Pwndbg

Mensi Mohamed Amine

Abstract

Pwndbg is a GDB plugin designed to improve the debugging experience for exploit development and reverse engineering. It enhances GDB with advanced visualization, context-aware analysis, and custom commands, streamlining tasks like memory inspection and disassembly. Built in Python, Pwndbg is modular, fast, and widely used in security research and CTFs.

1 Introduction

Pwndbg enhances GDB for security research and reverse engineering by providing better context, improved displays, and custom features for vulnerability analysis, streamlining the debugging process.

Contents

1 Introduction **1**

2 Pwndbg Workflow **13**
2.1 Setting Up and Launching Pwndbg . . . 13
2.2 Initial Binary Reconnaissance . . . 13
2.3 Start Execution and Set Breakpoints . . . 13
2.4 Stack and Register Inspection . . . 13
2.5 Step, Continue, and Break Flow . . . 14
2.6 Exploiting a Buffer Overflow: A Practical Example . . . 14
2.7 Heap Analysis and Debugging . . . 14
2.8 Virtual Memory and Address Mapping . . . 14
2.9 Memory Searching and Scanning . . . 15
2.10 GOT/PLT Inspection . . . 15
2.11 Heap Exploitation with Fake Chunks . . . 15
2.12 Thread and TLS Management . . . 15
2.13 Memory Manipulation . . . 15
2.14 Pwndbg-Specific and Developer Tools . . . 15
2.15 Integration Tools . . . 16
2.16 Smart Context Tips . . . 16
2.17 Final Exploit Dev Flow Summary . . . 16

3 Start Commands **17**
3.1 `attachp` . . . 17
3.2 `entry` . . . 17
3.3 `sstart` . . . 18
3.4 `start [main, init]` . . . 18
3.5 Summary . . . 18

4 Step/Next/Continue Commands **19**
4.1 `nextcall` . . . 19
4.2 `nextjmp [nextjump]` . . . 19
4.3 `nextproginstr` . . . 19
4.4 `nextret` . . . 19
4.5 `nextsyscall [nextsc]` . . . 20
4.6 `stepover [so]` . . . 20
4.7 `stepret` . . . 20
4.8 `stepsyscall [stepsc]` . . . 20
4.9 `stepuntilasm` . . . 20
4.10 `xuntil` . . . 21
4.11 Summary of Commands . . . 21

5 Context Commands **22**
5.1 `context [ctx]` . . . 22
5.2 `contextnext [ctxn]` . . . 22
5.3 `contextoutput [ctx-out]` . . . 22
5.4 `contextprev [ctxp]` . . . 22
5.5 `contextsearch [ctxsearch]` . . . 22
5.6 `contextunwatch [ctx-unwatch, cunwatch]` . . . 23
5.7 `contextwatch [ctx-watch, cwatch]` . . . 23
5.8 `regs` . . . 23
5.9 Summary of Commands . . . 23

6 GLibc ptmalloc2 Heap Commands **24**
6.1 arena . . . 24
6.2 arenas . . . 24
6.3 bins . . . 24
6.4 fastbins . . . 24
6.5 find-fake-fast . . . 24
6.6 heap . . . 25
6.7 heap-config . . . 25
6.8 hi . . . 25
6.9 largebins . . . 25
6.10 malloc-chunk . . . 25
6.11 mp . . . 25
6.12 smallbins . . . 26
6.13 tcache . . . 26
6.14 tcachebins . . . 26
6.15 top-chunk . . . 26
6.16 try-free . . . 26
6.17 unsortedbin . . . 26
6.18 vis-heap-chunks [vis] . . . 27

7 jemalloc Heap Commands in pwndbg **28**
7.1 jemalloc-extent-info . . . 28
7.2 jemalloc-find-extent . . . 28
7.3 jemalloc-heap . . . 28

8 Breakpoint Commands **29**
8.1 break-if-taken . . . 29
8.2 break-if-not-taken . . . 29
8.3 breakrva [brva] (alias: brva) . . . 29
8.4 ignore . . . 29

9 Memory Commands **30**
9.1 distance . . . 30
9.2 gdt . . . 30
9.3 go-dump (alias: god) . . . 30
9.4 go-type (alias: goty) . . . 30
9.5 hexdump . . . 30
9.6 leakfind . . . 30
9.7 memfrob . . . 31
9.8 mmap . . . 31
9.9 mprotect . . . 31
9.10 p2p . . . 31
9.11 probeleak . . . 31
9.12 search . . . 31
9.13 telescope . . . 32
9.14 vmmap (aliases: lm, address, vprot, libs) . . . 32
9.15 vmmap-add . . . 32
9.16 vmmap-clear . . . 32
9.17 vmmap-explore . . . 32
9.18 xinfo . . . 32
9.19 xor . . . 33

10 Stack Commands **34**
10.1 canary . . . 34
10.2 retaddr . . . 34
10.3 stack . . . 34
10.4 stack-explore . . . 35
10.5 stackf . . . 35

11 Register Commands **36**
11.1 cpsr (aliases: xpsr, pstate) . . . 36
11.2 fsbase . . . 36
11.3 gsbase . . . 37
11.4 setflag . . . 37

12 Process Commands **38**
12.1 killthreads . . . 38
12.2 pid (alias: getpid) . . . 38
12.3 procinfo . . . 39

13 Linux/libc/ELF Commands **40**
13.1 argc . . . 40
13.2 argv . . . 40
13.3 aslr . . . 40
13.4 auxv and auxv-explore . . . 41
13.5 elfsections . . . 41
13.6 envp [env, environ] . . . 41
13.7 errno . . . 41
13.8 got and gotplt . . . 41
13.9 libcinfo . . . 42
13.10 linkmap . . . 42
13.11 onegadget . . . 42
13.12 piebase . . . 42
13.13 plt . . . 42
13.14 strings . . . 42
13.15 threads . . . 43
13.16 tls . . . 43
13.17 track-got and track-heap . . . 43

14 Disassemble Commands **44**
14.1 emulate . . . 44
14.2 nearpc (aliases: pdisass, u) . . . 44

15 Misc Commands **46**
15.1 asm . . . 46
15.2 checksec . . . 46
15.3 comm . . . 46
15.4 cyclic . . . 46
15.5 cymbol . . . 47
15.6 down . . . 47
15.7 dt . . . 47
15.8 dumpargs (alias: args) . . . 47
15.9 getfile . . . 47
15.10 hex2ptr . . . 47
15.11 hijack-fd . . . 48
15.12 ipi . . . 48
15.13 patch . . . 48

15.14 patch-list . . . 48
15.15 patch-revert . . . 48
15.16 plist . . . 48
15.17 sigreturn . . . 49
15.18 spray . . . 49
15.19 tips . . . 49
15.20 up . . . 49
15.21 valist . . . 49
15.22 vmmap-load . . . 49

16 Kernel Commands **51**
16.1 binder . . . 51
16.2 kbase . . . 51
16.3 kchecksec . . . 51
16.4 kcmdline . . . 51
16.5 kconfig . . . 51
16.6 klookup . . . 52
16.7 knft-dump . . . 52
16.8 knft-list-chains . . . 52
16.9 knft-list-exprs . . . 52
16.10 knft-list-flowtables . . . 52
16.11 knft-list-objects . . . 52
16.12 knft-list-rules . . . 53
16.13 knft-list-sets . . . 53
16.14knft-list-tables . . . 53
16.15 kversion . . . 53
16.16 pcplist . . . 53
16.17 slab . . . 53

17 Integration Commands **54**
17.1 ai . . . 54
17.2 bn-sync [bns] . . . 54
17.3 decomp . . . 54
17.4 ghidra . . . 54
17.5 j . . . 54
17.6 r2 [radare2] . . . 55
17.7 r2pipe . . . 55
17.8 rop [ropgadget] . . . 55
17.9 ropper . . . 55
17.10rz [rizin] . . . 55
17.11 rzpipe . . . 56
17.12 save-ida . . . 56

18 WinDbg Commands **57**
18.1 bc . . . 57
18.2 bd . . . 57
18.3 be . . . 57
18.4 bl . . . 57
18.5 bp . . . 57
18.6 da / ds . . . 58
18.7 db . . . 58
18.8 dc . . . 58
18.9 dd . . . 58
18.10 dds (aliases: kd, dps, dqs) . . . 58

18.11 dq . . . 58
18.12 dw . . . 59
18.13 eb . . . 59
18.14 ed . . . 59
18.15 eq . . . 59
18.16 ew . . . 59
18.17 ez / eza . . . 59
18.18 go . . . 60
18.19 k (alias: bt) . . . 60
18.20 ln . . . 60
18.21 pc . . . 60
18.22 peb . . . 60

19 Pwndbg Commands **61**
19.1 bugreport . . . 61
19.2 config . . . 61
19.3 configfile . . . 61
19.4 memoize . . . 61
19.5 profiler . . . 62
19.6 pwndbg . . . 62
19.7 reinit-pwndbg . . . 62
19.8 reload . . . 62
19.9 theme . . . 62
19.10 themefile . . . 63
19.11 version . . . 63

20 Pwndbg Developer Commands **64**
20.1 dev-dump-instruction . . . 64
20.2 log-level . . . 64

Pwndbg Command Reference

Start Commands

Command [Aliases]	Description
attachp	Attaches to a given pid, process name, process found with partial argv match or to a device file.
entry	Start the debugged program stopping at its entrypoint address.
sstart	Alias for `tbreak __libc_start_main; run`.
start [main, init]	Start the debugged program stopping at the first convenient location.

Step/Next/Continue Commands

Command [Aliases]	Description
nextcall	Breaks at the next call instruction.
nextjmp [nextjump]	Breaks at the next jump instruction.
nextproginstr	Breaks at the next instruction that belongs to the running program.
nextret	Breaks at next return-like instruction.
nextsyscall [nextsc]	Breaks at the next syscall not taking branches.
stepover [so]	Breaks on the instruction after this one.
stepret	Breaks at next return-like instruction by 'stepping' to it.
stepsyscall [stepsc]	Breaks at the next syscall by taking branches.
stepuntilasm	Breaks on the next matching instruction.
xuntil	Continue execution until an address or expression.

Context Commands

Command [Aliases]	Description
context [ctx]	Print out the current register, instruction, and stack context.
contextnext [ctxn]	Select next entry in context history.
contextoutput [ctx-out]	Sets the output of a context section.
contextprev [ctxp]	Select previous entry in context history.
contextsearch [ctxsearch]	Search for a string in the context history and select that entry.
contextunwatch [ctx-unwatch, cunwatch]	Removes an expression previously added to be watched.
contextwatch [ctx-watch, cwatch]	Adds an expression to be shown on context.
regs	Print out all registers and enhance the information.

GLibc ptmalloc2 Heap Commands

Command [Aliases]	Description
arena	Print the contents of an arena.
arenas	List this process's arenas.
bins	Print the contents of all an arena's bins and a thread's tcache.
fastbins	Print the contents of an arena's fastbins.

find-fake-fast	Find candidate fake fast or tcache chunks overlapping the specified address.
heap	Iteratively print chunks on a heap.
heap-config	Shows heap related configuration.
hi	Searches all heaps to find if an address belongs to a chunk. If yes, prints the chunk.
largebins	Print the contents of an arena's largebins.
malloc-chunk	Print a chunk.
mp	Print the mp_ struct's contents.
smallbins	Print the contents of an arena's smallbins.
tcache	Print a thread's tcache contents.
tcachebins	Print the contents of a tcache.
top-chunk	Print relevant information about an arena's top chunk.
try-free	Check what would happen if free was called with given address.
unsortedbin	Print the contents of an arena's unsortedbin.
vis-heap-chunks [vis]	Visualize chunks on a heap.

jemalloc Heap Commands

Command [Aliases]	**Description**
jemalloc-extent-info	Prints extent information for the given address.
jemalloc-find-extent	Returns extent information for pointer address allocated by jemalloc.
jemalloc-heap	Prints all extents information.

Breakpoint Commands

Command [Aliases]	**Description**
break-if-not-taken	Breaks on a branch if it is not taken.
break-if-taken	Breaks on a branch if it is taken.
breakrva [brva]	Break at RVA from PIE base.
ignore	Set ignore-count of breakpoint number N to COUNT.

Memory Commands

Command [Aliases]	**Description**
distance	Print the distance between the two arguments, or print the offset to the address's page base.
gdt	Decode X86-64 GDT entries at address.
go-dump [god]	Dumps a Go value of a given type at a specified address.
go-type [goty]	Dumps a Go runtime reflection type at a specified address.
hexdump	Hexdumps data at the specified address or module name.
leakfind	Attempt to find a leak chain given a starting address.
memfrob	Memfrobs a region of memory (xor with '*').
mmap	Calls the mmap syscall and prints its resulting address.
mprotect	Calls the mprotect syscall and prints its result value.
p2p	Pointer to pointer chain search. Searches given mapping for all pointers that point to specified mapping.
probeleak	Pointer scan for possible offset leaks.

search	Search memory for byte sequences, strings, pointers, and integer values.
telescope	Recursively dereferences pointers starting at the specified address.
vmmap [lm, address, vprot, libs]	Print virtual memory map pages.
vmmap-add	Add virtual memory map page.
vmmap-clear	Clear the vmmap cache.
vmmap-explore	Explore a page, trying to guess permissions.
xinfo	Shows offsets of the specified address from various useful locations.
xor	XOR ‘count‘ bytes at ‘address‘ with the key ‘key‘.

Stack Commands

Command [Aliases]	Description
canary	Print out the current stack canary.
retaddr	Print out the stack addresses that contain return addresses.
stack	Dereferences on stack data with specified count and offset.
stack-explore	Explore stack from all threads.
stackf	Dereferences on stack data, printing the entire stack frame with specified count and offset.

Register Commands

Command [Aliases]	Description
cpsr [xpsr, pstate]	Print out ARM CPSR or xPSR register.
fsbase	Prints out the FS base address. See also $fsbase.
gsbase	Prints out the GS base address. See also $gsbase.
setflag [flag]	Modify the flags register.

Process Commands

Command [Aliases]	Description
killthreads	Kill all or given threads.
pid [getpid]	Gets the pid.
procinfo	Display information about the running process.

Linux/libc/ELF Commands

Command [Aliases]	Description
argc	Prints out the number of arguments.
argv	Prints out the contents of argv.
aslr	Check the current ASLR status, or turn it on/off.
auxv	Print information from the Auxiliary ELF Vector.
auxv-explore	Explore and print information from the Auxiliary ELF Vector.
elfsections	Prints the section mappings contained in the ELF header.
envp [env, environ]	Prints out the contents of the environment.
errno	Converts errno (or argument) to its string representation.
got	Show the state of the Global Offset Table.
gotplt	Prints any symbols found in the .got.plt section if it exists.

libcinfo	Show libc version and link to its sources.
linkmap	Show the state of the Link Map.
onegadget	Find gadgets which single-handedly give code execution.
piebase	Calculate VA of RVA from PIE base.
plt	Prints any symbols found in the .plt section if it exists.
strings	Extracts and displays ASCII strings from readable memory pages of the debugged process.
threads	List all threads belonging to the selected inferior.
tls	Print out base address of the current Thread Local Storage (TLS).
track-got	Controls GOT tracking.
track-heap	Manages the heap tracker.

Disassemble Commands

Command [Aliases]	Description
emulate	Like nearpc, but will emulate instructions from the current $PC forward.
nearpc [pdisass, u]	Disassemble near a specified address.

Misc Commands

Command [Aliases]	Description
asm	Assemble shellcode into bytes.
checksec	Prints out the binary security settings using ‘checksec‘.
comm	Put comments in assembly code.
cyclic	Cyclic pattern creator/finder.
cymbol	Add, show, load, edit, or delete custom structures in plain C.
down	Select and print stack frame called by this one.
dt	Dump out information on a type (e.g. ucontext_t).
dumpargs [args]	Prints determined arguments for call instruction.
getfile	Gets the current file.
hex2ptr	Converts a space-separated hex string to a little-endian address.
hijack-fd	Replace a file descriptor of a debugged process.
ipi	Start an interactive IPython prompt.
patch	Patches given instruction with given code or bytes.
patch-list	List all patches.
patch-revert	Revert patch at given address.
plist	Dumps the elements of a linked list.
sigreturn	Display the SigreturnFrame at the specific address.
spray	Spray memory with cyclic() generated values.
tips	Shows tips.
up	Select and print stack frame that called this one.
valist	Dumps the arguments of a va_list.
vmmap-load	Load virtual memory map pages from ELF file.

Kernel Commands

Command [Aliases]	Description
binder	Show Android Binder information.

kbase	Finds the kernel virtual base address.
kchecksec	Checks for kernel hardening configuration options.
kcmdline	Return the kernel commandline (/proc/cmdline).
kconfig	Outputs the kernel config (requires CONFIG_IKCONFIG).
klookup	Lookup kernel symbols.
knft-dump	Dump all nftables: tables, chains, rules, expressions.
knft-list-chains	Dump netfilter chains form a specific table.
knft-list-exprs	Dump only expressions from specific rule.
knft-list-flowtables	Dump netfilter flowtables from a specific table.
knft-list-objects	Dump netfilter objects from a specific table.
knft-list-rules	Dump netfilter rules form a specific chain.
knft-list-sets	Dump netfilter sets from a specific table.
knft-list-tables	Dump netfliter tables from a specific network namespace.
kversion	Outputs the kernel version (/proc/version).
pcplist	Print Per-CPU page list.
slab	Prints information about the slab allocator.

Integrations Commands

Command [Aliases]	Description
ai	Ask GPT-3 a question about the current debugging context.
bn-sync [bns]	Synchronize Binary Ninja's cursor with GDB.
decomp	Use the current integration to decompile code near an address.
ghidra	Decompile a given function using Ghidra.
j	Synchronize IDA's cursor with GDB.
r2 [radare2]	Launches radare2.
r2pipe	Execute stateful radare2 commands through r2pipe.
rop [ropgadget]	Dump ROP gadgets with Jon Salwan's ROPgadget tool.
ropper	ROP gadget search with ropper.
rz [rizin]	Launches rizin.
rzpipe	Execute stateful rizin commands through rzpipe.
save-ida	Save the ida database.

WinDbg Commands

Command [Aliases]	Description
bc	Clear the breakpoint with the specified index.
bd	Disable the breakpoint with the specified index.
be	Enable the breakpoint with the specified index.
bl	List breakpoints.
bp	Set a breakpoint at the specified address.
da	Dump a string at the specified address.
db	Starting at the specified address, dump N bytes.
dc	Starting at the specified address, hexdump.
dd	Starting at the specified address, dump N dwords.
dds [kd, dps, dqs]	Dump pointers and symbols at the specified address.
dq	Starting at the specified address, dump N qwords.
ds	Dump a string at the specified address.
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.
eq	Write hex qwords at the specified address.
ew	Write hex words at the specified address.
ez	Write a string at the specified address.
eza	Write a string at the specified address.
go	Windbg compatibility alias for 'continue' command.
k	Print a backtrace (alias 'bt').
ln	List the symbols nearest to the provided value.
pc	Windbg compatibility alias for 'nextcall' command.
peb	Not be windows.

pwndbg Commands

Command [Aliases]	**Description**
bugreport	Generate a bug report.
config	Shows pwndbg-specific configuration.
configfile	Generates a configuration file for the current pwndbg options.
memoize	Toggles memoization (caching).
profiler	Utilities for profiling pwndbg.
pwndbg	Prints out a list of all pwndbg commands.
reinit-pwndbg	Makes pwndbg reinitialize all state.
reload	Reload pwndbg.
theme	Shows pwndbg-specific theme configuration.
themefile	Generates a configuration file for the current pwndbg theme options.
version	Displays Pwndbg and its important deps versions.

Developer Commands

Command [Aliases]	**Description**
dev-dump-instruction	Dump internal PwndbgInstruction attributes.
log-level	Set the log level.

2 Pwndbg Workflow

Goal of this Guide

This section covers how to:

- Load and inspect a binary
- Use advanced Pwndbg commands
- Perform heap/stack analysis
- Analyze function calls and memory layout
- Conduct exploit development tasks

2.1 Setting Up and Launching Pwndbg

```
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh
gdb ./vuln
```

2.2 Initial Binary Reconnaissance

Check security protections:

```
checksec
```

Inspect functions and symbols:

```
info functions
plt
got
```

2.3 Start Execution and Set Breakpoints

```
start
entry
break vuln
break *0x401234
run < input.txt
```

2.4 Stack and Register Inspection

```
context
regs
telescope $rsp
x/10x $rsp
x/s $rdi
```

2.5 Step, Continue, and Break Flow

```
stepover
nextcall
nextret
nextjmp
stepsyscall
```

2.6 Exploiting a Buffer Overflow: A Practical Example

Example C code:

```
void vuln() {
    char buf[64];
    gets(buf);
    puts("Done!");
}
int main() {
    vuln();
    return 0;
}
```

Compile:

```
gcc -no-pie -fno-stack-protector -o vuln vuln.c
```

Trigger and analyze crash:

```
gdb ./vuln
break vuln
run <<< $(cyclic 100)
info registers rip
pattern_offset 0x6161616e
```

Controlled RIP:

```
python3 -c 'print("A"*offset + "BBBB")' > payload
run < payload
info registers rip
```

2.7 Heap Analysis and Debugging

```
heap
bins
tcache
fastbins
top-chunk
malloc-chunk 0x602010
```

2.8 Virtual Memory and Address Mapping

```
vmmap
xinfo 0x601050
piebase
```

2.9 Memory Searching and Scanning

```
search "flag"
search 0xdeadbeef
p2p
leakfind
```

2.10 GOT/PLT Inspection

```
gotplt
plt
got
rop
rop --search 'pop rdi; ret'
```

2.11 Heap Exploitation with Fake Chunks

```
find-fake-fast 0xdeadbeef
try-free 0x602020
vis-heap-chunks
```

2.12 Thread and TLS Management

```
threads
stack-explore
tls
```

2.13 Memory Manipulation

```
patch 0x401000 "nop"
patch-list
patch-revert 0x401000
mprotect 0x601000 0x1000 rwx
set {int}0x601050 = 1234
```

2.14 Pwndbg-Specific and Developer Tools

```
theme
config
pwndbg
reload
```

2.15 Integration Tools

Tool	Command	Description
Ghidra	ghidra	Decompile using Ghidra
IDA	j	Sync cursor with IDA
BinaryNinja	bn-sync	Sync with Binary Ninja
radare2	r2 / r2pipe	Launch or sync r2
ROPGadget	rop	Gadget search
ropper	ropper	Another gadget finder

2.16 Smart Context Tips

```
contextwatch *(char**)($rsp+0x8)
contextunwatch *(char**)($rsp+0x8)
```

2.17 Final Exploit Dev Flow Summary

1. Load binary: `gdb ./vuln`
2. Start with `start`, `entry`, or `break main`
3. Run a cyclic pattern: `run <<<` $(cyclic 200) $Checkcrash$: `info reg rip, pattern_offset`
4. Craft payload: padding + control data
5. Inspect memory layout: `vmmap`, `telescope`
6. Dump and patch memory as needed
7. Look for ROP gadgets: `rop --search`
8. Leak addresses using: `got`, `heap`, `libcinfo`
9. Repeat until code execution is achieved

3 Start Commands

In the context of `pwndbg`, which is a powerful debugging tool designed for use with `GDB` (GNU Debugger), these commands are used for various tasks related to debugging binary programs, particularly in binary exploitation and reverse engineering. Let's go through each of the commands listed in your query in detail, explaining them with examples:

3.1 attachp

Purpose: This command allows you to **attach** GDB to a running process. You can attach using several methods: by specifying a **PID** (process ID), a **process name**, a **partial match of the command line arguments** (argv), or even a **device file**.

Usage:

```
pwndbg> attachp <pid>
pwndbg> attachp <process_name>
pwndbg> attachp <partial_argv>
pwndbg> attachp <device_file>
```

Examples:

- Attaching to a process by PID:

  ```
  pwndbg> attachp 1234
  ```

  This attaches the debugger to the process with ID 1234.

- Attaching to a process by name:

  ```
  pwndbg> attachp firefox
  ```

  This would attach to the `Firefox` process, assuming it's running.

- Attaching using a partial match:

  ```
  pwndbg> attachp 'sshd'
  ```

  This will attach to any process whose command line includes the string `sshd`.

3.2 entry

Purpose: This command will **start the program being debugged** and cause GDB to **break (stop) execution at the entry point** of the program. The entry point is the address where the program starts executing, typically the `_start` symbol or the `main()` function.

Usage:

```
pwndbg> entry
```

Example: When you execute the command, GDB will start the program, but it will halt as soon as the entry point is reached.

```
pwndbg> entry
```

If the program is designed to start execution at `main()`, the debugger will stop execution just before `main()` is called.

3.3 sstart

Purpose: This command is an **alias** for a specific sequence of actions that break (stop) the program at `__libc_start_main` (which is part of the C runtime library and a common starting point for many programs). Then it runs the program.

Usage:

```
pwndbg> sstart
```

Example: This command effectively does two things:

- **Sets a temporary breakpoint** (using `tbreak`) at the address of `__libc_start_main`, which is called early in program startup.
- **Runs the program** from the beginning after setting the breakpoint.

Once the program hits the `__libc_start_main` function, execution will stop, and you can start debugging the execution flow from that point.

3.4 start [main, init]

Purpose: This command starts the debugged program, but it will stop at the **first convenient location** (such as the entry point or the beginning of the `main` function or `init` function). This is useful if you want to begin debugging from the start, but without having to manually specify the breakpoints yourself.

Usage:

```
pwndbg> start main
pwndbg> start init
```

Examples:

- **Stopping at `main()`**:

  ```
  pwndbg> start main
  ```

  This will start the program and stop execution at the `main` function if it's defined. This is the typical location where you want to begin debugging in most programs.

- **Stopping at the `init` function**:

  ```
  pwndbg> start init
  ```

  This will cause the debugger to stop at the `initialization routines` that might be present in the program before the main logic is executed.

3.5 Summary

- `attachp`: Attach the debugger to a running process using a PID, process name, partial argv, or device file.
- `entry`: Start the program and break at the entry point.
- `sstart`: Set a breakpoint at `__libc_start_main` and start the program.
- `start [main, init]`: Start the program and stop at `main()` or `init()` or another first convenient location, depending on what's specified.

These commands provide different ways to begin the debugging process in `pwndbg` and `GDB`, offering flexibility in how you start or attach to processes for analysis.

4 Step/Next/Continue Commands

The **Step/Next/Continue Commands** in **pwndbg** (a GDB extension for advanced debugging, particularly in binary exploitation) allow you to control the flow of execution in a debugged program by stepping through code, continuing to specific points, or breaking on specific events. Below is an explanation of each command, along with examples.

4.1 `nextcall`

- **Purpose**: Breaks at the **next call instruction**.
- **Usage**:

```
pwndbg> nextcall
```

- **Example**: If the next instruction is `call foo`, the debugger breaks before the call to `foo`.

4.2 `nextjmp [nextjump]`

- **Purpose**: Breaks at the **next jump instruction** (e.g., `jmp`, `je`, `jz`).
- **Usage**:

```
pwndbg> nextjmp
```

- **Example**: Stops at the first encountered jump, useful for tracking loops or conditional branches.

4.3 `nextproginstr`

- **Purpose**: Breaks at the next instruction that belongs to the running program (excluding system/library code).
- **Usage**:

```
pwndbg> nextproginstr
```

- **Example**: Avoids stopping inside system/library functions.

4.4 `nextret`

- **Purpose**: Breaks at the **next return-like instruction**.
- **Usage**:

```
pwndbg> nextret
```

- **Example**: Useful for breaking just before exiting a function.

4.5 nextsyscall [nextsc]

- **Purpose**: Breaks at the **next system call** that does not take branches.
- **Usage**:

```
pwndbg> nextsyscall
```

- **Example**: Stops just before a syscall like `sys_read`.

4.6 stepover [so]

- **Purpose**: Steps over the next instruction, skipping into function calls.
- **Usage**:

```
pwndbg> stepover
```

- **Example**: If the next instruction is `call foo`, it will run `foo` and stop afterward.

4.7 stepret

- **Purpose**: Steps to the **next return-like instruction** by executing until reaching a return.
- **Usage**:

```
pwndbg> stepret
```

- **Example**: Exits the current function automatically and stops at return.

4.8 stepsyscall [stepsc]

- **Purpose**: Steps through the **next system call**, including branches.
- **Usage**:

```
pwndbg> stepsyscall
```

- **Example**: Traces detailed execution inside a syscall like `write`.

4.9 stepuntilasm

- **Purpose**: Breaks on the **next matching instruction** pattern in the disassembly.
- **Usage**:

```
pwndbg> stepuntilasm
```

- **Example**: Useful for locating the next `call` or `jmp` instruction.

4.10 `xuntil`

- **Purpose**: Continues execution until a specified **address or expression** evaluates true.
- **Usage**:

```
pwndbg> xuntil <address_or_expression>
```

- **Example**:

```
pwndbg> xuntil 0x601000
```

Stops execution when the program reaches address `0x601000`.

4.11 Summary of Commands

- `nextcall`: Break at the next function call.
- `nextjmp`: Break at the next jump instruction.
- `nextproginstr`: Break at the next instruction in the main program.
- `nextret`: Break at the next return-like instruction.
- `nextsyscall`: Break at the next syscall (non-branching).
- `stepover`: Step over a function call.
- `stepret`: Step until the next return-like instruction.
- `stepsyscall`: Step through a syscall, including branches.
- `stepuntilasm`: Break at the next matching assembly instruction.
- `xuntil`: Continue until a specific address or condition is met.

These commands offer fine-grained control over program execution, enhancing your ability to debug effectively.

5 Context Commands

The **Context Commands** in **pwndbg** provide detailed runtime information such as register states, disassembly, and stack layout, allowing the user to monitor and interact with the debugger's internal views. These commands are especially useful for real-time tracking of program state and historical changes.

5.1 `context [ctx]`

- **Purpose**: Displays the current context including **registers, disassembly, stack**, and other sections.
- **Usage**:

```
pwndbg> context
```

- **Example**: Shows a combined overview of the program's current execution state.

5.2 `contextnext [ctxn]`

- **Purpose**: Selects the **next entry in the context history**.
- **Usage**:

```
pwndbg> contextnext
```

- **Example**: Move forward through previously displayed context snapshots.

5.3 `contextoutput [ctx-out]`

- **Purpose**: Sets the output destination of a context section (e.g., redirecting output).
- **Usage**:

```
pwndbg> contextoutput
```

- **Example**: Could be used to route context output to a log file or console.

5.4 `contextprev [ctxp]`

- **Purpose**: Selects the **previous entry** in the context history.
- **Usage**:

```
pwndbg> contextprev
```

- **Example**: Move backward through context snapshots.

5.5 `contextsearch [ctxsearch]`

- **Purpose**: Searches the context history for a given string and jumps to the corresponding entry.
- **Usage**:

```
pwndbg> contextsearch <search_string>
```

- **Example**: Find an earlier context where a specific value or symbol was present.

5.6 contextunwatch [ctx-unwatch, cunwatch]

- **Purpose**: Removes an expression from being watched in the context view.
- **Usage**:

```
pwndbg> contextunwatch <expression>
```

- **Example**: Stop tracking a variable or memory location.

5.7 contextwatch [ctx-watch, cwatch]

- **Purpose**: Adds an expression to the context display to be watched over time.
- **Usage**:

```
pwndbg> contextwatch <expression>
```

- **Example**: Watch a local variable or memory address across function calls.

5.8 regs

- **Purpose**: Prints all register values, enhancing their display with annotations (e.g., function names, strings).
- **Usage**:

```
pwndbg> regs
```

- **Example**: Outputs a formatted table of CPU registers with contextual insights.

5.9 Summary of Commands

- `context`: Show current context (registers, stack, disassembly).
- `contextnext`: Move to next context history entry.
- `contextoutput`: Set the output destination of context sections.
- `contextprev`: Move to previous context history entry.
- `contextsearch`: Search for a keyword in context history.
- `contextunwatch`: Stop watching an expression in the context view.
- `contextwatch`: Add an expression to be monitored in the context.
- `regs`: Print all registers with enhanced information.

These commands are essential for visualizing and navigating the execution state of a program during debugging sessions.

6 GLibc ptmalloc2 Heap Commands

These commands in `pwndbg` provide introspection into the **GLibc** `ptmalloc2` heap allocator. They're particularly useful when analyzing heap-based vulnerabilities, debugging memory corruption bugs, or understanding memory allocation behavior.

6.1 `arena`

Purpose: Print the contents of the current **arena** (a structure used by `ptmalloc2` to manage memory).
Usage:

```
pwndbg> arena
```

Example: Displays metadata like top chunk, bins, and other arena-specific data structures.

6.2 `arenas`

Purpose: List all arenas associated with the current process.
Usage:

```
pwndbg> arenas
```

Example: Shows `main_arena` and any thread-specific arenas (useful in multi-threaded applications).

6.3 `bins`

Purpose: Print the contents of all bins (fast, small, large, unsorted) and the thread's `tcache`.
Usage:

```
pwndbg> bins
```

Example: Useful to get a snapshot of all free chunks and where they are stored.

6.4 `fastbins`

Purpose: Display the fastbins (single-linked lists for small freed chunks).
Usage:

```
pwndbg> fastbins
```

Example: Shows the layout and state of fastbins in the current arena.

6.5 `find-fake-fast`

Purpose: Search for fake fastbin or tcache chunks that may overlap a specific address.
Usage:

```
pwndbg> find-fake-fast <address>
```

Example: Used in heap exploitation to find overlap candidates for creating fake chunks.

6.6 heap

Purpose: Iteratively print heap chunks from the base to the top of the heap.
Usage:

```
pwndbg> heap
```

Example: Visualizes each chunk with metadata (size, prev_size, in-use bit).

6.7 heap-config

Purpose: Show pwndbg-specific configuration related to heap inspection.
Usage:

```
pwndbg> heap-config
```

Example: Lists visual options like whether color or ASCII views are enabled.

6.8 hi

Purpose: Check if a given address belongs to a heap chunk. If yes, display it.
Usage:

```
pwndbg> hi <address>
```

Example:

```
pwndbg> hi 0x602010
```

6.9 largebins

Purpose: Print the largebins (used for large chunks).
Usage:

```
pwndbg> largebins
```

Example: Display contents of bins used for allocations larger than smallbin thresholds.

6.10 malloc-chunk

Purpose: Print metadata of a single heap chunk at the given address.
Usage:

```
pwndbg> malloc-chunk <address>
```

Example:

```
pwndbg> malloc-chunk 0x603000
```

6.11 mp

Purpose: Print the contents of the mp_ struct (tunable malloc parameters).
Usage:

```
pwndbg> mp
```

Example: Shows options like `trim_threshold`, `mmap_threshold`, etc.

6.12 smallbins

Purpose: Print the contents of an arena's smallbins.
Usage:

```
pwndbg> smallbins
```

Example: Shows free chunks of sizes between `0x20` and `0x400`.

6.13 tcache

Purpose: Print the tcache (thread-local caching system).
Usage:

```
pwndbg> tcache
```

Example: Displays the structure holding recently freed chunks for quick reuse.

6.14 tcachebins

Purpose: Print contents of the tcache bins.
Usage:

```
pwndbg> tcachebins
```

Example: Breaks down the contents of the thread-local tcache into per-size bins.

6.15 top-chunk

Purpose: Show information about the top chunk (unused chunk at the end of the heap).
Usage:

```
pwndbg> top-chunk
```

Example: Helpful in checking whether a `malloc` will request more memory.

6.16 try-free

Purpose: Simulate a `free()` on a given address to see what would happen.
Usage:

```
pwndbg> try-free <address>
```

Example: Helps you identify double free or invalid free issues.

6.17 unsortedbin

Purpose: Print the contents of the unsorted bin.
Usage:

```
pwndbg> unsortedbin
```

Example: Shows recently freed chunks that haven't been sorted into size-specific bins yet.

6.18 `vis-heap-chunks [vis]`

Purpose: Visualize heap chunks graphically or structurally.
Usage:

```
pwndbg> vis-heap-chunks
```

Example: Provides a tree/graph-like visualization of chunks and relationships.

Summary of Commands

Below is a concise summary of each `pwndbg` heap command:

- `arena` – Print current arena's contents.
- `arenas` – List all arenas in the current process.
- `bins` – Print all bins and thread-local tcache.
- `fastbins` – Display the fastbins (small freed chunks).
- `find-fake-fast` – Search for fake fast chunks overlapping an address.
- `heap` – Print all heap chunks from base to top.
- `heap-config` – Show heap inspection configuration settings.
- `hi` – Check if an address belongs to a heap chunk.
- `largebins` – Print the largebins used for large chunks.
- `malloc-chunk` – Show metadata of a heap chunk at a specific address.
- `mp` – Print the `mp_` struct (malloc tunable parameters).
- `smallbins` – Show contents of the smallbins.
- `tcache` – Print the thread-local caching system.
- `tcachebins` – Show contents of tcache bins.
- `top-chunk` – Display the top chunk (end of heap).
- `try-free` – Simulate a `free()` on an address.
- `unsortedbin` – Print the contents of the unsorted bin.
- `vis-heap-chunks` – Graphically visualize heap chunks and relationships.

7 jemalloc Heap Commands in `pwndbg`

Here is a clear explanation of the three `pwndbg` commands for inspecting **jemalloc** heap internals.

7.1 `jemalloc-extent-info`

Description:
Displays detailed information about the jemalloc *extent* (a contiguous chunk of memory managed by jemalloc) that contains a given address.

Use case:

- Size of the extent
- Allocation flags
- Whether it's committed or dirty
- Associated arena or region ID

Example:

```
pwndbg> jemalloc-extent-info 0x555555758000
```

7.2 `jemalloc-find-extent`

Description:
Finds and returns the *extent* structure associated with a jemalloc-allocated pointer. Useful for tracing an allocation back to its metadata.

Use case:
Ideal for reverse-engineering or debugging jemalloc-managed binaries to understand memory layout and allocation patterns.

Example:

```
pwndbg> jemalloc-find-extent 0x602000
```

7.3 `jemalloc-heap`

Description:
Prints information about all known extents currently allocated by jemalloc in the process.

Use case:
Provides a snapshot of the entire jemalloc heap — useful for heap analysis, forensic debugging, and identifying memory usage patterns.

Example:

```
pwndbg> jemalloc-heap
```

8 Breakpoint Commands

Here's a detailed explanation of each of the **pwndbg** breakpoint-related commands, commonly used in GDB with the Pwndbg enhancement for exploit development and binary debugging.

8.1 `break-if-taken`

Description: Sets a breakpoint on a branch instruction **only if the branch is taken** during execution.

Use case: Suppose you want to break only when a conditional jump (like `jne`, `je`, `jg`, etc.) is actually taken. This helps focus on a specific path in code flow.

Example:

```
break-if-taken
```

Pwndbg identifies the next branch instruction at the current execution point and sets a conditional breakpoint that will only trigger if the branch is taken.

8.2 `break-if-not-taken`

Description: Breaks only if the branch instruction is **not taken** (i.e., the condition is false and execution continues sequentially).

Use case: Useful for debugging when you expect a branch to be taken and want to catch the case when it isn't (e.g., failed condition or incorrect input).

8.3 `breakrva [brva]` (alias: `brva`)

Description: Break at a **Relative Virtual Address (RVA)** offset from the base of a **Position-Independent Executable (PIE)** binary.

Use case: For PIE binaries (which are loaded at random addresses), absolute addresses vary at runtime. This command allows you to specify a breakpoint relative to the base address of the binary.

Example:

```
breakrva 0x1234
```

This sets a breakpoint at the address `base_address + 0x1234`.

8.4 `ignore`

Description: Tells GDB to ignore a breakpoint a certain number of times before stopping.

Syntax:

```
ignore <breakpoint-number> <count>
```

Use case: If a breakpoint is hit often but you only want it to stop the program after a certain number of hits, you can use this.

Example:

```
ignore 2 5
```

This tells GDB to ignore breakpoint #2 for the first 5 times it's hit, and only break on the 6th time.

These commands are particularly useful in reverse engineering, CTFs, or exploit development where controlling exactly when and how a program breaks is crucial.

9 Memory Commands

Here's a detailed explanation of each **Pwndbg memory command**, including usage, purpose, and examples. These commands are invaluable for memory analysis, reverse engineering, and vulnerability research during GDB sessions with Pwndbg.

9.1 distance

Description: Prints the distance between two addresses, or the offset of an address from the start of its page (page base).

Example:

```
distance 0x7ffff7dd18c0 0x7ffff7dd1000
distance 0x7ffff7dd18c0
```

The first shows the byte difference, the second gives the offset from page base.

9.2 gdt

Description: Decodes the Global Descriptor Table (GDT) entries in x86-64 at a given address.

Example:

```
gdt 0x560000
```

9.3 go-dump (alias: god)

Description: Dumps a Go language value of a given type from a specified memory address.

Example:

```
go-dump 0x55555575e000 *main.SomeStruct
```

9.4 go-type (alias: goty)

Description: Prints the Go runtime reflection type at a specified address.

Example:

```
go-type 0x55555575f000
```

9.5 hexdump

Description: Prints a hex dump of memory from a given address or module.

Example:

```
hexdump 0x601000 64
hexdump main
```

9.6 leakfind

Description: Tries to find a leak chain (pointer path) from one address to another.

Example:

```
leakfind 0x602020
```

9.7 memfrob

Description: Applies XOR with '*' (0x2A) to a memory region.
Example:

```
memfrob 0x601000 32
```

9.8 mmap

Description: Calls the `mmap()` syscall and prints the result.
Examples:

```
mmap
mmap size=0x1000 prot=7 flags=0x22
```

9.9 mprotect

Description: Calls the `mprotect()` syscall on a memory region.
Example:

```
mprotect 0x601000 0x1000 7
```

9.10 p2p

Description: Performs a pointer-to-pointer chain search.
Example:

```
p2p libc heap
```

9.11 probeleak

Description: Scans memory for possible offset leaks.
Example:

```
probeleak 0x601000
```

9.12 search

Description: Searches memory for strings, byte patterns, or pointers.
Examples:

```
search "password"
search 0xdeadbeef
```

9.13 telescope

Description: Recursively dereferences memory starting at an address.
Example:

```
telescope \$rsp
```

9.14 vmmap (aliases: lm, address, vprot, libs)

Description: Shows the virtual memory map of the current process.
Example:

```
vmmap
```

9.15 vmmap-add

Description: Manually adds a memory mapping to cache.
Example:

```
vmmap-add 0xdead0000 0x1000 rwx
```

9.16 vmmap-clear

Description: Clears the memory map cache.
Example:

```
vmmap-clear
```

9.17 vmmap-explore

Description: Explores a memory page to guess content type.
Example:

```
vmmap-explore 0x601000
```

9.18 xinfo

Description: Shows detailed information and offsets of an address.
Example:

```
xinfo 0x7ffff7dd18c0
```

Outputs:

- Module location
- Offset from module base
- Stack/heap/libc segment info

9.19 xor

Description: XORs a region of memory with a specific key.
Example:

```
xor 0x601000 16 0x41
```

These tools make Pwndbg a powerful environment for reverse engineers and CTF participants by automating memory introspection, leak analysis, and debugging tasks.

10 Stack Commands

Here's a detailed explanation of each **Pwndbg stack-related command**, including descriptions, practical use cases, and examples. These are particularly helpful for analyzing control flow, detecting stack corruption, and inspecting function frames during binary exploitation and reverse engineering.

10.1 canary

Description:
Prints out the current **stack canary** value for the program being debugged. Stack canaries (a.k.a. stack cookies) are used as a security feature to detect buffer overflows before the return address is overwritten.

Use Case:
When debugging a binary with stack protections (e.g., compiled with `-fstack-protector`), you might want to inspect the current value of the canary to ensure it's intact or understand when it's been tampered with.

Example:

```
canary
```

Output Example:

```
Stack Canary: 0x00dff2b7a400dc00
```

If the value changes unexpectedly or is zeroed out, it may indicate a buffer overflow has occurred.

10.2 retaddr

Description:
Prints the **return addresses** stored on the stack. These are critical for understanding control flow and potential locations for return-oriented programming (ROP) attacks.

Use Case:

- Identifying where functions will return to.
- Setting breakpoints on return addresses.
- Finding potential targets for hijacking execution flow.

Example:

```
retaddr
```

Output Example:

```
Found return address at: 0x7fffffffded8 --> 0x4006f3 (main+0x33)
```

10.3 stack

Description:
Dumps and dereferences a portion of the stack with a specified count and offset. It provides a snapshot of the stack from a relative position.

Syntax:

```
stack [count] [offset]
```

Example:

```
stack 10 0
```

This prints 10 entries from the current stack pointer (`$rsp`), starting at offset 0.
Use Case:

- Manually inspecting local variables, saved registers, or function arguments on the stack.
- Tracing pointer dereferences.

10.4 stack-explore

Description:
Explores the stack for **all threads** in the current process, giving a multithreaded overview of stack contents.
Use Case:
In multithreaded programs, especially those involving concurrency bugs, being able to inspect each thread's stack is very useful.
Example:

```
stack-explore
```

Output:
You'll get a list of stack pointers and values for each thread.

10.5 stackf

Description:
Similar to `stack`, but provides **full function stack frame** context. It prints the stack content with annotations and helps reconstruct a function's frame.
Syntax:

```
stackf [count] [offset]
```

Example:

```
stackf 20 0
```

Prints 20 entries starting from `$rsp`, including return addresses, saved registers, function arguments, etc.
Use Case:

- Visualizing the entire layout of a stack frame.
- Understanding the calling conventions and saved state.
- Detecting stack corruption.

These commands are essential in exploitation and forensic debugging, where precise insight into the stack's layout and integrity can reveal vulnerabilities or help manipulate control flow.

11 Register Commands

Here's a detailed explanation of each **Pwndbg register-related command**, including descriptions, practical use cases, and examples. These are particularly useful for examining and modifying low-level processor state during debugging, binary exploitation, or architecture-specific reverse engineering.

11.1 `cpsr` (aliases: `xpsr`, `pstate`)

Description:
Prints out the **Current Program Status Register (CPSR)** on ARM or **xPSR/PSTATE** on ARM Cortex-M or ARM64 platforms. This register holds processor flags such as Zero (Z), Negative (N), Carry (C), and Overflow (V), as well as processor mode and interrupt state.

Use Case:

- Checking condition flags after arithmetic operations (e.g., Z for zero result).
- Verifying if interrupts are disabled.
- Understanding execution state (Thumb/ARM mode, etc.).

Example:

```
cpsr
```

Output Example:

```
CPSR = 0x200001f3 [N Z C V Q J GE E A I F T M=System]
```

This shows the current flag state and mode in a human-readable form.

11.2 `fsbase`

Description:
Prints the **FS segment base address** on x86 and x86_64 systems. This is often used for accessing thread-local storage (TLS) or segment-based data like TCB (Thread Control Block) in Linux.

Use Case:

- Inspecting thread-local storage location.
- Analyzing segmentation-based access (especially in glibc or for TLS variables).

Example:

```
fsbase
```

Output Example:

```
FS base: 0x7ffff7fc2700
```

Note: You can also use the GDB convenience variable `$fsbase`.

11.3 gsbase

Description:
Prints the **GS segment base address**, another segment used for thread-local storage, particularly on 64-bit Linux systems.

Use Case:

- Especially useful in glibc-based binaries where `gs:[0]` points to the TLS/TCB.
- Relevant for programs that use `__thread` variables.

Example:

```
gsbase
```

Output Example:

```
GS base: 0x7ffff7fe7000
```

Note: `$gsbase` is also available as a convenience variable in GDB.

11.4 setflag

Description:
Manually sets or clears individual flags in the **FLAGS (x86)** or **CPSR/xPSR (ARM)** register.

Use Case:

- Modifying execution behavior (e.g., forcing a condition to be true).
- Simulating a CPU state (e.g., set Zero flag to force a conditional jump).
- Testing flag-sensitive code paths.

Example:

```
setflag ZF
```

This sets the **Zero Flag (ZF)** in the EFLAGS register.

```
setflag -ZF
```

This clears the Zero Flag.

These commands are especially useful when working close to the CPU level, analyzing processor state changes after instructions, or emulating specific conditions during exploit development or hardware debugging.

12 Process Commands

Here's a detailed explanation of the **Pwndbg Process-related commands**, complete with usage, descriptions, and examples. These commands are helpful when inspecting or managing process-level state in debugging, particularly in multi-threaded or multi-process environments.

12.1 killthreads

Description:
Terminates all threads in the current process except the main thread, or terminates specific threads if provided.

Use Cases:

- Simplify debugging by removing concurrency.
- Focus analysis on the main thread when others aren't relevant.
- Stop problematic threads that are interfering with inspection or crashing the process.

Syntax:

```
killthreads [<thread-id>...]
```

Example 1: Kill all threads except the main one.

```
killthreads
```

Example 2: Kill a specific thread by its ID.

```
killthreads 2
```

This is especially useful in multi-threaded applications like network servers, where background threads may obscure debugging of the main control flow.

12.2 pid (alias: getpid)

Description:
Displays the process ID (PID) of the running program being debugged.

Use Cases:

- Useful when attaching external tools (e.g., `strace`, `gcore`, `perf`) to the current process.
- For scripting automation that requires referencing the active debugged process.

Example:

```
pid
```

Output Example:

```
Process PID: 4312
```

This can be especially important when dealing with forked or spawned processes.

12.3 procinfo

Description:
Displays various runtime details about the current process, such as:

- PID
- Command-line arguments
- Current working directory
- Environment variables
- Thread count

Use Cases:

- Get a snapshot of the runtime environment.
- Identify key characteristics of the running binary and its state.
- Debug programs that rely on specific environment variables or arguments.

Example:

```
procinfo
```

Output Example (partial):

```
PID: 4312
CWD: /home/user/project
Executable: ./vulnerable
Threads: 4
Arguments: ['./vulnerable', 'arg1']
Environment: {'PATH': '/usr/bin', ...}
```

This information is useful when verifying the runtime context matches what's expected or debugging issues that may stem from a misconfigured execution environment.

These commands give you low-level insight and control over the debugged process, especially when dealing with concurrency, runtime environments, and external tools.

13 Linux/libc/ELF Commands

Here's a detailed explanation of each **Pwndbg Linux/libc/ELF-related command**, complete with practical examples, use cases, and descriptions. These commands are especially valuable when analyzing dynamic linking, environment setup, memory layout, and ELF internals during binary exploitation or reverse engineering.

13.1 argc

Description: Prints the number of arguments passed to the program.

Use Case: Helpful when analyzing argument-based vulnerabilities or debugging programs with varying input lengths.

Example:

```
argc
```

Output Example:

```
argc = 3
```

13.2 argv

Description: Displays the contents of `argv[]`, i.e., command-line arguments.

Use Case: Inspect user-supplied input, verify argument parsing logic.

Example:

```
argv
```

Output Example:

```
argv[0]: ./vuln
argv[1]: test
argv[2]: input
```

13.3 aslr

Description: Checks or toggles the current ASLR status.

Use Case: Control memory layout reproducibility.

Example:

```
aslr
aslr off
```

Output:

```
ASLR is currently enabled.
```

13.4 auxv and auxv-explore

Description: Print or explore information from the Auxiliary ELF Vector.
Use Case: Identify page size, system values, random seeds, etc.
Example:

```
auxv
auxv-explore
```

13.5 elfsections

Description: Print ELF section mappings.
Use Case: Analyze layout of sections like `.text`, `.data`, etc.
Example:

```
elfsections
```

13.6 envp [env, environ]

Description: Prints environment variables passed to the program.
Example:

```
envp
```

Output Example:

```
PATH=/usr/bin:/bin
LD_PRELOAD=./evil.so
```

13.7 errno

Description: Converts `errno` to string representation.
Example:

```
errno
errno 13
```

Output Example:

```
errno (2): No such file or directory
```

13.8 got and gotplt

Description: Displays state of the GOT or GOT/PLT section.
Use Case: Analyze dynamic linking and GOT overwrite scenarios.
Example:

```
got
gotplt
```

13.9 libcinfo

Description: Show the loaded `libc` version and source link.
Example:

```
libcinfo
```

Output Example:

```
libc version: 2.31
Build ID: xyz123
Source: https://sourceware.org/...
```

13.10 linkmap

Description: Displays the link map for shared library loading.
Example:

```
linkmap
```

13.11 onegadget

Description: Lists one-gadget RCE payloads in libc.
Example:

```
onegadget
```

13.12 piebase

Description: Converts RVA to virtual address using PIE base.
Example:

```
piebase 0x1234
```

Output:

```
PIE base: 0x555555554000 + 0x1234 = 0x555555555234
```

13.13 plt

Description: Prints PLT function entries.
Example:

```
plt
```

13.14 strings

Description: Extracts readable ASCII strings from memory.
Example:

```
strings
```

13.15 `threads`

Description: Lists all threads of the debugged process.
Example:

```
threads
```

13.16 `tls`

Description: Prints the base address of TLS.
Example:

```
tls
```

13.17 `track-got` and `track-heap`

Description: Enables/disables GOT or heap tracking in Pwndbg.
Example:

```
track-got on
track-heap on
```

14 Disassemble Commands

Here's a detailed explanation of the **Pwndbg Disassemble-related commands**, which are crucial for examining and understanding instruction-level execution during debugging, reverse engineering, or exploit development.

14.1 `emulate`

Description:
The `emulate` command functions similarly to `nearpc` but goes one step further by **emulating** instructions starting from the current program counter (`$pc`). It shows how each instruction affects registers and memory in a step-by-step manner.

Use Cases:

- Understand instruction effects without executing them.
- Analyze complex logic statically but with insight into register state changes.
- Debug self-modifying or obfuscated code by emulating how it executes.

Example:

```
emulate
```

Sample Output:

```
0x555555554000:  mov eax, 0x1
=> eax = 0x1
0x555555554005:  add eax, 0x2
=> eax = 0x3
```

This output shows each instruction and how it changes CPU state, like register values. It's extremely helpful for low-level debugging and understanding exactly what each instruction does.

14.2 `nearpc` (aliases: `pdisass`, `u`)

Description:
The `nearpc` command **disassembles instructions near the current program counter (`$pc`)**. It gives a static view of the surrounding code, typically centered on where the execution currently is.

Use Cases:

- Quickly view code around the current execution point.
- Spot function prologues/epilogues or nearby gadgets.
- Confirm shellcode or injected code is correct and executable.

Example:

```
nearpc
```

Sample Output:

```
0x555555554000:  push   rbp
0x555555554001:  mov    rbp, rsp
=> 0x555555554004:  sub    rsp, 0x10
0x555555554008:  mov    DWORD PTR [rbp-0x4], 0x0
```

You can also pass an address to inspect arbitrary code locations:

```
nearpc 0x555555554050
```

Summary

- `emulate`: Shows the effect of instructions from `$pc` on the CPU state (e.g., registers) without actually executing them. Useful for understanding how instructions modify the state step-by-step.
- `nearpc` (aliases: `pdisass`, `u`): Provides a static disassembly view around the current instruction pointer or a specified address. Helps in analyzing surrounding machine code for debugging or exploitation.

15 Misc Commands

Here's a detailed explanation of each **Pwndbg Miscellaneous Command**, complete with descriptions, use cases, and examples to help understand their purpose during debugging, exploitation, and reverse engineering tasks.

15.1 asm

Description: Assembles a given instruction or shellcode into machine code bytes.
Use Case: Quickly turn instructions into opcodes for shellcode or patching.
Example:

```
asm "jmp eax"
\end{verbatim}
\textbf{Output:}
\begin{verbatim}
\xff\xe0
```

15.2 checksec

Description: Displays the binary's security features using `checksec` (e.g., PIE, RELRO, NX).
Use Case: Identify potential exploit mitigations.
Example:

```
checksec
```

Output:

```
RELRO           STACK CANARY      NX            PIE
Partial RELRO   No canary found   NX enabled    No PIE
```

15.3 comm

Description: Add custom comments to addresses in disassembly view.
Use Case: Annotate findings while reversing.
Example:

```
comm 0x555555554000 "Entry point"
```

15.4 cyclic

Description: Create or search a cyclic pattern.
Use Case: Determine the offset for control over RIP/EIP.
Example:

```
cyclic 100
cyclic -l 0x6161616c
```

15.5 cymbol

Description: Manage user-defined structures in plain C.
Use Case: Load and use custom C structs in memory analysis.
Example:

```
cymbol add mystruct "struct mystruct { int a; char b; };"
```

15.6 down

Description: Move one stack frame down (to the callee).
Use Case: Navigate call stack during backtracing.
Example:

```
down
```

15.7 dt

Description: Dump type information (e.g., `ucontext_t`, structs).
Use Case: Inspect layout and fields of structs.
Example:

```
dt ucontext_t
```

15.8 dumpargs (alias: args)

Description: Show function call arguments at the current instruction.
Use Case: Understand what values are being passed in a call.
Example:

```
dumpargs
```

15.9 getfile

Description: Show the file being debugged.
Use Case: Confirm target binary.
Example:

```
getfile
```

15.10 hex2ptr

Description: Convert hex string to little-endian pointer.
Use Case: Decode shellcode arguments or byte strings.
Example:

```
hex2ptr 41 42 43 44
```

Output:

```
0x44434241
```

15.11 hijack-fd

Description: Replace a file descriptor in the running process.
Use Case: Redirect input/output streams (e.g., hijack stdin/out).
Example:

```
hijack-fd 1 /tmp/output.txt
```

15.12 ipi

Description: Opens an interactive IPython shell.
Use Case: Advanced Python scripting and debugging.
Example:

```
ipi
```

15.13 patch

Description: Patch bytes or instructions in memory.
Use Case: Modify instructions or bypass checks.
Example:

```
patch 0x555555554000 "nop"
```

15.14 patch-list

Description: List all patches made during the session.
Use Case: Track binary modifications.
Example:

```
patch-list
```

15.15 patch-revert

Description: Revert a patch at a given address.
Use Case: Undo previous modifications.
Example:

```
patch-revert 0x555555554000
```

15.16 plist

Description: Dump elements of a linked list.
Use Case: Traverse and analyze in-memory linked lists.
Example:

```
plist 0x602010 next
```

15.17 sigreturn

Description: Show `SigreturnFrame` at a specified address.
Use Case: Analyze or debug SROP (Sigreturn-Oriented Programming).
Example:

```
sigreturn 0xdeadbeef
```

15.18 spray

Description: Fill memory with a cyclic pattern.
Use Case: Identify buffer overflows or corrupted memory.
Example:

```
spray 0x601000 64
```

15.19 tips

Description: Display helpful Pwndbg tips.
Use Case: Discover features or shortcuts.
Example:

```
tips
```

15.20 up

Description: Move one stack frame up (to the caller).
Use Case: Analyze calling function context.
Example:

```
up
```

15.21 valist

Description: Show contents of a `va_list`.
Use Case: Inspect variadic function arguments.
Example:

```
valist
```

15.22 vmmap-load

Description: Load memory maps from an ELF file into GDB.
Use Case: Simulate memory layout of a binary without running it.
Example:

```
vmmap-load ./vuln
```

16 Kernel Commands

Here's a detailed explanation of each **Pwndbg Kernel Command**, including use cases and examples to help you understand their role in kernel-level debugging or analysis (especially useful in Android, embedded, and exploit dev contexts):

16.1 `binder`

Description: Displays Android Binder driver information, a critical IPC mechanism in Android systems.
Use Case: Debug or reverse-engineer Android kernel and userland interactions.
Example:

```
binder
```

16.2 `kbase`

Description: Finds the base address of the kernel in memory.
Use Case: Helps in kernel address space layout analysis, useful for KASLR bypass.
Example:

```
kbase
```

Output:

```
Kernel base: 0xffffffff81000000
```

16.3 `kchecksec`

Description: Checks kernel security configurations (like KASLR, SMEP, etc.).
Use Case: Determine the hardening features enabled in the kernel.
Example:

```
kchecksec
```

16.4 `kcmdline`

Description: Displays the contents of `/proc/cmdline`, which holds kernel boot parameters.
Use Case: Inspect kernel flags that may weaken security or change behavior.
Example:

```
kcmdline
```

16.5 `kconfig`

Description: Prints out the kernel config (if $\texttt{CONFIG}_{I}KCONFIGisenabled$).
`Use Case: Verify build-time configuration options like module support, debugging flags, etc.`
`Example:`

```
kconfig
```

16.6 klookup

Description: Resolves kernel symbol names to addresses.
Use Case: Identify the location of functions and structures in kernel memory.
Example:

```
klookup sys_call_table
```

16.7 knft-dump

Description: Dumps all nftables components: tables, chains, rules, and expressions.
Use Case: Audit or reverse-engineer Linux netfilter firewall configurations.
Example:

```
knft-dump
```

16.8 knft-list-chains

Description: Lists chains in a given nftables table.
Use Case: Inspect filtering logic of a specific firewall table.
Example:

```
knft-list-chains filter
```

16.9 knft-list-exprs

Description: Shows expressions (matching conditions or actions) in a specific rule.
Use Case: Analyze complex rule logic in netfilter.
Example:

```
knft-list-exprs filter input 0
```

16.10 knft-list-flowtables

Description: Lists flowtables in a specific table.
Use Case: Inspect connection tracking rules and policies.
Example:

```
knft-list-flowtables inet
```

16.11 knft-list-objects

Description: Dumps objects (e.g., counters, quotas) from a specific nftables table.
Use Case: Track metrics and usage stats defined in firewall rules.
Example:

```
knft-list-objects filter
```

16.12 knft-list-rules

Description: Lists rules from a specific chain.
Use Case: Full inspection of rule logic and order.
Example:

```
knft-list-rules filter input
```

16.13 knft-list-sets

Description: Dumps all sets (collections of IPs, ports, etc.) from a given table.
Use Case: View whitelists/blacklists or dynamic match sets.
Example:

```
knft-list-sets filter
```

16.14 knft-list-tables

Description: Lists all nftables tables, optionally by network namespace.
Use Case: Get overview of all firewall structures.
Example:

```
knft-list-tables
```

16.15 kversion

Description: Prints the kernel version string from /proc/version.
Use Case: Confirm kernel build version, toolchain, and compiler used.
Example:

```
kversion
```

16.16 pcplist

Description: Displays per-CPU memory page lists (used in slab/page allocation).
Use Case: Low-level memory allocator debugging or optimization.
Example:

```
pcplist
```

16.17 slab

Description: Shows information about the slab allocator and caches.
Use Case: Memory profiling or hunting for use-after-free vulnerabilities.
Example:

```
slab
```

17 Integration Commands

Here's a detailed explanation of each Pwndbg Integration Command, including use cases and examples. These commands are designed to integrate Pwndbg with powerful reverse engineering tools such as IDA, Binary Ninja, Ghidra, radare2, and Rizin, as well as assist with decompilation and ROP gadget discovery.

17.1 ai

Description: Ask GPT-3 a question about the current debugging context.
Use Case: Use AI assistance to understand disassembly, stack frames, or suggest exploitation techniques.
Example:

```
ai what does this function do?
```

17.2 bn-sync [bns]

Description: Synchronize Binary Ninja's cursor with the current GDB address.
Use Case: Bridge static analysis and runtime debugging for efficient binary exploration in Binary Ninja.
Example:

```
bn-sync
```

17.3 decomp

Description: Use the currently selected decompiler integration (e.g., IDA, Ghidra, Binary Ninja) to decompile code near the current instruction pointer.
Use Case: Quickly obtain a higher-level view of assembly code during dynamic analysis.
Example:

```
decomp
```

17.4 ghidra

Description: Decompile a given function using Ghidra.
Use Case: If you've set up GhidraBridge, this lets you get readable pseudocode of functions during debugging.
Example:

```
ghidra main
```

17.5 j

Description: Synchronize IDA Pro's cursor with the current GDB address.
Use Case: Keep dynamic analysis in GDB aligned with static analysis in IDA Pro.
Example:

```
j
```

17.6 r2 [radare2]

Description: Launch radare2 for the current binary.
Use Case: Start reverse engineering in radare2 directly from Pwndbg.
Example:

```
r2
```

17.7 r2pipe

Description: Send stateful radare2 commands using the r2pipe interface.
Use Case: Interactively use radare2's scripting API while debugging, without reopening the binary.
Example:

```
r2pipe pd 10
```

17.8 rop [ropgadget]

Description: Use ROPgadget to find Return-Oriented Programming gadgets.
Use Case: For exploit development where you need to find gadgets for building a ROP chain.
Example:

```
rop
```

17.9 ropper

Description: Find ROP gadgets using the ropper tool (alternative to ROPgadget).
Use Case: Useful for crafting exploits; some users prefer its output and filtering features.
Example:

```
ropper
```

17.10 rz [rizin]

Description: Launch Rizin (a fork of radare2) for the current binary.
Use Case: Begin reverse engineering using Rizin from within a GDB debugging session.
Example:

```
rz
```

17.11 rzpipe

Description: Execute commands in Rizin using rzpipe (Rizin's equivalent to r2pipe).
Use Case: Interact with Rizin's analysis engine during runtime debugging.
Example:

```
rzpipe afl
```

17.12 save-ida

Description: Saves the current IDA database state from Pwndbg.
Use Case: Preserve progress in IDA Pro while switching between static and dynamic analysis.
Example:

```
save-ida
```

18 WinDbg Commands

Here is a detailed explanation of each Pwndbg WinDbg-Compatible Command, including use cases and examples. These commands replicate familiar WinDbg syntax and behavior for users transitioning from Windows kernel/userland debugging to pwndbg on Linux.

18.1 bc

Description: Clear (remove) the breakpoint with the specified index.
Use Case: Remove a previously set breakpoint using its index from the breakpoint list.
Example:

```
bc 0
```

18.2 bd

Description: Disable the breakpoint with the specified index.
Use Case: Temporarily ignore a breakpoint without deleting it.
Example:

```
bd 1
```

18.3 be

Description: Enable a previously disabled breakpoint.
Use Case: Re-activate a disabled breakpoint.
Example:

```
be 1
```

18.4 bl

Description: List all currently defined breakpoints along with their indices and states.
Use Case: Check which breakpoints are active, disabled, or hit.
Example:

```
bl
```

18.5 bp

Description: Set a breakpoint at the specified address.
Use Case: Intercept execution at a particular instruction for inspection.
Example:

```
bp *0x8048500
```

18.6 da / ds

Description: Dump a string from the specified memory address.
Use Case: Print readable ASCII string content from memory.
Example:

```
da 0x601050
```

18.7 db

Description: Dump raw bytes starting at a specific memory address.
Use Case: Examine memory at byte granularity.
Example:

```
db 0x601000 L10
```

18.8 dc

Description: Hexdump memory starting at a specified address.
Use Case: Get a quick, color-formatted dump of memory for inspection.
Example:

```
dc 0x601000
```

18.9 dd

Description: Dump 32-bit doublewords from memory.
Use Case: Useful for inspecting dword-aligned data (e.g., function pointers, integers).
Example:

```
dd 0x601000 L8
```

18.10 dds (aliases: kd, dps, dqs)

Description: Dump pointers and attempt to resolve symbols from those addresses.
Use Case: Useful for reverse-engineering pointer tables, vtables, or symbol-rich memory regions.
Example:

```
dds 0x601000
```

18.11 dq

Description: Dump 64-bit qwords (quadwords) from memory.
Use Case: Inspect qword-aligned data structures like 64-bit pointers or large integers.
Example:

```
dq 0x601000 L4
```

18.12 dw

Description: Dump 16-bit words from memory.
Use Case: Use for checking halfword-aligned data or narrow integer fields.
Example:

```
dw 0x601000 L8
```

18.13 eb

Description: Write a byte value to the specified memory address.
Use Case: Patch memory (e.g., NOP a function) during runtime.
Example:

```
eb 0x601000 0x90
```

18.14 ed

Description: Write a 32-bit dword to a memory location.
Use Case: Modify global variables or function pointers dynamically.
Example:

```
ed 0x601000 0xdeadbeef
```

18.15 eq

Description: Write a 64-bit qword to a memory location.
Use Case: Alter 64-bit data like addresses or large integers in memory.
Example:

```
eq 0x601000 0x4141414142424242
```

18.16 ew

Description: Write a 16-bit word to memory.
Use Case: Smaller-scale memory patching.
Example:

```
ew 0x601000 0x9090
```

18.17 ez / eza

Description: Write an ASCII string to memory at the given address.
Use Case: Inject strings into memory for use in format string exploits, shellcode, etc.
Example:

```
ez 0x601000 "exploit"
```

18.18 go

```
Description:  Alias for the continue command.
   Use Case:  Continue program execution after a breakpoint or signal.
   Example:
```

```
go
```

18.19 k (alias: bt)

```
Description:  Print a stack backtrace.
   Use Case:  Identify call history and crash origin.
   Example:
```

```
k
```

18.20 ln

```
Description:  List symbols nearest to the given address.
   Use Case:  Find out which function or variable an address belongs to.
   Example:
```

```
ln 0x8048500
```

18.21 pc

```
Description:  Alias for the nextcall command (i.e., step over a function call).
   Use Case:  Used in step-wise debugging to move over function calls without diving into them.
   Example:
```

```
pc
```

18.22 peb

```
Description:  Not functional on Linux | placeholder from WinDbg.
   Use Case:  No use in current Linux Pwndbg context.
```

19 Pwndbg Commands

Here is a detailed explanation of each Pwndbg Meta/Utility Command, with examples and use cases.
These commands primarily manage pwndbg itself---its configuration, state, and development tools---rathe
than interacting directly with the debugged program.

19.1 bugreport

Description:
Generates a detailed bug report, including system info and configuration.
Use Case:
Used when reporting issues to Pwndbg maintainers|it includes your system setup, Python version,
GDB version, etc.
Example:

```
bugreport
```

19.2 config

Description:
Displays or sets Pwndbg configuration options.
Use Case:
Inspect or modify runtime settings like context-sections, display-assembly, etc.
Example:

```
config
config context-sections code,regs
```

19.3 configfile

Description:
Generates a .gdbinit-style config file reflecting your current Pwndbg config.
Use Case:
Save your customized configuration for reuse or sharing.
Example:

```
configfile > ~/.pwndbg-config
```

19.4 memoize

Description:
Toggles memoization (function result caching) for Pwndbg internals.
Use Case:
Disable memoization to troubleshoot performance issues or caching-related bugs.
Example:

```
memoize
```

19.5 profiler

Description:
Provides subcommands for profiling the performance of Pwndbg itself.
Use Case:
Identify bottlenecks in custom commands or internal behaviors when extending or debugging Pwndbg.
Example:

```
profiler enable
profiler report
```

19.6 pwndbg

Description:
Lists all available Pwndbg commands.
Use Case:
Quickly browse through all integrated Pwndbg features and commands.
Example:

```
pwndbg
```

19.7 reinit-pwndbg

Description:
Forces Pwndbg to reinitialize all its internal state.
Use Case:
Useful after modifying source code or configuration, without restarting GDB.
Example:

```
reinit-pwndbg
```

19.8 reload

Description:
Reloads Pwndbg from source.
Use Case:
Use when developing or after pulling new changes to reload the module into GDB without restart.
Example:

```
reload
```

19.9 theme

Description:
Displays current theme options or changes theme-related settings (like colors and symbols).
Use Case:
Customize the appearance of UI elements in your debugging view.
Example:

```
theme
theme context-title-style bold
```

19.10 themefile

Description:
Generates a config file that captures your current Pwndbg theme settings.
Use Case:
Export your visual preferences to keep them consistent across machines or share with others.
Example:

```
themefile > ~/.pwndbg-theme
```

19.11 version

Description:
Prints out the version of Pwndbg and its dependencies (GDB, Python, Capstone, Unicorn, etc.).
Use Case:
Used for bug reports or ensuring compatibility with other tools or exploits.
Example:

```
version
```

20 Pwndbg Developer Commands

Here's a detailed explanation of each Pwndbg Developer Command, including descriptions, use cases, and practical examples. These are mainly used for debugging or customizing the behavior of Pwndbg itself during development or advanced usage.

20.1 dev-dump-instruction

Description:
Dumps the internal attributes of a disassembled instruction as interpreted by Pwndbg.
Use Case:
Useful when developing Pwndbg or analyzing how Pwndbg parses and processes machine instructions. It helps reveal details such as the instruction mnemonic, operands, size, and memory references as Pwndbg sees them internally.
Example:

```
dev-dump-instruction
```

This command dumps detailed information about the current instruction at $pc (the program counter). Example output may include:

- address: the memory address of the instruction
- mnemonic: e.g., mov, call, jmp
- operands: source/destination registers or values
- size: instruction length in bytes
- is_call, is_jump, is_ret: boolean flags for control-flow types

20.2 log-level

Description:
Sets the verbosity level of internal logging messages in Pwndbg.
Use Case:
Control the amount of debug or informational output shown by Pwndbg. Helpful for developers who want to trace command execution, error handling, or internal decision-making.
Example:

```
log-level debug
```

Available levels usually include (from most to least verbose):

- debug
- info
- warning
- error
- critical

Setting it to debug enables full internal logging, which is ideal during feature development or troubleshooting.