noted that this command
                           can also be used to interact with Binary Ninja (using the script `binja_gef.py`)
                           using the same interface. (alias: binaryninja-interact, bn, binja)
ksymaddr                -- Solve kernel symbols from kallsyms table.
nop                     -- Patch the instruction(s) pointed by parameters with NOP.
patch                   -- Write specified values to the specified address.
pattern                 -- This command will create or search a De Bruijn cyclic pattern to facilitate
                           determining the offset in memory. The algorithm used is the same as the one
                           used by pwntools, and can therefore be used in conjunction.
pcustom                 -- Dump user defined structure.
                           This command attempts to reproduce WinDBG awesome `dt` command for GDB and allows
                           to apply structures (from symbols or custom) directly to an address.
                           Custom structures can be defined in pure Python using ctypes, and should be stored
                           in a specific directory, whose path must be stored in the `pcustom.struct_path`
                           configuration setting. (alias: dt)
printchar               -- Simply evaluates the provided expression and prints the result as an ASCII char.
                           Only exists to fix `p/c` which is broken in GDB when output-radix is set to 16.
                           See https://sourceware.org/bugzilla/show_bug.cgi?id=8678. (alias: pchar)
process-search          -- List and filter process. (alias: ps)
process-status          -- Extends the info given by GDB `info proc`, by giving an exhaustive description of the
                           process status (file descriptors, ancestor, descendants, etc.).  (alias: status)
```

GEF Commands

```
registers              -- Display full details on one, many or all registers value from current architecture.
reset-cache            -- Reset cache of all stored data.
ropper                 -- Ropper (http://scoding.de/ropper) plugin
search-pattern         -- SearchPatternCommand: search a pattern in memory. If given an hex value (starting with 0x)
                          the command will also try to look for upwards cross-references to this address. (alias: grep, xref)
shellcode              -- ShellcodeCommand uses @JonathanSalwan simple-yet-awesome shellcode API to
                          download shellcodes.
stub                   -- Stub out the specified function.
theme                  -- Customize GEF appearance.
trace-run              -- Create a runtime trace of all instructions executed from $pc to LOCATION specified.
unicorn-emulate        -- Use Unicorn-Engine to emulate the behavior of the binary, without affecting the GDB runtime.
                          By default the command will emulate only the next instruction, but location and number of
                          instruction can be changed via arguments to the command line. By default, it will emulate
                          the next instruction from current PC. (alias: emulate)
vmmap                  -- Display virtual memory mapping
xfiles                 -- Shows all libraries (and sections) loaded by binary (Truth is out there).
xinfo                  -- Get virtual section information for specific address
xor-memory             -- XOR a block of memory.
```

A lot of these commands are the ones we saw in PEDA but they work across multiple architectures now

- Is this important to you and your use case?

PWNDBG

Pwndbg

- https://github.com/pwndbg/pwndbg
- Pwndbg is a Python module which is loaded directly into GDB, and provides a suite of utilities and crutches to hack around all of the cruft that is GDB and smooth out the rough edges.
- Many other projects from the past (e.g., gdbinit, PEDA) and present (e.g. GEF) exist to fill some these gaps. Unfortunately, they're all either unmaintained, unmaintainable, or not well suited to easily navigating the code to hack in new features (respectively)

Dependencies

- pwndbg relies on
  - Unicorn Engine for emulation
    - Multi-architectures: Arm, Arm64 (Armv8), M68K, Mips, Sparc, & X86 (include X86_64)
    - http://www.unicorn-engine.org/
    - Yes, pwndbg supports the same architectures as gef
  - Capstone-engine for disassembly
    - http://www.capstone-engine.org/

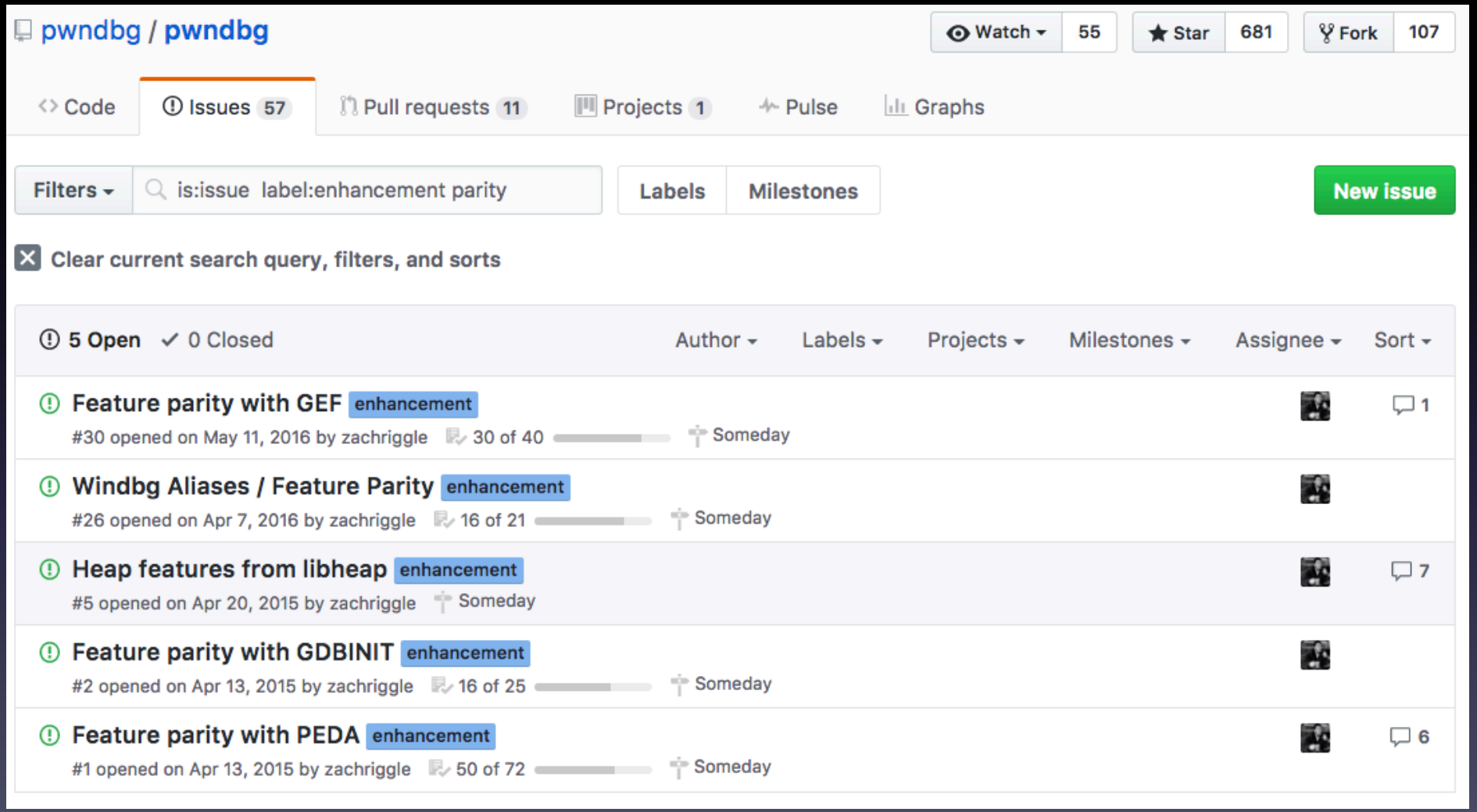
The install script takes care of dependencies

- So you need fairly strong permissions to get everything installed
- Takes care of everything

Just as cocky as gef

- Pwndbg exists not only to replace all of its predecessors, but also to have a clean implementation that runs quickly and is resilient against all the weird corner cases that come up

Feature Parity - Someday



Familiar Display

Not surprisingly we see a scrolling huge display that shows

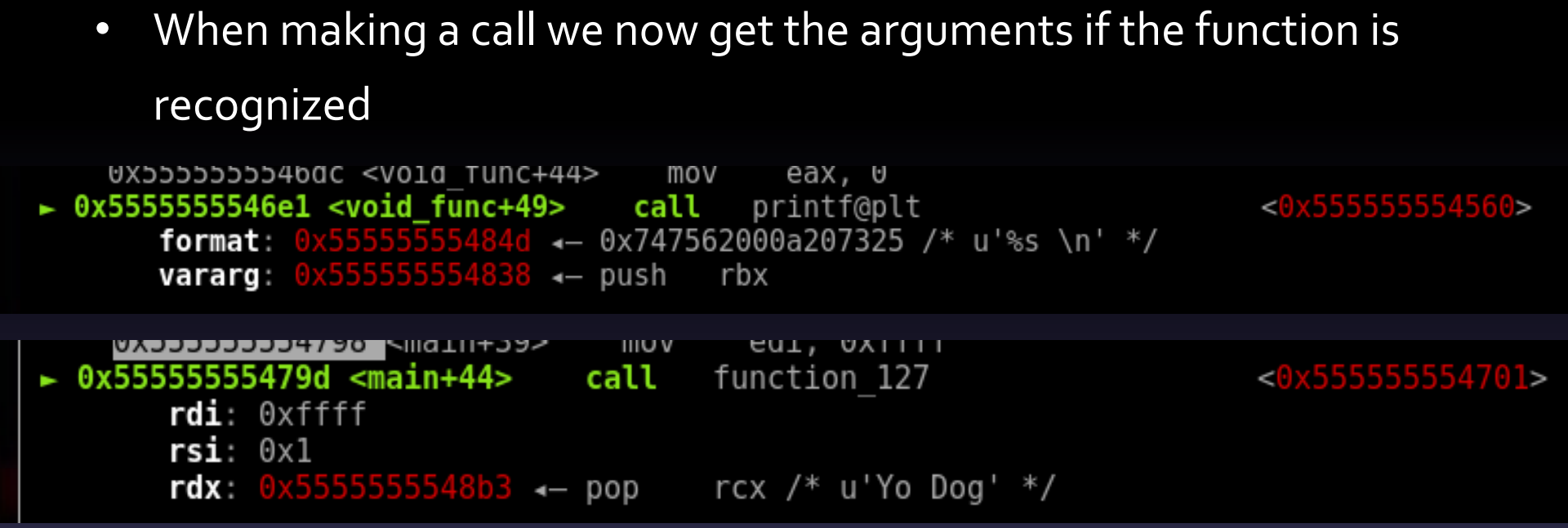
- Registers
- Disasm
- Source (if available)
- Stack
- Backtrace

And the other features we expect

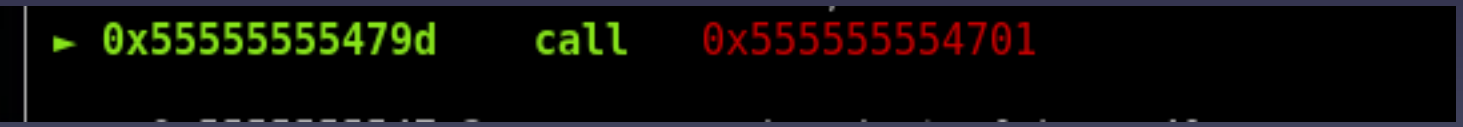
- Color coded memory addresses
- Telescoping
- Post jump code display

Unfamiliar Display

- When making a call we now get the arguments if the function is recognized



There is nothing on the stripped version, the call to printf() still is annotated though



pwndbg commands

- Like it says it tries to do a lot

```
Reading symbols from ./stacker...done.
(gdb) source ~/.gdbinit_pwndbg
Loaded 108 commands. Type pwndbg [filter] for a list.
pwndbg>
```

All those extra memory commands and feature parity add up compared to gef

```
Reading symbols from ./stacker...done.
(gdb) source ~/.gdbinit_gef
GEF for linux ready, type `gef' to start, `gef config' to configure
51 commands loaded for GDB 8.0.50.20170511-git using Python engine 2.7
[*] 2 commands could not be loaded, run `gef missing` to know why.
gef>
```

pwndbg commands

```
pwndbg>
address             Windbg compatibility alias for 'vmmap' command.
arena               Prints out the main arena or the arena at the specified by address.
arenas              Prints out allocated arenas
argc                Prints out the number of arguments.
args                Prints out the contents of argv.
argv                Prints out the contents of argv.
aslr                Inspect or modify ASLR status
auxv                Print information from the Auxiliary ELF Vector.
bc                  Clear the breapoint with the specified index.
bd                  Disable the breapoint with the specified index.
be                  Enable the breapoint with the specified index.
bins                Prints out the contents of the fastbins, unsortedbin, smallbins, and largebins from the
bl                  List breakpoints
bp                  Set a breakpoint at the specified address.
canary              Print out the current stack canary
checksec            Prints out the binary security settings. Attempts to call the binjitsu
config              Shows pwndbg-specific configuration points
configfile          Generates a configuration file for the current Pwndbg options
context             Print out the current register, instruction, and stack context.
cpsr                Print out the ARM CPSR register
```

pwndbg commands

```
cpsr                Print out the ARM cpsr register
da                  Dump a string at the specified address.
db                  Starting at the specified address, dump N bytes
dc                  None
dd                  Starting at the specified address, dump N dwords
dds                 Dump pointers and symbols at the specified address.
distance            Print the distance between the two arguments
dps                 Dump pointers and symbols at the specified address.
dq                  Starting at the specified address, dump N qwords
dqs                 Dump pointers and symbols at the specified address.
ds                  Dump a string at the specified address.
dt                  Dump out information on a type (e.g. ucontext_t).
dumpargs            If the current instruction is a call instruction, print that arguments.
dw                  Starting at the specified address, dump N words
eb                  Write hex bytes at the specified address.
ed                  Write hex dwords at the specified address.
elfheader           Prints the section mappings contained in the ELF header.
emulate             Like nearpc, but will emulate instructions from the current $PC forward.
entry               Set a breakpoint at the first instruction executed in
entry_point         GDBINIT compatibility alias to print the entry point.
env                 Prints out the contents of the environment.
environ             Prints out the contents of the environment.
envp                Prints out the contents of the environment.
eq                  Write hex qwords at the specified address.
errno               Converts errno (or argument) to its string representation.
ew                  Write hex words at the specified address.
ez                  Write a string at the specified address.
eza                 Write a string at the specified address.
```

pwndbg commands

```
eza                 Write a string at the specified address.
fastbins            Prints out the contents of the fastbins of the main arena or the arena
find_fake_fast      Finds candidate fake fast chunks that will overlap with the specified
fsbase              Prints out the FS base address.  See also $fsbase.
getfile             None
getpid              None
go                  Windbg compatibility alias for 'continue' command.
got                 Show the state of the Global Offset Table
gotplt              Prints any symbols found in the .got.plt section if it exists.
gsbase              Prints out the GS base address.  See also $gsbase.
heap                Prints out all chunks in the main_arena, or the arena specified by `addr`.
hexdump             Hexdumps data at the specified address (or at $sp)
init                GDBINIT compatibility alias for 'start' command.
j                   Synchronize IDA's cursor with GDB
k                   Print a backtrace (alias 'bt')
kd                  Dump pointers and symbols at the specified address.
largebins           Prints out the contents of the large bin of the main arena or the arena
libs                GDBINIT compatibility alias for 'libs' command.
lm                  Windbg compatibility alias for 'vmmap' command.
ln                  List the symbols nearest to the provided value.
main                GDBINIT compatibility alias for 'main' command.
malloc_chunk        Prints out the malloc_chunk at the specified address.
memfrob             memfrob(address, count)
mp                  Prints out the mp_ structure from glibc
nearpc              Disassemble near a specified address.
next_syscall        Breaks at the next syscall.
nextcall            Breaks at the next call instruction
nextjmp             Breaks at the next jump instruction
nextjump            Breaks at the next jump instruction
nextsc              Breaks at the next syscall.
```

pwndbg commands

```
nextsc                Breaks at the next syscall.
pc                    Windbg compatibility alias for 'nextcall' command.
pdisass               Compatibility layer for PEDA's pdisass command
peb                   None
pid                   None
plt                   Prints any symbols found in the .plt section if it exists.
procinfo              Display information about the running process.
pwndbg                Prints out a list of all pwndbg commands. The list can be optionally filtered if filter_pattern
is passed.
r2                    .
regs                  Print out all registers and enhance the information.
reinit_pwndbg         Makes pwndbg reinitialize all state.
reload                None
retaddr               Print out the stack addresses that contain return addresses
rop                   Dump ROP gadgets with Jon Salwan's ROPgadget tool.
ropgadget             None
ropper                ROP gadget search with ropper.
search                Search memory for byte sequences, strings, pointers, and integer values
smallbins             Prints out the contents of the small bin of the main arena or the arena
so                    Alias for stepover
sstart                GDBINIT compatibility alias for 'tbreak __libc_start_main; run' command.
stack                 dereferences on stack data with specified count and offset
start                 Set a breakpoint at a convenient location in the binary,
stepover              Sets a breakpoint on the instruction after this one
telescope             Recursively dereferences pointers starting at the specified address
theme                 Shows pwndbg-specific theme configuration points
themefile             Generates a configuration file for the current Pwndbg theme options
top_chunk             Prints out the address of the top chunk of the main arena, or of the arena
u                     Starting at the specified address, disassemble
unsortedbin           Prints out the contents of the unsorted bin of the main arena or the
version               Displays gdb, python and pwndbg versions.
vmmap                 Print the virtal memory map, or the specific mapping for the
vprot                 Windbg compatibility alias for 'vmmap' command.
xor                   xor(address, key, count)
```

pwndbg commands

- Some of those are convenience commands to make it easier for people used to WinDbg to feel at home
  - Just some basic stuff don't expect !analyze or !exploitable

pwndbg Code

- Well organized
  - Each command in its own file
  - Supportability is important to the author
- Heavier Duty Python
  - Uses more advanced python
    - More decorators
    - More outside library calls
    - More internal functions

What Does This All Mean?

- There are ready made packages that make gdb more pleasant to use
- As a beginner you may be helped by an enhanced display
- These enhancements add commands to make life easier
  - I'm shocked how useful shortcuts like next_syscall, nextcall, and nextjump are during assessment of code
  - In spite of pwndbg plans certain commands only exist in certain enhancements

Which Is Right for You

- None of the packages is perfect
  - Do you need the commands they support or just a better display?
  - Are you looking for a pure gdb or pure python solution?
    - Are you willing to have a few extra things installed for more features?
- What files can you get to where you are running gdb?

What is not covered

- We did not cover the other architectures that are supported
- We did not cover using a remote MI interface like gdb-server
- We did not go into every command
  - Explore them on your own, we would be here a long time demonstrating them all
- We did not cover stripped binaries
  - It is trickier to set breakpoints at the start and documentation features are more limited
    - b _start vs b main
    - getting past the library loading code)
  - But hey there is one less part of the display to show
    - no need to see source when there isn't any

Questions?

No QUESTIONS!

Do your labs!

- OK maybe a few questions…

- Do you want a part 2?
  - Tell me what you are interested in
    - More on writing python gdb extensions?
    - Something on stripped executable debugging?
    - More cowbell?