

Gef 使用手册

Gef 使用手册

aliases 命令

 创建/删除 简写命令(别名)

 加入 PEDA 或 WinDBG的喜好设定

aslr 命令

assemble 命令

canary 命令

capstone-disassemble 命令

checksec 命令

config 命令

context 命令

 编辑上下文布局

 将上下文输出重定向到另一个 TTY/file

 举例

dereference 命令

edit-flags 命令

elf-info 命令

entry-break 命令

\$(eval) 命令

format-string-helper 命令

functions 命令

gef-remote 命令

 使用本地副本

 没有本地副本

 QEMU用户模式

heap 命令

 heap chunks 命令

 heap chunk 命令

 heap arenas 命令

 heap set-arena 命令

 heap bins 命令

 heap bins fast 命令

 其他的 heap bins x 命令

heap-analysis-helper 命令

help 命令

hexdump 命令

hijack-fd 命令

ida-interact 命令

ksymaddr 命令

memory 命令

 添加一个查看

 移除一个查看

 列出所有查看

 清楚所有查看

nop 命令

patch 命令

pattern 命令

创建
查找
pcustom 命令
 相关配置
 使用用户定义的结构体
pie 命令
 pie breakpoint 命令
 pie info 命令
 pie delete 命令
 pie attach 命令
 pie remote 命令
 pie run 命令
print-format 命令
process-search 命令
process-status 命令
registers 命令
reset-cache 命令
ropper 命令
scan 命令
search-pattern 命令
set-permission 命令
shellcode 命令
stub 命令
 例子
theme 命令
 改变颜色
tmux-setup 命令
trace-run 命令
unicorn-emulate 命令
vmmap 命令
xfiles 命令
xinfo 命令
xor-memory 命令
版权信息
PDF下载地址

aliases 命令

列举出所有的简写命令(别名)

```
gef> aliases
[+] Aliases defined:
fmtstr-helper      → format-string-helper
telescope          → dereference
dps                → dereference
dq                 → hexdump qword
dd                 → hexdump dword
dw                 → hexdump word
dc                 → hexdump byte
cs-dis             → capstone-disassemble
ctx                → context
```

```
start-break      → entry-break
ps               → process-search
[...]
```

创建/删除 简写命令(别名)

GEF定义了自己的别名机制，该机制覆盖了GDB提供的传统别名。

用户可以通过编辑位于 `~/.gef.rc` 的GEF配置文件来创建/修改/删除别名。

别名必须位于配置文件的“aliases”部分中。

创建新别名就像在本节中创建新条目一样简单：

```
$ nano ~/.gef.rc
[...]
[aliases]
my-new-alias = gdb-or-gef-command <arg1> <arg2> <etc...>
```

加入 PEDA 或 WinDBG的喜好设定

例如对于那些使用WinDBG并喜欢其命令的人（比如我），可以通过GEF别名将它们集成到GDB中，如下所示：

```
$ nano ~/.gef.rc
[...]
[aliases]
# some windbg aliases
dps = dereference
dq = hexdump qword
dd = hexdump dword
dw = hexdump word
dc = hexdump byte
dt = pcustom
bl = info breakpoints
bp = break
be = enable breakpoints
bd = disable breakpoints
bc = delete breakpoints
tbp = tbreak
tba = thbreak
pa = advance
ptc = finish
t = stepi
p = nexti
g = gef run
uf = disassemble
```

注意：Gef本身就已经支持了这里面的许多别名 (例如 `eb` 命令)。

这里有一些PEDA别名用于过去使用PEDA的人。

```
# some peda aliases
telescope = dereference
start = entry-break
stack = dereference $sp 10
argv = show args
kp = info stack
findmem = search-pattern
```

下次加载GDB（和GEF）时将加载这些新增的别名。或者您可以使用以下命令强制GEF重新加载设置：

```
gef> gef restore
```

aslr 命令

轻松的在被调试的二进制文件上检查，启用或禁用ASLR。

检查ASLR启用状态：

```
gef> aslr
ASLR is currently disabled
```

启用ASLR：

```
gef> aslr on
[+] Enabling ASLR
gef> aslr
ASLR is currently enabled
```

禁用ASLR：

```
gef> aslr off
[+] Disabling ASLR
```

注意：此命令不能影响已加载的进程，以后GDB将附加到该进程。禁用随机化的唯一方法是设置内核设置 `/proc/sys/kernel/randomize_va_space` 为0。

assemble 命令

如果您已经安装了keystone，那么gef将提供一个方便的命令来将本机指令直接组装到您当前正在调试的体系结构的操作码上。

通过 `assemble` 或它的别名来调用该命令 `asm`：

```
gef> asm [INSTRUCTION [; INSTRUCTION ...]]
```

```
gef> asm mov eax, 1; mov ebx, 0xffffd500; mov ecx, 3; int 80h
[+] Assembling 4 instructions for x86 (32 bits):
"\xb8\x01\x00\x00\x00"          # mov eax, 1
"\xbb\x00\xd5\xff\xff"          # mov ebx, 0xffffd500
"\xb9\x03\x00\x00\x00"          # mov ecx, 3
"\xcd\x80"                      # int 80h
gef>
```

使用 `-l LOCATION` 选项，`gef` 会将 `keystone` 生成的汇编代码直接写入指定的内存位置。这使得简单地覆盖操作码非常方便。

```
gef> x/2i 0x40061e
0x40061e: call 0x400560
0x400623: cmp  eax,0x0
gef> p/d 0x400623-0x40061e
$2 = 5
gef> asm -l 0x40061e nop; nop; nop; nop; nop
[+] Assembling 5 instructions for i386:x86-64: (little endian)
[+] Overwriting 5 bytes at 0x000000000040061e
gef> x/2i 0x40061e
0x40061e: nop
0x40061f: nop
gef>
```

canary 命令

如果使用Smash Stack Protector (SSP) 编译当前调试的进程即 `-fstack-protector` 标志已传递给编译器，则此命令将显示该 `canary` 的值。这样可以方便地避免在内存中手动搜索此值。

命令 `canary` 不接受任何参数。

```
gef> canary
```

```
0x10000588 <main+52> ble  cr7, 0x10000594 <main+64>
[0] Id 1, Name: "greetz-ssp", stopped, reason: BREAKPOINT
[0] RetAddr: 0x10000574, Name: main()
gef> canary
[+] Found AT_RANDOM at 0xbffff932, reading 4 bytes
[+] The canary of process 566 is 0xe195c700
gef> x/x 0xbffff932
0xbffff932: 0xe195c73b
```

capstone-disassemble 命令

如果已安装 `capstone` 库及其Python绑定，则可以使用它来反汇编调试会话中的任何内存。这个插件的创建是为了提供“GDB”的反汇编功能的替代方案，它有时会让事情变得混乱。

您可以使用其别名 `cs-disassemble` 或 `cs` 加要反汇编的位置。如果没有指定位置，它将使用 `$pc`。

```
gef> cs main
```

```
8      buf1 = malloc(10);          ←$pc ; buf1=0x0000007fffffff5e0 →0x0
9      strcpy(buf1, "AAAAAAAA");
10     free(buf1);
11
12     buf2 = malloc(10);
13     strcpy(buf2, "BBBBBBBBB");
                                     [trace]
#0  main (ac=1, av=0x7fffffff738) at mal.c:8
Temporary breakpoint 1, main (ac=1, av=0x7fffffff738) at mal.c:8
8      buf1 = malloc(10);
gef> cs malloc
0x0000007fb7fe69c0      mov     x1, x0
0x0000007fb7fe69c4      movz   x0, #0x8
0x0000007fb7fe69c8      b      #0x7fb7fd3ad0
0x0000007fb7fe69cc      orr    x2, xzr, #0xffffffff
0x0000007fb7fe69d0      orr    x3, x0, x1
0x0000007fb7fe69d4      cmp    x3, x2
0x0000007fb7fe69d8      mul    x2, x0, x1
0x0000007fb7fe69dc      b.hi   #0x7fb7fe69e8
0x0000007fb7fe69e0      mov    x0, x2
0x0000007fb7fe69e4      b      #0x7fb7fd3ae0
0x0000007fb7fe69e8      cbz    x1, #0x7fb7fe69e0
0x0000007fb7fe69ec      udiv   x1, x2, x1
0x0000007fb7fe69f0      cmp    x1, x0
0x0000007fb7fe69f4      b.eq   #0x7fb7fe69e0
0x0000007fb7fe69f8      movz   x0, #0
0x0000007fb7fe69fc      ret
gef> x/10i malloc
0x7fb7fe69c0 <malloc>:      mov     x1, x0
0x7fb7fe69c4 <malloc+4>:      mov     x0, #0x8
0x7fb7fe69c8 <malloc+8>:      b      0x7fb7fd3ad0 <_libc_memalign@plt> // #8
0x7fb7fe69cc <malloc>:      mov     x2, #0xffffffff // #4294967295
0x7fb7fe69d0 <malloc+4>:      orr     x3, x0, x1
0x7fb7fe69d4 <malloc+8>:      cmp     x3, x2
0x7fb7fe69d8 <malloc+12>:      mul     x2, x0, x1
0x7fb7fe69dc <malloc+16>:      b.hi    0x7fb7fe69e8 <malloc+28>
0x7fb7fe69e0 <malloc+20>:      mov     x0, x2
0x7fb7fe69e4 <malloc+24>:      b      0x7fb7fd3ae0 <malloc@plt>
gef> □
```

checksec 命令

`checksec` 命令来源于 [checksec.sh](#)。它提供了一种方便的方法来确定在二进制文件中启用了哪些安全保护。

您可以在当前调试的进程上使用该命令：

```
gef> checksec
[+] checksec for '/vagrant/test-bin'
Canary:                               No
NX Support:                           Yes
PIE Support:                           No
No RPATH:                             Yes
No RUNPATH:                           Yes
Partial RelRO:                         Yes
Full RelRO:                           No
```

或者直接指定要检查的二进制文件，例如：

```
$ gdb -ex "checksec ./tests/test-x86"
```

config 命令

除了可以从 `~/.gef.rc` 读取配置外，还可以在运行时使用 `gef config` 命令配置 `gef`。

要查看加载的所有命令的所有设置：

```
gef> gef config
```

```
gef> gef config
[ GEF configuration settings ]
checksec.readelf_path      (str) = /usr/bin/readelf
context.clear_screen       (bool) = True
context.enable             (bool) = True
context.layout             (str) = regs stack code source threads trace
context.nb_lines_backtrace (int) = 10
context.nb_lines_code      (int) = 5
context.nb_lines_stack     (int) = 8
context.redirect           (str) =
context.show_registers_raw (bool) = True
context.show_stack_raw     (bool) = False
context.use_capstone       (bool) = False
dereference.max_recursion (int) = 7
gef.debug                 (bool) = False
gef.follow_child          (bool) = True
gef.no_color              (bool) = False
gef.readline_compat       (bool) = False
ida-interact.host         (str) = 127.0.1.1
ida-interact.port         (int) = 1337
pattern.length            (int) = 1024
pcustom.struct_path       (str) = /tmp/gef/structs
process-search.ps_command (str) = /bin/ps auxww
retdec.key                (str) = 1dd7cb8f-ca9f-4663-811b-2095b87d7faa
retdec.path               (str) = /tmp/gef
trace-run.max_tracing_recursion (int) = 1
trace-run.tracefile_prefix (str) = ./gef-trace-
unicorn-emulate.show_disassembly (bool) = False
unicorn-emulate.verbose   (bool) = False
gef>
```

或者获取某一个设置项的值:

```
gef> gef config pcustom.struct_path
```

当然, 您可以编辑这些设置的值。例如, 如果要在抵达断点显示当前上下文之前清除屏幕:

```
gef> gef config context.clear_screen 1
```

要将 GEF 的当前设置保存到系统, 以使这些选项在所有未来的 GEF 会话中保持不变, 只需运行:

```
gef> gef save
[+] Configuration saved to '/home/vagrant/.gef.rc'
```

启动时, 如果 gef 找到文件 `${HOME}/.gef.rc`, 它将自动加载其值。

要在会话期间重新加载设置, 只需运行:

```
gef> gef restore
[+] Configuration from '/home/hugsy/.gef.rc' restored
```

您也可以在 gdb 会话之外调整此配置文件以满足您的需要。

context 命令

```
[ registers ]
$0 0x00000000 $1 0x00000633 $2 0x00000006 $3 0x2abb3390 $4 0x2abb3850 $5 0x2abb3390
$6 0x00000000 $7 0x0000010c $8 0x00000001 $9 0x00000028 $10 0x00000003 $11 0x00000005
$12 0x0000010c $sp 0x7efff390 $lr 0x2aaf20ff $pc 0x2aae38e6 $cpsr 0x200f0030
Flags: [THUMB fast interrupt overflow CARRY zero negative]

[ stack ]
0x7efff390 +0x00: 0x7efff3a4 → 0x20 ← $sp
0x7efff394 +0x04: 0x2aaf20ff → cmn.w r0, #4096 ; 0x1000
0x7efff398 +0x08: 0x2abb00b4 → 0x0
0x7efff39c +0x0c: 0x2aaf4957 → ldr r3, [pc, #264] ; (0x2aaf4a60 <__GI_abort+456>)
0x7efff3a0 +0x10: 0x2aacb050 → 0x2aacb958 → 0x0
0x7efff3a4 +0x14: 0x20
0x7efff3a8 +0x18: 0x0
0x7efff3ac +0x1c: 0x0

[ code:arm:thumb ]
0x2aae38cb <__gcc_personality_v0+42> movs r4, r1
0x2aae38cd movs r0, r0
0x2aae38cf movs r0, r0
0x2aae38d1 <__aeabi_read_tp> mrc 15, 0, r0, cr13, cr0, {3}
0x2aae38d5 <__aeabi_read_tp+4> bx lr
0x2aae38e7 <__libc_do_syscall+6> pop {r7, pc} ← $pc
0x2aae38e9 nop.w
0x2aae38ed nop.w
0x2aae38f1 <__GI___errno_location> ldr r3, [pc, #12] ; (0x2aae3900 <__GI___errno_location+16>)
0x2aae38f3 <__GI___errno_location+2> mrc 15, 0, r0, cr13, cr0, {3}
0x2aae38f7 <__GI___errno_location+6> add r3, pc

[ threads ]
[#0] Id 1, Name: "stack-bof-armv7", stopped, reason: SIGABRT

[ trace ]
[#0] RetAddr: 0x2aae38e6, Name: __libc_do_syscall()
[#1] RetAddr: 0x2aaf20fe, Name: __GI_raise(sig=6)
[#2] RetAddr: 0x2aaf4956, Name: __GI_abort()
[#3] RetAddr: 0x2ab18de0, Name: __libc_message(do_abort=1,fmt=0x2ab9e25c "*** %s ***: %s terminated\n")
[#4] RetAddr: 0x2ab6a6d2, Name: __GI___fortify_fail(msg=0x2ab9e23c "stack smashing detected")
[#5] RetAddr: 0x2ab6a692, Name: stack_chk_fail()
[#6] RetAddr: 0x8514, Name: say_hello()
[#7] RetAddr: 0x41414140

gef>
```

gef (与 PEDA 或 fg! famous gdbinit 不同) 在遇到断点时提供全面的上下文菜单。

- 寄存器上下文框显示当前寄存器值。红色值表示自上次执行停止以来该寄存器的值已更改。它可以方便地跟踪值。也可以通过reg命令访问和/或取消引用寄存器值。
- 堆栈上下文框显示堆栈指针寄存器指向的内存中的10个（默认情况下可以调整）条目。如果这些值是指针，则它们被连续解除引用。
- 代码上下文框显示要执行的下一条指令，默认显示10条指令（默认情况下可以调整）。

编辑上下文布局

gef 允许您通过重新排列显示上下文的顺序来配置您自己的显示设置。

```
gef> gef config context.layout
```

目前有6个部分可以显示：

- `legend` : 颜色代码的文字说明
- `regs` : 寄存器的状态
- `stack` : `$sp` 寄存器指向的内存内容
- `code` : 正在执行的代码
- `args` : 如果在函数调用处停止，则打印调用参数
- `source` : 如果用source编译，这将显示相应的源代码行
- `threads` : 所有线程
- `trace` : 执行调用跟踪
- `extra` : 如果检测到漏洞（易受攻击的格式字符串，堆漏洞等），它将显示在此窗格中

- `memory` : 查看任意内存位置

要隐藏一个部分，只需使用 `context.layout` 设置，并在部分名称前加上 `-` 或者省略它。

```
gef> gef config context.layout "-legend regs stack code args -source -threads -trace extra
memory"
```

此配置不会显示 `source` , `threads` 和 `trace` 部分。

`memory` 窗格将显示 `memory` 命令指定的所有位置的内容。例如，

```
gef> memory watch $sp 0x40 byte
```

这将打印堆栈的0x40字节的hexdump版本。此命令便于跟踪内存中任意位置的变化。跟踪位置可以使用 `memory` `unwatch` 逐个删除，或者与 `memory reset` 一起删除。

大多数部分的大小也可以自定义：

- `nb_lines_stack` : 配置要显示的堆栈行数。
- `nb_lines_backtrack` : 配置要显示的回溯线数。
- `nb_lines_code` 和 `nb_lines_code_prev` : 分别配置在PC之后和之前显示的行数。
- `context.nb_lines_threads` : 确定线程窗格内显示的行数。在调试大量多线程应用程序 (apache2, firefox等) 时，这很方便。它接收一个整数作为值：如果该值为“-1”，则将显示所有线程状态。否则，如果该值设置为“N”，则最多将显示“N”个线程状态。

要使堆栈在顶部显示最大堆栈地址（即向下增加堆栈），请启用以下设置：

```
gef> gef config context.grow_stack_down True
```

如果保存的指令指针不在显示的堆栈部分内，则创建一个包含已保存的ip并且根据架构指示帧指针的部分。

```
0x00007fffffff9e8|+0x00: 0x00007ffff7a2d830 → <__main+240> mov edi, eax
($current_frame_savedip)
0x00007fffffff9e0|+0x00: 0x00000000004008c0 → <__init+0> push r15 ← $rbp
. . . (440 bytes skipped)
0x00007fffffff9e8|+0x38: 0x0000000000000000
0x00007fffffff9e0|+0x30: 0x0000000000000026 ("&")
0x00007fffffff9d8|+0x28: 0x00000000001958ac0
0x00007fffffff9d0|+0x20: 0x00007ffff7ffa2b0 → 0x5f6f7364765f5f00
0x00007fffffff9c8|+0x18: 0x00007fff00000000
0x00007fffffff9c0|+0x10: 0x00007fffffff9c950 → 0x0000000000000000
0x00007fffffff9b8|+0x08: 0x0000000000000000
0x00007fffffff9b0|+0x00: 0x00007ffff7fc7e4 → 0x0000000000000000 ← $rsp
```

将上下文输出重定向到另一个 TTY/file

默认情况下，`gef` 上下文将显示在当前TTY上。这可以通过设置 `context.redirect` 变量来覆盖，以将上下文发送到另一个部分。

为此，请使用 `gef config` 选择你希望上下文重定向到的TTY/file/socket等。

```
$ tty
/dev/pts/0
```

```
gef> gef config context.redirect /dev/pts/0
```

```
gef> gef config context.clear_screen 1
gef> gef config context.redirect /devpts/0
gef> entry
[+] Breaking at '{<text variable, no debug info> 0x400f78 <main>'

Temporary breakpoint 2, 0x000000000400800c in main ()
gef> nl
0x0000000004008004 in main ()
gef> 
0x0000000004008008 in main ()
gef> 
```

[registers]					
x0	0x0000000000000001	x1	0x0000007fffffff6f8	x2	0x0000007fffffff708
x3	0x000000000000407f8	x4	0x0000000000000000	x5	0x0000007fbfcfc08
x6	0x0000007fb7ccaa0	x7	0xab7acfbcc93979c	x8	0x2f2f2f2f2f2f2f2f
x9	0x0000007fb7febe0	x10	0x00000000fffffffff	x11	0x0000000000000090
x12	0x0000000000000730	x13	0x0000007fb7ff028	x14	0x0000000000000040
x15	0x0000007fb7ffffe0	x16	0x0000007fb7ebd1c	x17	0x00000000000011910
x18	0x0000007fffffff450	x19	0x0000000000000000	x20	0x0000000000000000
x21	0x000000000040630	x22	0x0000000000000000	x23	0x0000000000000000
x24	0x0000000000000000	x25	0x0000000000000000	x26	0x0000000000000000
x27	0x0000000000000000	x28	0x0000000000000000	x29	0x0000007fffffff570
x30	0x0000007fb7eb9e8	x31	0x0000007fffffff570	>	0x0000000004008008
	0x0000000002000000		0x0000000000000000	>	0x0000000000000000

Flags: [fast Interrupt overFlow CARRY ZERO negative]

[stack]			
0x0000007fffffff570	+0x000:	0x0000007fffffff5b0	→ 0x0000007fffffff6f0 → x1 ←\$x29, \$sp
0x0000007fffffff578	+0x008:	0x0000007fb7eb9e8	→ 0x0000000000000000
0x0000007fffffff580	+0x010:	0x0	
0x0000007fffffff588	+0x018:	0x0	
0x0000007fffffff590	+0x020:	0x0000007fffffff6f8	→ 0x0000007fffffff903 → "root/bof-aarch64"
0x0000007fffffff598	+0x028:	0x17eb9dc8	
0x0000007fffffff5a0	+0x030:	0x0	
0x0000007fffffff5a8	+0x038:	0x0	

```
0x0000000000004007f8 <great+96> ldp x29, x30, [sp], #112
0x0000000000004007f4 <great+100> ret
0x0000000000004007f8 <main> stp x29, x30, [sp,#-64]
<main+4> mov x29, sp
<main+8> str w0, [x29,#4]
0x000000000000400808 <main+16> str x2, [x29,#24] ←$pc
0x00000000000040080c <main+20>ldr w0, [x29,#44]
0x000000000000400810 <main+24>cmp w0, #0x2
0x000000000000400814 <main+28>b.eq 0x400820,<main+40>
0x000000000000400818 <main+32>mov w0, #0x1 // #1
0x00000000000040081c <main+36>b 0x400804,<main+140>
```

[code:aarch64]

```
root@debian-aarch64:~# ls -l
total 16
-rwxr-xr-x 1 root root 9881 Jul 9 01:18 bof-aarch64
-rw-r--r-- 1 root root 656 Jul 9 01:18 g.c
root@debian-aarch64:~#
```

[threads]
#0 Id 1, Name: "bof-aarch64", stopped

[trace]
#0 RetAddr: 0x400808, Name: <main>

```
gef> gef config context.redirect ""
```

```
gef> gef config context.layout "code regs stack"
```

```
gef> gef config context.enable 0
```

```
gef> gef config context.clear_screen 1
```

```
gef> gef config context.show_registers_raw 1
```

```
gef> gef config context.peek_calls False
```

- 从寄存器视图中隐藏指定寄存器。

```
gef> gef config context.ignore_registers "$cs $ds $gs"
```

dereference 命令

`dereference` 命令（也就是PEDA中的别名 `telescope`）旨在简化GDB中地址的解除引用，以确定它实际指向的内容。

这是一个有用的便利功能，可以在GDB中使用连续的“x / x”手动跟踪值。

`dereference` 需要一个强制参数，一个地址（或符号或寄存器等）来取消引用：

```
gef> dereference $sp
0x00007fffffff258|+0x00: 0x0000000000400489 → hlt      ← $rsp
gef> telescope 0x7ffff7b9d8b9
0x00007ffff7b9d8b9|+0x00: 0x0068732f6e69622f ("/bin/sh"?)
```

它还可以选择接受第二个参数，即取消引用的连续地址数（默认为“1”）。

例如，如果要取消引用函数上下文中的所有堆栈条目（在64位体系结构上）：

```
gef> p ($rbp - $rsp)/8
$3 = 4
gef> dereference $rsp 5
0x00007fffffff170|+0x00: 0x0000000000400690 → push r15      ← $rsp
0x00007fffffff178|+0x08: 0x0000000000400460 → xor ebp, ebp
0x00007fffffff180|+0x10: 0x00007fffffff270 → 0x1
0x00007fffffff188|+0x18: 0x1
0x00007fffffff190|+0x20: 0x0000000000400690 → push r15      ← $rbp
```

edit-flags 命令

`edit-flags` 命令（别名：`flags`）提供了一种快速且易于理解的方式来查看和编辑支持它的体系结构的标志寄存器。如果没有参数，该命令将只返回一个人性化的寄存器标志显示。

可以按照以下语法提供一个或多个参数：

```
gef> flags [(+|-|~)FLAGNAME ...]
```

其中 `FLAGNAME` 是标志的名称（不区分大小写），而 `+|-|~` 表示是否设置，取消设置或切换标志的操作。

例如，在x86架构上，如果我们不想进行条件跳转（例如 `jz` 指令），但我们想要设置 `carry` 标志，只需使用：

```
gef> flags -ZERO +CARRY
```

elf-info 命令

`elf-info` (别名 `elf`) 提供了有关当前加载的ELF二进制文件的一些基本信息:

```
gef> elf
Magic           : 7f 45 4c 46
Class           : 0x2 - 64-bit
Endianness      : 0x1 - Little-Endian
Version         : 0x1
OS ABI          : 0x0 - System V
ABI Version      : 0x0
Type            : 0x2 - Executable
Machine         : 0x3e - x86-64
Program Header Table : 0x0000000000000040
Section Header Table : 0x00000000000000c8
Header Table     : 0x0000000000000040
ELF Version      : 0x1
Header size      : 0 (0x0)
Entry point      : 0x0000000000400460
```

entry-break 命令

`entry-break` (别名 `start`) 命令的目标是在二进制文件中可用的最明显的入口点找到并中断。由于二进制文件将开始运行, 因此一些“PLT”条目也将被解析, 从而使进一步的调试变得更容易。

它将执行以下操作:

1. 查找 `main`。如果找到, 设置临时断点并继续。2. 否则, 它会查找 `__libc_start_main`。如果找到, 设置临时断点并继续。3. 最后, 如果找不到前两个符号, 它将从ELF头获取入口点, 设置断点并运行。如果ELF二进制文件具有有效结构, 则此情况永远不会失败。

```
gef> entry-break
[+] Breaking at '{<text variable, no debug info>} 0x846c <main>'
Temporary breakpoint 1 at 0x846c
[+] Starting execution
-----[regs]
$r0      0x00000001 $r1      0xbffff874 $r2      0xbffff87c $r3      0x0000846c
$r4      0x00000000 $r5      0x00000000 $r6      0x000083c0 $r7      0x00000000
$r8      0x00000000 $r9      0x00000000 $r10     0xb6ffe000 $r11     0x00000000
$r12     0xb6fc1000 $sp      0xbffff728 $lr      0xb6eab81c $pc      0x0000846c
-----[stack]
0xbffff728: 0xbffff728 -> 0xb6fc1000 -> 0x12cf28
0xbffff72c: 0xbffff72c -> 0xbffff874 -> 0xbffff96a -> "/home/pi/malloc"
0xbffff730: 0xbffff730 -> 0x1
0xbffff734: 0xbffff734 -> 0x0000846c
```

\$(eval) 命令

`$` 命令试图模仿WinDBG中的 `?` 命令。

当提供一个参数时, 它将评估表达式, 并尝试以各种格式显示结果:

[illegible]

有两个参数，它只会计算它们之间的差值：

```
gef> vmmap libc
Start                End                Offset                Perm
0x00007ffff7812000 0x00007ffff79a7000 0x0000000000000000  r-x  /lib/x86_64-linux-gnu/libc-
2.24.so
0x00007ffff79a7000 0x00007ffff7ba7000 0x00000000000195000  ---  /lib/x86_64-linux-gnu/libc-
2.24.so
0x00007ffff7ba7000 0x00007ffff7bab000 0x00000000000195000  r--  /lib/x86_64-linux-gnu/libc-
2.24.so
0x00007ffff7bab000 0x00007ffff7bad000 0x00000000000199000  rw-  /lib/x86_64-linux-gnu/libc-
2.24.so

gef> $ 0x00007ffff7812000 0x00007ffff79a7000
-1658880
1658880

gef> $ 1658880
1658880
0x195000
0b1100101010000000000000
b'\x19P\x00'
b'\x00P\x19'
```

format-string-helper 命令

`format-string-helper` 命令将创建一个 GEF 特定类型的断点，专门用于在使用 Glibc 库时检测可能不安全的格式字符串。

它将针对多个目标使用此新断点，包括：

- `printf()`
- `sprintf()`
- `fprintf()`
- `snprintf()`
- `vsnprintf()`

只需调用该命令即可启用此功能。

`fmtstr-helper` 是一个较短的别名。

```
gef> fmtstr-helper
```

然后开始执行。

```
gef> r
```

如果找到潜在的不安全条目，则断点将触发，停止进程执行，显示触发的原因以及关联的上下文。

```
gef> g
Not vulnerable
===== [ Format String Detection ] =====
[+] Possible insecure format string '$r0' -> 0x7efff6f0: 'Vulnerable'
[+] Triggered by 'printf()'
[*] Reason:
Call to 'printf()' with format string argument in position #0 is in
page 0x7efdf000 ([stack]) that has write permission
-----[regs]
$r0    0x7efff6f0 $r1    0x00008524 $r2    0x00000000 $r3    0x7efff6f0
$r4    0x00000000 $r5    0x00000000 $r6    0x00008388 $r7    0x00000000
$r8    0x00000000 $r9    0x00000000 $r10   0x76ffff00 $r11   0x7efff704
$r12   0x76ed8990 $sp    0x7efff6d8 $lr    0x00008480 $pc    0x76ed8994
-----[stack]
0x7efff6d8: 0x7efff6f0 -> "Vulnerable\n"
0x7efff6dc: 0x00008524 -> 0x7ffffe64
```

functions 命令

`functions` 命令将列出GEF提供的所有 [便利功能](#)。

- `$_bss([offset])` -- 返回当前的bss基址加上给定的偏移量。
- `$_got([offset])` -- 返回当前的bss基址加上给定的偏移量。
- `$_heap([offset])` -- 返回当前堆基址加上可选的偏移量。
- `$_pie([offset])` -- 返回当前的PIE基址和可选的偏移量。
- `$_stack([offset])` -- 返回当前栈基址加上可选的偏移量。

这些函数可以用作其他命令的参数，以动态计算值。

```
gef> deref $_heap() 14
0x000000000602000|+0x00: 0x0000000000000000    ← $r8
0x000000000602008|+0x08: 0x0000000000000021 ("!?")
0x000000000602010|+0x10: 0x0000000000000000    ← $rax, $rdx
0x000000000602018|+0x18: 0x0000000000000000
gef> deref $_heap(0x20) 14
0x000000000602020|+0x00: 0x0000000000000000    ← $rsi
0x000000000602028|+0x08: 0x00000000000020fe1
0x000000000602030|+0x10: 0x0000000000000000
0x000000000602038|+0x18: 0x0000000000000000
```

gef-remote 命令

可以在远程调试环境中使用 `gef`。所需文件将自动下载并缓存在临时目录（大多数Unix系统上的 `/tmp/gef`）中。如果更改目标文件，请记得手动删除缓存，否则 `gef` 将使用旧的版本。

使用本地副本

如果你想远程调试你已经拥有的二进制文件，你只需要告诉 `gdb` 在哪里找到调试信息。

例如，如果我们想调试 `uname`，我们在服务器上执行：

```
$ gdbserver 0.0.0.0:1234 /bin/uname
Process /bin/uname created; pid = 32280
Listening on port 1234
```

```
pi@rpi2-1 ~ $ gdbserver 0.0.0.0:1234 ./vuln
Process ./vuln created; pid = 18502
Listening on port 1234
Remote debugging from host 192.168.69.129
readchar: Got EOF
Remote side has terminated connection. GDBserver will reopen the connection.
Listening on port 1234
Remote debugging from host 192.168.69.129

Child terminated with signal = 0xb (SIGSEGV)
GDBserver exiting
```

在客户端上，只需运行 `gdb`：

```
$ gdb /bin/uname
gef> target remote 192.168.56.1:1234
Process /bin/uname created; pid = 10851
Listening on port 1234
```

或者

```
$ gdb
gef> file /bin/uname
gef> target remote 192.168.56.1:1234
```

没有本地副本

可以使用 `gdb` 内部函数来复制我们的目标二进制文件。

按照前面的例子，如果我们想调试 `uname`，运行 `gdb` 并连接到我们的 `gdbserver`。为了能够在 `/proc` 结构中找到正确的进程，命令 `gef-remote` 需要1个参数，即目标主机和端口。必须提供选项 `-p` 并指示远程主机上的进程PID，仅当使用扩展模式（`-E`）时。

```
$ gdb
gef> gef-remote 192.168.56.1:1234
[+] Connected to '192.168.56.1:1234'
[+] Downloading remote information
[+] Remote information loaded, remember to clean '/tmp/gef/10851' when your session is over
```

正如您所看到的，如果找不到调试信息，gef 将尝试自动下载目标文件并存储在本地临时目录中（在大多数Unix 的 /tmp 上）。如果成功，它将自动将调试信息加载到 gdb 并继续调试。

```
gef> set architecture armv5
the target architecture is assumed to be armv5
gef> gef-remote -t rpi2-1:1234 -p 18502
Reading /home/pi/vuln from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to access files locally instead.
Reading /home/pi/vuln from remote target...
Reading symbols from target:/home/pi/vuln...done.
Reading /lib/ld-linux-armhf.so.3 from remote target...
Reading /lib/ld-linux-armhf.so.3 from remote target...
Reading /lib/583a81a76e410c759c93bb5486b1e70a96f8ed.debug from remote target...
Reading /lib/.debug/583a81a76e410c759c93bb5486b1e70a96f8ed.debug from remote target...
0x76fcfae0 in ?? () from target:/lib/ld-linux-armhf.so.3
[+] Connected to 'rpi2-1:1234'
[+] Downloading remote information
[+] Remote information loaded, remember to clean '/tmp/18502' when your session is over
gef> entry
[+] Breaking at '{int (int, char **, char **)} 0x10420 <main>'
Temporary breakpoint 1 at 0x10438: file vuln.c, line 6.
[+] Starting execution
Reading /usr/lib/arm-linux-gnueabi/libc.so.6 from remote target...
Reading /lib/arm-linux-gnueabi/libc.so.6 from remote target...
Reading /lib/arm-linux-gnueabi/.debug/20dca88afd42e36bae9b86c87a97667313f7e1.debug from remote target...
Reading /lib/arm-linux-gnueabi/.debug/20dca88afd42e36bae9b86c87a97667313f7e1.debug from remote target...

$R0 0x00000001 $r1 0x7efff7a4 $r2 0x7efff7ac $r3 0x00010420 $r4 0x00010464 $r5 0x00000000 $r6 0x000102f8
$r7 0x00000000 $r8 0x00000000 $r9 0x00000000 $r10 0x76fff000 $r11 0x7efff64c $r12 0x7efff6d0 $sp 0x7efff5f8
PC 0x76e88c19 $pc 0x00010438 $cpsr 0x60000010
Flags: [ thumb fast interrupt overflow CARRY ZERO negative ]

0x7efff5f8 +0x00: 0x6e43a318 ←$sp
0x7efff5fc +0x04: 0x7efff7ac →0x7efff8be →"TERM=screen-256color"
0x7efff600 +0x08: 0x7efff7a4 →0x7efff8b7 →"/vuln"
0x7efff604 +0x0c: 0x1
0x7efff608 +0x10: 0x76ff8000
0x7efff60c +0x14: 0x7efff680 →0x0
0x7efff610 +0x18: 0x76e75a2c →0xf63d4e2e
0x7efff614 +0x1c: 0x831

0x10428 <main+0> sub sp, sp, #80 ; 0x50
0x1042c <main+12> str r0, [r11, #-72] ; 0xffffffffb8
0x10430 <main+16> str r1, [r11, #-76] ; 0xffffffffb4
0x10434 <main+20> str r2, [r11, #-80] ; 0xffffffffb0
0x10438 <main+24> ldr r3, [r11, #-76] ; 0xffffffffb4 ←$pc
0x1043c <main+28> add r3, r3, #4
0x10440 <main+32> ldr r2, [r3]
0x10444 <main+36> sub r3, r11, #68 ; 0x44
0x10448 <main+40> mov r1, r2

#0 main (argc=1, argv=0x7efff7a4, envp=0x7efff7ac) at vuln.c:6
```

然后，您可以将下载的文件重新用于将来的调试会话，在IDA使用它等。这使得整个远程调试过程（特别是对于Android应用程序）变得很简单。

QEMU用户模式

虽然GDB通过QEMU用户工作，但QEMU仅支持 gdbremote 协议中存在的所有命令的有限子集。例如，不支持诸如 remote get 或 remote put （分别从远程目标下载和上载文件）的命令。因此，gef 的默认 remote 模式也不起作用，因为 gef 将无法获取远程 procfs 的内容。

为了避免这种情况并且仍然享受QEMU用户的 gef 功能，可以人工添加一个简单的存根，使用 geq-remote 选项 -q 选项。请注意，您需要首先正确设置架构：

```
$ qemu-arm -g 1234 ./my/arm/binary
$ gdb-multiarch ./my/arm/binary
gef> set architecture arm
gef> gef-remote -q localhost:1234
```



```
[ registers ]
$R0 : 0x00000090 → 0x00000000 → 0xe59ff018 → 0xe59ff018
$R1 : 0x00000001 → 0x18e59ff0 → 0x18e59ff0
$R2 : 0x03bffff50 → 0x00000000 → 0xe59ff018 → 0xe59ff018
$R3 : 0x03bfffff → 0x00000000 → 0xe59ff018 → 0xe59ff018
$R4 : 0x101f1018 → 0x00000090 → 0x00000000 → 0xe59ff018 → 0xe59ff018
$R5 : 0x00000001 → 0x18e59ff0 → 0x18e59ff0
$R6 : 0x07ffff50 → 0x00000001 → 0x18e59ff0 → 0x18e59ff0
$R7 : 0x07ffff4f → 0x00000100 → 0x00000005 → 0x18e59ff0 → 0x18e59ff0
$R8 : 0x0000007f → 0x00000000 → 0xe59ff018 → 0xe59ff018
$R9 : 0x07ffff70 → 0x6f686365 → 0x6f686365 ("echo?")
$R10 : 0x0444ab48 → 0x65676150 → 0x65676150 ("Page?")
$R11 : 0x00000000 → 0xe59ff018 → 0xe59ff018
$R12 : 0x01010101 → 0x00000000 → 0xe59ff018 → 0xe59ff018
$Sp : 0x03ffffa4 → 0x07ffff4f → 0x00000100 → 0x00000005 → 0x18e59ff0 → 0x18e59ff0
$lr : 0x000107ec → 0xe3100040 → 0xe3100040
$pc : 0x000107ec → 0xe3100040 → 0xe3100040
$Cpsr : 0x60000193 → 0x60000193

[ stack ]
0x03ffffa4|+0x00: 0x07ffff4f → 0x00000100 → 0x00000005 → 0x18e59ff0 → 0x18e59ff0 ←$
sp
0x03ffffa8|+0x04: 0x0001092c → 0xe4c40001 → 0xe4c40001
0x03ffffac|+0x08: 0x80000110 → 0x80000110
0x03ffffb0|+0x0c: 0x03ffffc4 → 0x07ffff4f → 0x00000100 → 0x00000005 → 0x18e59ff0 → 0x18e59ff0
ff0
0x03ffffb4|+0x10: 0x00000000 → 0xe59ff018 → 0xe59ff018
0x03ffffb8|+0x14: 0x000100f8 → 0xe8bd0018 → 0xe8bd0018
0x03ffffbc|+0x18: 0x00000064 → 0x00000000 → 0xe59ff018 → 0xe59ff018
0x03ffffc0|+0x1c: 0x80000110 → 0x80000110

[ code:arm ]
0x107d4 b 0x10164
0x107d8 push {r4, lr}
0x107dc movw r4, #4120 ; 0x1018
0x107e0 movt r4, #4127 ; 0x101f
0x107e4 mov r0, r4
0x107e8 bl 0x1017c
→0x107ec tst r0, #64 ; 0x40
0x107f0 beq 0x107e4
0x107f4 mov r0, #4096 ; 0x1000
0x107f8 movt r0, #4127 ; 0x101f
0x107fc pop {r4, lr}
0x10800 b 0x1017c

[ threads ]
[#0] Id 1, Name: "", stopped, reason: SIGINT
gef> 
```

heap 命令

heap 命令提供有关指定为参数的堆块的信息。目前，它只支持Glibc堆格式(参见 [this link](#) 获取 malloc 结构信息)。子命令的语法很简单：

```
gef> heap <sub_commands>
```

heap chunks 命令

展示堆段的所有 chunks。

```
gef> heap chunks
```

在某些情况下，分配将从内存页的头立即开始。如果是，请指定第一个块的基址，如下所示：

```
gef> heap chunks <LOCATION>
```

```
gef> heap arenas
Arena (base=0x7ffff7ba7c20, top=0x55555775910, last_remainder=0x0, next=0x7ffff7ba7c20, next_free=0x0, system_mem=0x21000)
gef> vmmap heap
Start      End      Offset    Perm Path
0x000055555774000 0x000055555796000 0x0000000000000000 rw- [heap]
gef> heap chunks 0x000055555774000
gef> heap chunks 0x000055555775000
Chunk(addr=0x55555775010, size=0x250, flags=PREV_INUSE)
Chunk(addr=0x55555775260, size=0x230, flags=PREV_INUSE)
Chunk(addr=0x55555775490, size=0x80, flags=PREV_INUSE)
Chunk(addr=0x55555775510, size=0x410, flags=PREV_INUSE)
Chunk(addr=0x55555775920, size=0x206f0, flags=PREV_INUSE) ← top chunk
gef>
```

heap chunk 命令

此命令提供Glibc malloc-ed chunked的可视信息。只需将地址提供给chunk的用户内存指针，以显示与特定chunk相关的信息：

```
gef> heap chunk <LOCATION>
```

```
gef> heap chunk 0x0000000000601010
[ Chunk (used): 0x601000 ]
Chunk size: 32 (0x20)
Usable size: 16 (0x10)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MMAPPED flag: Off
NON_MAIN_ARENA flag: Off
```

heap arenas 命令

多线程程序有不同的分配区，而且main_arena的知识还不够。gef因此提供arena子命令，以帮助您在调用命令时列出程序中分配的所有分配区。

```
gef> heap arenas
[+] Listing active arena(s):
Arena (base=0x7ffff7dd5b20, top=0x7ffff7dd5b78)
gef> █
```

heap set-arena 命令

如果调试符号不存在（例如静态剥离的二进制文件），则可以指示GEF使用以下命令在不同的位置找到main_arena：

```
gef> heap set-arena <LOCATION>
```

如果分配区地址正确，则所有heap命令都将起作用，并使用指定的地址为main_arena。

heap bins 命令

Glibc使用bins来保存已被free的chunk。这是因为通过sbrk（需要系统调用）进行分配开销很大。Glibc使用这些bins来记住以前分配的chunk。因为bin是单链表或双链表，我发现总是查询gdb以获取指针地址，取消引用它，获取值chunk等等是非常痛苦的...所以我决定实现heap bin子命令，允许获取以下信息：

- fastbins

- bins
 - unsorted
 - small bins
 - large bins

heap bins fast 命令

在利用堆损坏漏洞时，有时可以方便地了解 fastbinsY 数组的状态。

fast 子命令通过显示此列表中的 fastbins 列表来帮助实现。没有任何其他参数，它将显示 main_arena 的信息。它接受一个可选参数，即另一个 arena 的地址（您可以使用 heap arenas 轻松找到它）。

```
gef> heap bins fast
[+] FastbinsY of arena 0x7ffff7dd5b20
Fastbin[0] 0x00
Fastbin[1] → FreeChunk(0x600310) → FreeChunk(0x600350)
Fastbin[2] 0x00
Fastbin[3] 0x00
Fastbin[4] 0x00
Fastbin[5] 0x00
Fastbin[6] 0x00
Fastbin[7] 0x00
Fastbin[8] 0x00
Fastbin[9] 0x00
```

其他的 heap bins X 命令

heap bins 的所有其他子命令的工作方式与 fast 相同。如果没有提供参数，gef 将回退到 main_arena。否则，它将使用指向 malloc_state 结构的基址的地址并相应地打印出信息。

heap-analysis-helper 命令

heap-analysis-helper 命令旨在通过跟踪和分析内存块的分配和释放来帮助识别 Glibc 堆不一致的过程。

目前，可以跟踪以下问题：

- NULL free
- Use-after-Free
- Double Free
- Heap overlap

可以通过运行命令简单地激活帮助程序 heap-analysis-helper。

```
gef> heap-analysis
[+] Tracking malloc()
[+] Tracking free()
[+] Disabling hardware watchpoints (this may increase the latency)
[+] Dynamic breakpoints correctly setup, GEF will break execution if a possible vulnerability is found.
[+] To disable, clear the malloc/free breakpoints (`delete breakpoints`) and restore hardware breakpoints (`set can-use-hw-watchpoints 1`)
```

帮助程序将创建专门设计的破坏程序以保持分配，从而发现潜在的漏洞。一旦激活，只需清除

`__GI__libc_free()` 和 `__GI__libc_malloc()` 即可禁用堆分析断点。也可以通过 `gef config` 命令启用/禁用手动准时检查。

允许以下设置：

- `check_null_free`：在遇到 `free(NULL)` 时中断执行(默认情况下禁用)；
- `check_double_free`：在遇到 `double free` 时中断执行；

```
0x7ffff7ab6330 <free+0> mov rax, QWORD PTR [rip+0x31cbc1] # 0x7ffff7dd2ef8 ←$pc
0x7ffff7ab6337 <free+7> mov rax, QWORD PTR [rax]
0x7ffff7ab633a <free+10> test rax, rax
0x7ffff7ab633d <free+13> jne 0x7ffff7ab63b8 <__GI__libc_free+136>
0x7ffff7ab633f <free+15> test rdi, rdi
0x7ffff7ab6342 <free+18> je 0x7ffff7ab63c0 <__GI__libc_free+144>

[#0] Id 1, Name: "use-after-free", stopped, reason: BREAKPOINT

[#0] RetAddr: 0x7ffff7ab6330, Name: __GI__libc_free(mem=0x602010)
[#1] RetAddr: 0x400686, Name: main()

[*] Heap-Analysis
Double-free detected → free(0x602010) is called at 0x7ffff7ab6330 but is already in the free-ed list
Execution will likely crash...

gef> □
```

- `check_weird_free`：针对执行 `free()` 非跟踪指针调用时；
- `check_uaf`：在遇到可能的 `Use-after-Free` 条件时中断执行。

```
0x7ffff7ac3322 <__strcpy_sse2_unaligned+948> mov dx, WORD PTR [rsi+0x4]
0x7ffff7ac3326 <__strcpy_sse2_unaligned+950> mov DWORD PTR [rdi], ecx
0x7ffff7ac3328 <__strcpy_sse2_unaligned+952> mov WORD PTR [rdi+0x4], dx ←$pc
0x7ffff7ac332c <__strcpy_sse2_unaligned+956> ret
0x7ffff7ac332d <__strcpy_sse2_unaligned+957> nop DWORD PTR [rax]
0x7ffff7ac3330 <__strcpy_sse2_unaligned+960> mov ecx, DWORD PTR [rsi]
0x7ffff7ac3332 <__strcpy_sse2_unaligned+962> mov edx, DWORD PTR [rsi+0x3]
0x7ffff7ac3335 <__strcpy_sse2_unaligned+965> mov DWORD PTR [rdi], ecx

[#0] Id 1, Name: "use-after-free", stopped, reason: BREAKPOINT

[#0] RetAddr: 0x7ffff7ac3328, Name: __strcpy_sse2_unaligned()
[#1] RetAddr: 0x400692, Name: main()

[*] Heap-Analysis
Possible Use-after-Free:
Pointer 0x602010 was freed, but is attempt to be used at 0x7ffff7ac3328

gef>
```

就像格式字符串漏洞助手一样，`heap-analysis-helper` 可能无法检测复杂的堆场景和/或提供一些误报警报。必须手动确定每个发现。

`heap-analysis-helper` 还可以用来简单地跟踪的内存块的分配和释放。可以通过将上述所有配置设置为 `False` 来简单地启用跟踪：

```
gef> gef config heap-analysis-helper.check_double_free False
gef> gef config heap-analysis-helper.check_free_null False
gef> gef config heap-analysis-helper.check_weird_free False
gef> gef config heap-analysis-helper.check_uaf False
```

然后，gef 不会通知您检测到的任何不一致，而只是在分配/释放块时显示清除消息。

```
0x555555556939 xor eax, eax
0x55555555693b call 0x555555555cb0 <error@plt>
0x555555556940 xor ebp, ebp  ←$pc
0x555555556942 mov r9, rdx
0x555555556945 pop rsi
0x555555556946 mov rdx, rsp
0x555555556949 and rsp, 0xffffffffffffff0
0x55555555694d push rax

[#0] Id 1, Name: "id", stopped, reason: BREAKPOINT
[#0] RetAddr: 0x555555556940

gef> heap-analysis-helper
[*] This feature is under development, expect bugs and instability...
[+] Tracking malloc()
[+] Tracking free()
[+] Disabling hardware watchpoints (this may increase the latency)
[+] Dynamic breakpoints correctly setup, GEF will break execution if a possible vulnerability is found.
[+] To disable, clear the malloc/free breakpoints ('delete breakpoints') and restore hardware breakpoints ('set can-use-hw-watchpoints 1')
[*] Note: The heap analysis slows down noticeably the execution.
gef> c
continuing.
[*] Heap-Analysis - malloc(5)=0x55555575f010
[*] Heap-Analysis - free(0x55555575f010)
[*] Heap-Analysis - malloc(120)=0x55555575f030
[*] Heap-Analysis - malloc(12)=0x55555575f010
[*] Heap-Analysis - malloc(776)=0x55555575f0b0
[*] Heap-Analysis - malloc(112)=0x55555575f3c0
[*] Heap-Analysis - malloc(952)=0x55555575f440
[*] Heap-Analysis - malloc(216)=0x55555575f800
[*] Heap-Analysis - malloc(432)=0x55555575f8e0
[*] Heap-Analysis - malloc(104)=0x55555575faa0
[*] Heap-Analysis - malloc(88)=0x55555575fb10
[*] Heap-Analysis - malloc(120)=0x55555575fb70
```

要获取有关当前跟踪的块的信息，请使用 `show` 子命令：

```
gef> heap-analysis-helper show
```

```
gef> heap-analysis-helper show
[*] This feature is under development, expect bugs and instability...
[+] Tracked as in-use chunks:
x malloc(16) = 0x8f74410
x malloc(9) = 0x8f74490
x malloc(16) = 0x8f74428
x malloc(9) = 0x8f74440
x malloc(16) = 0x8f74450
x malloc(9) = 0x8f74468
x malloc(16) = 0x8f74478
x malloc(9) = 0x8f74508
x malloc(16) = 0x8f744a0
x malloc(9) = 0x8f744b8
x malloc(16) = 0x8f744c8
x malloc(9) = 0x8f744e0
x malloc(16) = 0x8f744f0
x malloc(14) = 0x8f74580
x malloc(16) = 0x8f74518
x malloc(14) = 0x8f74530
x malloc(16) = 0x8f74548
x malloc(14) = 0x8f74560
x malloc(16) = 0x8f74598
```

help 命令

显示已加载命令的帮助菜单。

```

gef for linux ready, type 'gef' to start, 'gef_config' to configure
58 commands loaded, using Python engine 3.5
[+] Configuration from '/home/hugsy/.gef.rc' restored
Syntax: gef help
[ GEF - GDB Enhanced Features ]
aslr                -- View/modify GDB ASLR behavior.
assemble           -- Inline code assemble. Architecture can be set in GEF runtime config (default is
                    x86). (alias: asm)
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-break)
fd                -- Enumerate file descriptors opened by process.
format-string-helper -- Exploitable format-string helper: this command will set up specific breakpoints
                    at well-known dangerous functions (printf, sprintf, 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-alias          -- GEF defined aliases
gef-remote         -- Gef wrapper for the 'target remote' command. This command will automatically
                    download the target binary in the local temporary directory (default /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.
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 pointed by parameters with NOP. If the return option is
                    specified, it will set the return register to the specific value.
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)
pid               -- ProcessIdCommand: print the process id of the process being debugged.
process-search    -- List and filter process. (alias: ps)
Host=ph0ny|Screen=fatal_dust|Load=17.357%|Bat=25%(1:zsh) >>> (2:[tmux]) <<< (3:vagrant@localhost:~) (4:zsh) >Wed 14 Dec 2016 - 17:03 KST

```

hexdump 命令

模仿WinDBG命令。

此命令至少需要2个参数，表示数据的格式，以及用作打印hexdump的位置的值/地址/符号。可选的第3个参数用于指定要显示的qword / dword / word / bytes的数量。

该命令默认提供与WinDBG兼容的别名：

- `hexdump qword` -> `dq`
- `hexdump dword` -> `dd`
- `hexdump word` -> `dw`
- `hexdump byte` -> `db`

如果字节是可打印的，`hexdump byte` 也会尝试显示ASCII字符值（类似于Linux 上的 `hexdump -C` 命令）。

语法如下：

```
hexdump (qword|dword|word|byte) LOCATION L[SIZE] [UP|DOWN]
```

例子：

- 显示 4 QWORD 的 `$pc`：


```
gef> dq $pc 14
0x7ffff7a5c1c0+0000 | 0x4855544155415641
0x7ffff7a5c1c0+0008 | 0x0090ec814853cd89
0x7ffff7a5c1c0+0010 | 0x377d6f058b480000
0x7ffff7a5c1c0+0018 | 0x748918247c894800
```

- 显示 32 bytes 的堆栈中的某个位置:

```
gef> db 0x00007ffffffffffe5e5 132
0x00007ffffffffffe5e5      2f 68 6f 6d 65 2f 68 75 67 73 79 2f 63 6f 64 65      /home/hugsy/code
0x00007ffffffffffe5f5      2f 67 65 66 2f 74 65 73 74 73 2f 77 69 6e 00 41      /gef/tests/win.A
```

hijack-fd 命令

gef 可用于修改已调试进程的文件描述符。新文件描述符可以指向文件，管道，套接字，设备等。

要使用它，只需运行


```
gef> hijack-fd FDNUM NEWFILE
```

例如,

```
gef> hijack-fd 1 /dev/null
```

将修改当前进程文件描述符以将STDOUT重定向到 `/dev/null`。

检查此asciicast以获取可视化示例:




此命令还支持连接到ip:port（如果它作为参数提供）。例如

```
gef> hijack-fd 0 localhost:8888
```

将STDIN重定向到localhost:8888

还有一个例子:



ida-interact 命令

gef 提供了一个简单的XML-RPC客户端，用于与在特定IDA Python插件中运行的服务器通信 `ida_gef.py`（可在[这里](#)免费下载）

只需下载此脚本，然后在IDA中运行即可。当服务器运行时，您将在“输出”窗口中看到一个文本，例如:

```
[+] Creating new thread for XMLRPC server: Thread-1
[+] Starting XMLRPC server: 0.0.0.0:1337
[+] Registered 6 functions.
```

这表明XML-RPC服务器已准备就绪并正在侦听。

gef 可以通过 `ida-interact` 命令与它进行交互。此命令接收要执行的函数的名称作为第一个参数，所有其他参数是远程函数的参数。

要枚举可用的功能，只需运行

```
gef> ida-interact -h
```

```
gef> ida -h
[!] Syntax
ida-interact METHOD [ARGS]
[+] Listing methods:

ida.add_comment(int addr, string comment) => None
[ add_comment ]
Add a comment to the current IDB at the location 'address'.
Example: ida.add_comment 0x40000 "Important call here!"

ida.set_color(int addr [, int color]) => None
[ set_color ]
Set the location pointed by 'address' in the IDB colored with 'color'.
Example: ida.set_color 0x40000

ida.shutdown() => None
[ shutdown ]
Cleanly shutdown the XML-RPC service.
```

现在，要执行RPC，请使用命令 `ida-interact` 并附加其参数（如果需要）。

例如：

```
gef> ida ida.set_color 0x40061E
```

将编辑远程IDB并设置0x40061E处的背景颜色为0x005500（默认值）。

另一个方便的例子是直接 from gef 向IDA中添加注释：

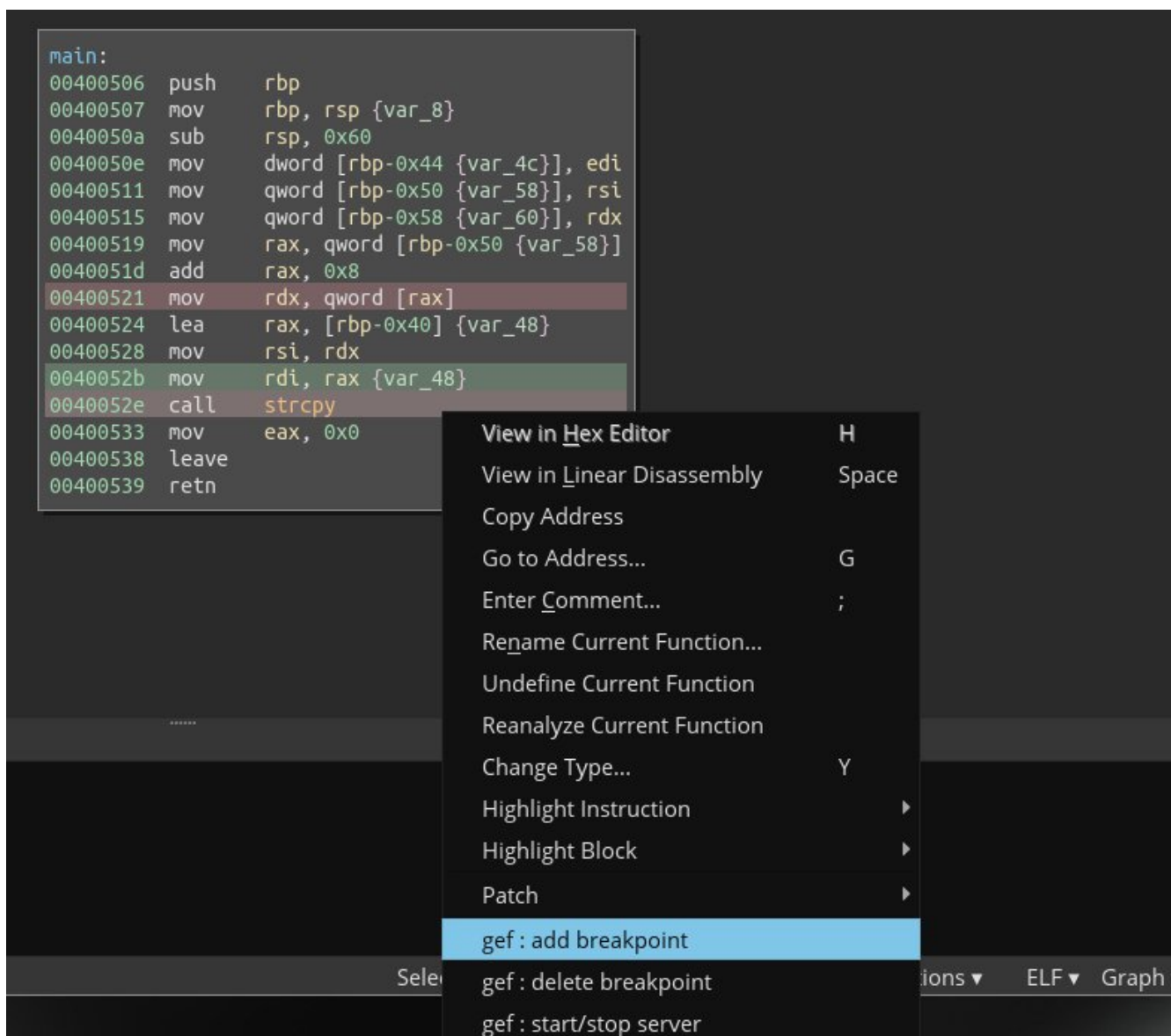
```
gef> ida ida.add_comment 0x40060C "<<<--- stack overflow"
[+] Success
```

结果：

```
.text:0000000000400602 mov     rsi, [rbp+arg_0], src
.text:0000000000400605 mov     rdi, rdx          ; dest
.text:0000000000400608 mov     [rbp+var_2C], eax
.text:000000000040060C call    _strcpy          ; <<<--- stack overflow
.text:0000000000400611 mov     rdi, [rbp+var_38] ; format
.text:0000000000400615 mov     [rbp+var_40], rax
.text:0000000000400619 mov     al, 0
.text:000000000040061B call    _printf
.text:0000000000400620 mov     [rbp+var_44], eax
.text:0000000000400623 add     rsp, 50h
```

请使用--help参数查看所有可用的方法及其语法。

值得注意的是，[Binary Ninja](#) 支持已被添加：



通过使用脚本 [binja_gef.py](#).

ksymaddr 命令

`ksymaddr` 有助于按名称查找内核符号。

语法很简单:

```
ksymaddr <PATTERN>
```

例如,

```
gef> ksymaddr commit_creds
[+] Found matching symbol for 'commit_creds' at 0xffffffff8f495740 (type=T)
[*] Found partial match for 'commit_creds' at 0xffffffff8f495740 (type=T): commit_creds
[*] Found partial match for 'commit_creds' at 0xffffffff8fc71ee0 (type=R):
__ksymtab_commit_creds
[*] Found partial match for 'commit_creds' at 0xffffffff8fc8d008 (type=r):
__kcrctab_commit_creds
[*] Found partial match for 'commit_creds' at 0xffffffff8fc9bfcd (type=r):
__kstrtab_commit_creds
```

memory 命令

只要在上下文布局中启用了“内存”部分（默认情况下），就可以注册地址，长度和分组大小。

```

0x0000000000620000    00 00 00 00 00 00 00 00 31 00 00 00 00 00 00 00    .....1.....
0x0000000000620010    0c 24 ad fb 00 00 00 00 00 00 00 00 00 00 00 00    .$.
0x0000000000620020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    .....
0x0000000000620030    00 00 00 00 00 00 00 00 d1 0f 02 00 00 00 00 00    .....
[ memory:0x620000 ]-----
0x00007fffffff528    c9 49 40 00 00 00 00 00 38 e5 ff ff ff 7f 00 00    .I@.....8.....
0x00007fffffff538    1c 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00    .....
0x00007fffffff548    90 e7 ff ff ff 7f 00 00 00 00 00 00 00 00 00 00    .....
0x00007fffffff558    98 e7 ff ff ff 7f 00 00 a9 e7 ff ff ff 7f 00 00    .....

```

添加一个查看

```
memory watch <ADDRESS> [SIZE] [(qword|dword|word|byte)]
```

移除一个查看

```
memory unwatch <ADDRESS>
```

列出所有查看

```
memory list
```

清楚所有查看

```
memory clear
```

nop 命令

`nop` 命令允许您轻松跳过指令。

```
gef> nop [-b NUM_BYTES] [-h] [LOCATION]
```

`LOCATION` 表示要绕过的指令的地址。如果未指定，它将使用程序计数器的当前值。

如果输入 `-b <bytes>`，gef 将显式修补指定的字节数。否则它会在目标位置修补 *whole* 指令。

patch 命令

将指定的值修补到指定的地址。

此命令自动别名为标准的WinDBG命令：`eb`，`ew`，`ed`，`eq`和`ea`。

```
gef> patch byte $eip 0x90
gef> eb 0x8048000 0x41
gef> ea 0xbffffd74 "This is a double-escaped string\\x00"
```

pattern 命令

此命令将创建或搜索一个 [De Bruijn](#) 循环模式，以便于确定内存中的偏移量。

应该注意的是，为了更好的兼容性，`GEF` 中实现的算法与 `pwntools` 中的算法相同，因此可以结合使用。

创建

子命令 `create` 允许创建一个新pattern：

```
gef> pattern create 128
[+] Generating a pattern of 128 bytes
aaaabaaacaaadaaaeaaafaaagaaahaaiaaajaakaaa1aaamaanaaaooaaapaaaqaaaraaasaataaaauaaaavaawaa
axaaayaaazaabbaabcaabdaabeaabfaabgaab
[+] Saved as '$_gef0'
```

该模式可以在以后用作输入。为了生成这个输入，`GET` 考虑了体系结构的大小（16,32或64位），以生成它。

与 `pwntools` 的等效命令是

```
from pwn import *
p = cyclic(128, n=8)
```

其中 `n` 是体系结构的字节数（8位为64位，4位为32位）。

查找

`search` 子命令寻找作为参数给出的值，试图在De Bruijn序列中找到它

```
gef> pattern search 0x6161616161616167
[+] Searching '0x6161616161616167'
[+] Found at offset 48 (little-endian search) likely
[+] Found at offset 41 (big-endian search)
```

请注意，寄存器也可以作为值传递：

```
gef> pattern search $rbp
[+] Searching '$rbp'
[+] Found at offset 32 (little-endian search) likely
[+] Found at offset 25 (big-endian search)
```

pcustom 命令

gef 提供了一种创建任何新结构体(以C结构体方式)和应用于当前调试环境的方法。除了简单地显示已知和用户定义的结构体之外，它还允许将这些结构体应用于当前上下文。它打算模仿非常有用的 [WinDBG 中的 dt 命令](#)。

这是通过命令 pcustom (用于 print custom) 实现的，或者你可以使用它的别名 dt (参考WinDBG命令)。

相关配置

新结构体可以存储在配置给出的位置：

```
gef> gef config pcustom.struct_path
```

默认情况下，此位置位于 `$TEMP/gef/structs` (例如 `/tmp/user/1000/gef/structs`)。

可以在一个名为 `<struct_name>.py` 的文件中创建为一个简单的 ctypes 结构体。

您可以将此路径设置为新位置

```
gef> gef config pcustom.struct_path /my/new/location
```

并保存此更改，以便下次使用 gdb 时可以直接使用它

```
gef> gef save
[+] Configuration saved to '~/gef.rc'
```

使用用户定义的结构体

使用如下命令您可以通过列出现有的自定义结构体

```
gef> dt -l
[+] Listing custom structures:
  → struct5
  → struct6
```

要创建或编辑结构体，请使用 `dt <struct_name> -e` 以使用目标结构体生成EDITOR。如果文件不存在，gef 将创建树和文件，并用 ctypes 模式填充它，你可以立即使用！

```
gef> dt mystruct_t -e
[+] Creating '/tmp/gef/structs/mystruct_t.py' from template
```

代码可以像任何Python (使用 ctypes) 代码一样定义。

```

from ctypes import *

'''
typedef struct {
    int age;
    char name[256];
    int id;
} person_t;
'''

class person_t(Structure):
    _fields_ = [
        ("age", c_int),
        ("name", c_char * 256),
        ("id", c_int),
    ]

    _values_ = [
        # You can define a function to substitute the value
        ("age", lambda age: "old" if age > 40 else "Young"),
        # or alternatively a list of 2-tuples
        ("id", [
            (0, "root"),
            (1, "normal user"),
            (None, "Invalid person")
        ])
    ]

```

`pcustom` 至少需要一个参数，即结构体的名称。只有一个参数时，`pcustom` 将转储此结构体的所有字段。

```

gef> dt person_t
+0000 age c_int (x4) → Young
+0004 name c_char_Array_256 (x100)
+0104 id c_int (x1) → normal user

```

通过提供地址或GDB符号，`gef` 将把这个用户定义的结构体应用于指定的地址：

```

gef> dt person $rsi
0x7fffffffdfb0+0x00 age:      18 (c_int)
0x7fffffffdfb0+0x04 name:    b'Bob' (c_char_Array_256)
0x7fffffffdfb0+0x104 id:     0 (c_int)

```

这意味着我们现在可以非常轻松地创建新的用户定义结构体

观看Asciinema的演示视频：

This recording has been archived

此外，如果您已成功配置IDA设置（请参阅命令 `ida-interact`），您还可以直接在GDB会话中直接导入在IDA中进行逆向工程的结构体：

```

00000000 ; -----
00000000
00000000 struc_1      struc ; (sizeof=0x10, mappedto_1)
00000000 field_0      dq ?
00000008 field_8      dd ?
0000000c field_C      dd ?
00000010 struc_1      ends
00000010
00000000 ; -----
00000000
00000000 struc_2      struc ; (sizeof=0x110, mappedto_2)
00000000 field_0      dq ?
00000008 field_8      dq ?
00000010 field_10     db 256 dup(?)
00000110 struc_2      ends
00000110

```

然后使用命令 `ida ImportStructs` 导入所有结构体，或者 `ida ImportStruct <StructName>` 只导入一个特定的结构体：

```

gef> ida ImportStructs
[+] Success

```

```

~/code/gef $ gdb tests/win
gef loaded, 'gef help' to start, 'gef config' to configure
37 commands loaded (21 sub-commands), using Python engine 3.5
[+] Configuration from '/home/hugsy/.gef.rc' restored
Reading symbols from tests/win...(no debugging symbols found)...done.
gef> ida ImportStructs
[+] Success
gef> dt -l
[+] Listing custom structures:
→ struc_1
→ struc_2
gef> dt struc_1
[!] Invalid structure name 'struct_1'
gef> dt struc_1
+0000 field_0 c_ulong (0x8)
+0008 field_8 c_uint (0x4)
+000c field_C c_uint (0x4)
gef> dt struc_2
+0000 field_0 c_ulong (0x8)
+0008 field_8 c_ulong (0x8)
+0010 field_10 c_byte_Array_256 (0x100)
gef> 
Host=ph0ny|Screen=charming supermarket|Load=0.10 0.13 0.15

```

pie 命令

`pie` 命令提供了一种为启用PIE的二进制文件设置断点的有用方法。`pie` 命令提供我们称之为“PIE断点”的东西。PIE断点只是一个虚拟断点，当进程附加时，它将被设置为实际断点。PIE断点的地址是二进制基址的偏移量。

请注意，您需要使用整个PIE命令序列来支持PIE断点，尤其是 `pie` 命令提供的“附加”命令，如 `pie attach`，`pie run` 等。

用法：

```
gef> pie <sub_commands>
```

pie breakpoint 命令

此命令设置新的PIE断点。它可以像gdb中的普通 breakpoint 命令一样使用。该位置只是与基址的偏移量。此命令后不会立即设置断点。相反，它将在您使用 pie attach, pie run, pie remote 实际附加到进程时设置，因此它可以解析正确的基址。

用法：

```
gef> pie breakpoint <LOCATION>
```

pie info 命令

由于PIE断点不是真正的断点，因此该命令提供了一种观察所有PIE断点状态的方法。

这就像gdb中的 info breakpoint。

```
gef> pie info
VNum    Num Addr
1       N/A 0xdeadbeef
```

VNum是虚拟号码，它是PIE断点的编号。Num是gdb中相应实际断点数的编号。地址是PIE断点的地址。

您可以忽略VNum参数以获取所有PIE断点的信息。

用法：

```
gef> pie info [VNum]
```

pie delete 命令

给定该PIE断点的VNum时，此命令将删除PIE断点。

用法：

```
gef> pie delete <VNum>
```

pie attach 命令

与gdb的 attach 命令相同。如果您有PIE断点，请始终使用此命令而不是原始 attach。这将在附加时设置真正的断点。

用法与 attach 相同。

pie remote 命令

与gdb的 remote 命令相同。如果您有PIE断点，请始终使用此命令而不是原始 remote。这将在附加时设置真正的断点。

用法与 `remote` 相同。

pie run 命令

与gdb的 `run` 命令相同。如果您有PIE断点，请始终使用命令而不是原始 `run`。这将在附加时设置真正的断点。

用法与 `run` 相同。

print-format 命令

- 命令 `print-format` (别名 `pf`) 将根据指定的编程语言的语法将任意位置转储为字节数组。目前，支持的输出语言是
 - Python (`py` - 默认)
 - C (`c`)
 - Assembly (`asm`)
 - Javascript (`js`)

```
gef> print-format -h
[+] print-format [-f FORMAT] [-b BITSIZE] [-l LENGTH] [-c] [-h] LOCATION
    -f FORMAT specifies the output format for programming language, available value is
    py, c, js, asm (default py).
    -b BITSIZE specifies size of bit, available values is 8, 16, 32, 64 (default is
    8).
    -l LENGTH specifies length of array (default is 256).
    -c The result of data will copied to clipboard (requires xclip)
    LOCATION specifies where the address of bytes is stored.
```

例如，此命令将从 `$rsp` 转储10个字节，并将结果复制到剪贴板。

```
gef> print-format -f py -b 8 -l 10 -c $rsp
[+] Copied to clipboard
buf = [0x87, 0xfa, 0xa3, 0xf7, 0xff, 0x7f, 0x0, 0x0, 0x30, 0xe6]
```

process-search 命令

`process-search` (又名 `ps`) 是一个方便的命令，用于在主机上列出和过滤进程。它的目的是在针对分叉过程（例如在新连接上分叉的tcp/listen守护进程）时使调试过程更容易一些。

如果没有参数，它将返回用户可以访问的所有进程：


```
gef> ps
1          root          0.0          0.4          ?          /sbin/init
2          root          0.0          0.0          ?          [kthreadd]
3          root          0.0          0.0          ?          [ksoftirqd/0]
4          root          0.0          0.0          ?          [kworker/0:0]
5          root          0.0          0.0          ?          [kworker/0:0H]
6          root          0.0          0.0          ?          [kworker/u2:0]
7          root          0.0          0.0          ?          [rcu_sched]
8          root          0.0          0.0          ?          [rcuos/0]
9          root          0.0          0.0          ?          [rcu_bh]
10         root          0.0          0.0          ?          [rcuob/0]
11         root          0.0          0.0          ?          [migration/0]
[...]
```

或者启用过滤器：

```
gef> ps bash
22590      vagrant      0.0      0.8      pts/0      -bash
```

`ps` 也允许使用以下选项：

- `-s` (`smart`) 将丢弃一些进程（属于不同的用户，用作参数的模式而不是命令等）
- `-a` (`attach`) 将自动附加到找到的第一个进程

因此，例如，如果您的目标进程名为 `/home/foobar/plop`，但现有实例通过 `socat` 使用，如

```
$ socat tcp-l:1234,fork,reuseaddr exec:/home/foobar/plop
```

每次向 `tcp/1234` 打开一个新连接时，`plop` 将被分叉，并且 `gef` 可以通过命令轻松附加到它

```
gef> ps -as plop
```

process-status 命令

此命令用于替换旧命令 `pid` 和 `fd`。

`process-status` 提供了对当前运行进程的详尽描述，通过扩展 GDB `info proc` 命令提供的信息，以及来自 `procfs` 结构的所有信息。

```
gef> ps -s zsh
22879
gef> attach 22879
[...]
gef> status
[+] Process Information
    PID → 22879
    Executable → /bin/zsh
    Command line → '-zsh'
[+] Parent Process Information
    Parent PID → 4475
```

```
Command line → 'tmux new -s cool vibe'
[+] Children Process Information
    PID → 26190 (Name: '/bin/sleep', CmdLine: 'sleep 100000')
[+] File Descriptors:
    /proc/22879/fd/0 → /dev/pts/4
    /proc/22879/fd/1 → /dev/pts/4
    /proc/22879/fd/2 → /dev/pts/4
    /proc/22879/fd/10 → /dev/pts/4
[+] File Descriptors:
    No TCP connections
```

registers 命令

`registers` 命令将打印所有寄存器并取消引用任何指针。它没有任何参数。

MIPS主机上的示例：

```
gef> reg
$zero      : 0x00000000
$at        : 0x00000001
$v0        : 0x7fff6cd8 -> 0x77e5e7f8 -> <__libc_start_main+200>: bnez v0,0x77e5e8a8
$v1        : 0x77ff4490
$a0        : 0x00000001
$a1        : 0x7fff6d94 -> 0x7fff6e85 -> "/root/demo-mips"
$a2        : 0x7fff6d9c -> 0x7fff6e91 -> "SHELL=/bin/bash"
$a3        : 0x00000000
$t0        : 0x77fc26a0 -> 0x0
$t1        : 0x77fc26a0 -> 0x0
$t2        : 0x77fe5000 -> "_dl_fini"
$t3        : 0x77fe5000 -> "_dl_fini"
$t4        : 0xf0000000
$t5        : 0x00000070
$t6        : 0x00000020
$t7        : 0x7fff6bc8 -> 0x0
$s0        : 0x00000000
$s1        : 0x00000000
$s2        : 0x00000000
$s3        : 0x00500000
$s4        : 0x00522f48
$s5        : 0x00522608
$s6        : 0x00000000
$s7        : 0x00000000
$t8        : 0x0000000b
$t9        : 0x004008b0 -> <main>: addiu sp,sp,-32
$k0        : 0x00000000
$k1        : 0x00000000
$s8        : 0x00000000
$status    : 0x0000a413
$badvaddr  : 0x77e7a874 -> <__cxa_atexit>: lui gp,0x15
$cause     : 0x10800024
$pc        : 0x004008c4 -> <main+20>: li v0,2
$sp        : 0x7fff6ca0 -> 0x77e4a834 -> 0x29bd
```

```
$hi      : 0x000001a5
$lo      : 0x00005e17
$fir     : 0x00739300
$fcsr    : 0x00000000
$ra      : 0x77e5e834 -> <__libc_start_main+260>: lw gp,16(sp)
$gp      : 0x00418b20
```

reset-cache 命令

这是一个过时的函数，用于重置GEF内部memoize缓存，不需要再从命令行调用它。

这个命令很快就会消失.....

ropper 命令

`ropper` 是一个gadget查找工具，可以通过 `pip` 轻松安装。它提供了一个非常方便的 `--search` 函数来从正则表达式搜索gadget：

```
gef> ropper --search "pop r?i; ret"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop r?i; ret

[INFO] File: /bin/ls
0x0000000000403fa6: pop rdi; ret;
0x0000000000404b98: pop rsi; ret;

gef> █
```

`ropper` 带有一整套选项，所有选项都记录在 `--help` 菜单中。

scan 命令

`scan` 搜索位于属于另一个程序的内存映射（haystack）中的地址。

```
gef> scan libc stack
[+] Searching for addresses in 'libc' that point to 'stack'
libc-2.23.so: 0x00007ffff7dd23d0|+0x13d0: 0x00007fffffe833 → "heap.out"
libc-2.23.so: 0x00007ffff7dd23d8|+0x13d8: 0x00007fffffe81b → "/vagrant/tests/binaries/heap.out"
gef>
```

`scan` 需要两个参数，第一个是要搜索的内存部分，第二个是要搜索的内容。参数是针对进程内存映射的（与 [vmmap](#) 一样，以确定要搜索的内存范围。

search-pattern 命令

`gef` 允许您在运行时搜索进程内存布局的所有段中的特定字符串。`search-pattern` 命令，别名 `grep`，旨在直接使用：

```
gef> search-pattern MyPattern
```

```
gef> grep /sh
[+] Searching '/sh' in memory
[+] In '/lib/x86_64-linux-gnu/libc-2.24.so'(0x7ffff7a3b000-0x7ffff7bd0000), permission=r-x
0x7ffff7a6c842 - 0x7ffff7a6c845 → "/sh[...]"
0x7ffff7a6c8bd - 0x7ffff7a6c8c0 → "/sh[...]"
0x7ffff7b9c700 - 0x7ffff7b9c70d → "/share/locale"
0x7ffff7b9c81d - 0x7ffff7b9c820 → "/sh"
0x7ffff7b9e36b - 0x7ffff7b9e372 → "/shells"
0x7ffff7b9ffc3 - 0x7ffff7b9ffc3 → "/share/locale/%L/%N:/usr/share/locale/%L/LC_MESSAGES[...]"
0x7ffff7b9ffa4 - 0x7ffff7b9ffdb → "/share/locale/%L/LC_MESSAGES/%N:/usr/share/locale/[...]"
0x7ffff7b9ffc8 - 0x7ffff7b9ffff → "/share/locale/%L/%N:/usr/share/locale/%L/LC_MESSAGES[...]"
0x7ffff7b9ffe0 - 0x7ffff7ba0000 → "/share/locale/%L/LC_MESSAGES/%N:"
0x7ffff7ba3754 - 0x7ffff7ba3761 → "/share/locale"
0x7ffff7ba39e4 - 0x7ffff7ba39f3 → "/share/zoneinfo"
[+] In '[stack]'(0x7ffffde000-0x7fffff0000), permission=rw-
0x7fffffed6f - 0x7fffffed6f → "/share:/usr/share:/usr/share/gdm:/var/lib/menu-xdg[...]"
0x7fffffed7a - 0x7fffffedb1 → "/share:/usr/share/gdm:/var/lib/menu-xdg:/usr/local[...]"
0x7fffffed85 - 0x7fffffedbc → "/share/gdm:/var/lib/menu-xdg:/usr/local/share:/usr[...]"
0x7fffffedac - 0x7fffffedec3 → "/share:/usr/share:/usr/share/gdm:/var/lib/menu-xdg[...]"
0x7fffffedb8 - 0x7fffffedec2 → "/share:/usr/share/gdm:/var/lib/menu-xdg/"
0x7fffffedc4 - 0x7fffffedec2 → "/share/gdm:/var/lib/menu-xdg/"
0x7fffffee03 - 0x7fffffee09 → "/share"
```

它将提供一个易于理解的特定字符串的发现，包括它/它们被发现的部分，以及与该部分相关的权限。

`search-pattern` 也可用于搜索地址。为此，只需确保您的字符串以“0x”开头并且是有效的十六进制地址。例如：

```
gef> search-pattern 0x4005f6
```

```
gef> search-pattern 0x4005f6
[+] Searching '0x4005f6' in memory
[+] In '/home/hugsy/tmp/a.out'(0x601000-0x602000), permission=rw-
0x601028 - 0x601034 → "\xf6\x05\x40[...]"
```

`search-pattern` 命令也可以用作搜索地址交叉引用的方法。因此，别名 `xref` 也指向命令 `search-pattern`。因此，上面的命令相当于 `xref 0x4005f6`，这使得它更直观。

set-permission 命令

添加此命令是为了便于漏洞利用过程，方法是直接从调试器更改特定内存页上的权限。

默认情况下，GDB 不允许您这样做，因此该命令将修改正在调试的二进制文件的代码部分，并添加本机 `mprotect` 系统调用存根。例如，对于 x86，将插入以下存根：

```
pushad
mov eax, mprotect_syscall_num
mov ebx, address_of_the_page
mov ecx, size_of_the_page
mov edx, permission_to_set
int 0x80
popad
```

在此存根之后添加断点，该点在命中时将恢复原始上下文，允许您继续执行。

`mprotect` 是 `set-permission` 的别名。举个例子，在这个二进制文件中将 `stack` 设置为 `READ|WRITE|EXECUTE`

```
gef> vmmap
      Start      End      Offset Perm Path
0x08048000 0x08049000 0x00000000 r-x  /home/hugsy/code/gef/win
0x08049000 0x0804a000 0x00000000 rw-  /home/hugsy/code/gef/win
0xf7dff000 0xf7e00000 0x00000000 rw-
0xf7e00000 0xf7fb1000 0x00000000 r-x  /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
0xf7fb1000 0xf7fb4000 0x001b0000 r--  /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
0xf7fb4000 0xf7fb6000 0x001b3000 rw-  /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
0xf7fb6000 0xf7fb8000 0x00000000 rw-
0xf7fd6000 0xf7fd8000 0x00000000 rw-
0xf7fd8000 0xf7fda000 0x00000000 r--  [vvar]
0xf7fda000 0xf7fdb000 0x00000000 r-x  [vdso]
0xf7fdb000 0xf7ffc000 0x00000000 r-x  /lib/i386-linux-gnu/ld-2.21.so
0xf7ffc000 0xf7ffd000 0x00020000 r--  /lib/i386-linux-gnu/ld-2.21.so
0xf7ffd000 0xf7ffe000 0x00021000 rw-  /lib/i386-linux-gnu/ld-2.21.so
0xffffdd00 0xfffffe00 0x00000000 rw-  [stack]
gef> mprotect 0xffffdd00
[+] Generating stub: sys_mprotect(0xffffdd00, 135168, 7)
[+] Saving original code
[+] Setting a restore breakpoint at *0x80484ae
Breakpoint 2 at 0x80484ae
[+] Overwriting current memory at 0xffffdd00 (24 bytes)
[+] Resuming execution
[+] Restoring original context
[+] Restoring $pc
```

运行

```
gef> mprotect 0xffffdd00
```

就是这样! `gef` 将使用内存运行时的信息来正确调整整个部分的保护。

```
gef> vmmap
      Start      End      Offset Perm Path
0x08048000 0x08049000 0x00000000 r-x  /home/hugsy/code/gef/win
0x08049000 0x0804a000 0x00000000 rw-  /home/hugsy/code/gef/win
0xf7dff000 0xf7e00000 0x00000000 rw-
0xf7e00000 0xf7fb1000 0x00000000 r-x  /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
0xf7fb1000 0xf7fb4000 0x001b0000 r--  /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
0xf7fb4000 0xf7fb6000 0x001b3000 rw-  /lib/i386-linux-gnu/i686/cmov/libc-2.21.so
0xf7fb6000 0xf7fb8000 0x00000000 rw-
0xf7fd6000 0xf7fd8000 0x00000000 rw-
0xf7fd8000 0xf7fda000 0x00000000 r--  [vvar]
0xf7fda000 0xf7fdb000 0x00000000 r-x  [vdso]
0xf7fdb000 0xf7ffc000 0x00000000 r-x  /lib/i386-linux-gnu/ld-2.21.so
0xf7ffc000 0xf7ffd000 0x00020000 r--  /lib/i386-linux-gnu/ld-2.21.so
0xf7ffd000 0xf7ffe000 0x00021000 rw-  /lib/i386-linux-gnu/ld-2.21.so
0xffffdd00 0xffffdd00 0x00000000 rw-
0xffffdd00 0xfffffe00 0x00000000 rwX [stack]
gef> █
```

或者在PowerPC VM上获得完整的演示视频:

This recording has been archived

shellcode 命令

`shellcode` 是@JonathanSalwan `shellcodes`数据库的命令行客户端。它可以用来直接通过 `GEF` 搜索和下载你正在寻找的shellcode。有两个原始子命令, `search` 和 `get`

```

gef> shellcode search arm
[+] Showing matching shellcodes
901      Linux/ARM      Add map in /etc/hosts file - 79 bytes
853      Linux/ARM      chmod("/etc/passwd", 0777) - 39 bytes
854      Linux/ARM      creat("/root/pwned", 0777) - 39 bytes
855      Linux/ARM      execve("/bin/sh", [], [0 vars]) - 35 bytes
729      Linux/ARM      Bind Connect UDP Port 68
730      Linux/ARM      Bindshell port 0x1337
[...]
gef> shellcode get 698
[+] Downloading shellcode id=698
[+] Shellcode written as '/tmp/sc-EfcWtM.txt'
gef> system cat /tmp/sc-EfcWtM.txt
/*
Title:      Linux/ARM - execve("/bin/sh", [0], [0 vars]) - 27 bytes
Date:       2010-09-05
Tested on:  ARM926EJ-S rev 5 (v5l)
Author:     Jonathan Salwan - twitter: @jonathansalwan

shell-storm.org

Shellcode ARM without 0x20, 0x0a and 0x00
[...]
```

stub 命令

`stub` 命令允许你存根函数，可选择指定返回值。

```
gef> stub [-h] [-r RETVAL] [LOCATION]
```

`LOCATION` 表示要绕过的功能的地址。如果未指定，gef将认为程序计数器处的指令是函数的开始。

如果提供了 `-r RETVAL`，gef会将返回值设置为提供的值。否则，它会将返回值设置为0。

例如，绕过 `fork()` 调用是微不足道的。由于返回值设置为0，因此它实际上将我们放入“子”进程。必须注意的是，这是一个与经典的“set follow-fork-mode child”不同的行为，因为在这里我们不会产生一个新的进程，我们只是欺骗父进程认为它已经成为了孩子。

例子

绕过 `fork()` 调用:

- Without stub:

```

[ registers ]
$rax 0x000000004005c7 $rbx 0x0000000000000000 $rcx 0x0000000000000000 $rdx 0x00007fffffff5a8
$rsp 0x00007fffffff4b0 $rbp 0x00007fffffff4b0 $rsi 0x00007fffffff598 $rdi 0x0000000000000001
$rip 0x000000004005cb $r8 0x00000000400690 $r9 0x00007ffff7de78e0 $r10 0x0000000000000846
$r11 0x00007ffff7a2e740 $r12 0x000000004004c0 $r13 0x00007fffffff590 $r14 0x0000000000000000
$r15 0x0000000000000000 $cs 0x0000000000000033 $ss 0x000000000000002b $ds 0x0000000000000000
$es 0x0000000000000000 $fs 0x0000000000000000 $gs 0x0000000000000000 $eflags 582
Flags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume virtualx86 identification]

[ stack ]
0x00007fffffff4b0 +0x00: 0x00000000400620 → push r15 + $rsp, $rbp
0x00007fffffff4b8 +0x08: 0x00007ffff7a2e830 → mov edi, eax
0x00007fffffff4c0 +0x10: 0x00
0x00007fffffff4c8 +0x18: 0x00007fffffff598 → 0x00007fffffff7da → "/code/snippets/fork64"
0x00007fffffff4d0 +0x20: 0x0100000000
0x00007fffffff4d8 +0x28: 0x000000004005c7 → push rbp
0x00007fffffff4e0 +0x30: 0x00
0x00007fffffff4e8 +0x38: 0x6524640a1a6ba0c2

[ code:i386:x86-64 ]
0x000000004005c4 <do_something+14> nop
0x000000004005c5 <do_something+15> pop rbp
0x000000004005c6 <do_something+16> ret
0x000000004005c7 <main+0> push rbp
0x000000004005c8 <main+1> mov rbp, rsp
0x000000004005cb <main+4> sub rsp, 0x20 + $pc
0x000000004005cf <main+8> mov DWORD PTR [rbp-0x14], edi
0x000000004005d2 <main+11> mov QWORD PTR [rbp-0x20], rsi
0x000000004005d6 <main+15> call 0x4004a0 <fork@plt>
0x000000004005db <main+20> mov DWORD PTR [rbp-0x4], eax
0x000000004005de <main+23> cmp DWORD PTR [rbp-0x4], 0x0

[ threads ]
[#0] Id 1, Name: "fork64", stopped, reason: BREAKPOINT

[ trace ]
[#0] RetAddr: 0x4005cb, Name: main()

gef> c
Continuing.
Parent process spawned child 5135
[New process 5135]
We are in the child process!
Doing something
[Inferior 2 (process 5135) exited normally]
gef>

```

- With stub:

```

[ registers ]
$rax 0x000000004005c7 $rbx 0x0000000000000000 $rcx 0x0000000000000000 $rdx 0x00007fffffff5a8
$rsp 0x00007fffffff4b0 $rbp 0x00007fffffff4b0 $rsi 0x00007fffffff598 $rdi 0x0000000000000001
$rip 0x000000004005cb $r8 0x00000000400690 $r9 0x00007ffff7de78e0 $r10 0x0000000000000846
$r11 0x00007ffff7a2e740 $r12 0x000000004004c0 $r13 0x00007fffffff590 $r14 0x0000000000000000
$r15 0x0000000000000000 $cs 0x0000000000000033 $ss 0x000000000000002b $ds 0x0000000000000000
$es 0x0000000000000000 $fs 0x0000000000000000 $gs 0x0000000000000000 $eflags 582
Flags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume virtualx86 identification]

[ stack ]
0x00007fffffff4b0 +0x00: 0x00000000400620 → push r15 + $rsp, $rbp
0x00007fffffff4b8 +0x08: 0x00007ffff7a2e830 → mov edi, eax
0x00007fffffff4c0 +0x10: 0x00
0x00007fffffff4c8 +0x18: 0x00007fffffff598 → 0x00007fffffff7da → "/code/snippets/fork64"
0x00007fffffff4d0 +0x20: 0x0100000000
0x00007fffffff4d8 +0x28: 0x000000004005c7 → push rbp
0x00007fffffff4e0 +0x30: 0x00
0x00007fffffff4e8 +0x38: 0x57567ddcda3324b4

[ code:i386:x86-64 ]
0x000000004005c4 <do_something+14> nop
0x000000004005c5 <do_something+15> pop rbp
0x000000004005c6 <do_something+16> ret
0x000000004005c7 <main+0> push rbp
0x000000004005c8 <main+1> mov rbp, rsp
0x000000004005cb <main+4> sub rsp, 0x20 + $pc
0x000000004005cf <main+8> mov DWORD PTR [rbp-0x14], edi
0x000000004005d2 <main+11> mov QWORD PTR [rbp-0x20], rsi
0x000000004005d6 <main+15> call 0x4004a0 <fork@plt>
0x000000004005db <main+20> mov DWORD PTR [rbp-0x4], eax
0x000000004005de <main+23> cmp DWORD PTR [rbp-0x4], 0x0

[ threads ]
[#0] Id 1, Name: "fork64", stopped, reason: BREAKPOINT

[ trace ]
[#0] RetAddr: 0x4005cb, Name: main()

gef> stub fork
Breakpoint 2 at 0x7ffff7ad9790: file ../sysdeps/nptl/fork.c, line 49.
[+] All calls to 'fork' will be skipped (with return value set to 0x0)
gef> c
Continuing.
[+] Ignoring call to 'fork' (setting $rax to 0x0)
We are in the child process!
Doing something
[Inferior 1 (process 5147) exited normally]

```

theme 命令

通过改变颜色方案来定制 GEF。

```
gef> theme
context_title_message      : red bold
default_title_message      : red bold
default_title_line         : green bold
context_title_line         : green bold
disable_color              : 0
xinfo_title_message        : blue bold
```

改变颜色

您可以使用 `theme` 命令更改 GEF 显示的着色属性。该命令接受2个参数，要更新的属性的名称及其新的着色值。

颜色可以是以下之一：

- red
- green
- blue
- yellow
- gray
- pink

Color还接受以下属性：

- bold
- underline
- highlight
- blink

任何其他值都会被忽略。

```
gef> theme context_title_message blue bold foobar
gef> theme
context_title_message      : blue bold
default_title_message      : red bold
default_title_line         : green bold
context_title_line         : green bold
disable_color              : 0
xinfo_title_message        : blue bold
```

tmux-setup 命令

为了使调试会话更容易，同时更有效，GEF 整合了两个命令：

- `tmux-setup`
- `screen-setup`


```

gef> tmux-setup
[*] tmux session found, splitting window...
[*] Setting context.redirect to '/dev/pts/5'...
[*] Done!
gef> set height unlimited
gef> start
[*] Breaking at '[int(int, char **, char **)] 0x048530 <main>'

Temporary breakpoint 1, main(argc=0x1, argv=0xffffd444, envp=0xffffdd4c) at /home/hugsy/labs/bof.c:17
17      if (argc != 2){
gef> 

```

```

[ registers ]
eax    0xffffd44c %ebx    0x00000000 %ecx    0xffffd444 %edx    0x00000001 %esp    0xffffd390
ebp    0xffffd3ab %esi    0x00000001 %edi    0xffffd400 %eip    0x004854f %cs    0x00000023
ds     0x00000020 ss     0x00000025 fs     0x00000020 gfs%  0x00000000 %gs     0x00000063
eflags  0         ← eip
Flags: [carry PARRY adjust zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification]

[ stack ]
0xfffffd390 → +0x00: 0xf7fa43dc → 0xf7fa51e0 → 0x0          ← $esp
0xfffffd394 → +0x04: 0xb084823c → 0xb080002f ("??")
0xfffffd398 → +0x08: 0xffffd44c → 0xffffd6a0 → "ALTERNATE_EDITOR=name"
0xfffffd39c → +0x0c: 0xffffd444 → 0xffffd627 → "/home/hugsy/labs/bof.x32"
0xfffffd3a0 → +0x10: 0x1
0xfffffd3a4 → +0x14: 0x0
0xfffffd3ab → +0x18: 0x0          ← %ebp
0xfffffd3ac → +0x1c: 0xf7e9276 → add esp, 0x10

[ code:1386 ]
0x004853c <main+12> mov ebx,DWORD PTR [ebp+0x8]
0x004853f <main+15> mov DWORD PTR [ebp+0x4],0x0
0x0048546 <main+22> mov DWORD PTR [ebp-0x0],edx
0x0048549 <main+25> mov DWORD PTR [ebp-0xc],ecx
0x004854c <main+28> mov DWORD PTR [ebp-0x10],eax
0x004854f <main+31> cmp DWORD PTR [ebp-0x8],0x2   ← $pc
0x0048553 <main+35> je 0xb0848576 <main+76>
0x0048559 <main+41> lea eax,ds 0xb048627
0x004855f <main+47> mov DWORD PTR [esp],eax
0x0048562 <main+50> call 0xb048359 <-printf@plt>
0x0048567 <main+55> mov DWORD PTR [ebp-0x4],0x1

[ source:/home/hugsy/labs/bof.c:17 ]
13 }
14
15 int main(int argc, char** argv, char** envp)
16 {
17     if (argc != 2){           ← $pc      ; argc=0x1
18         printf("Missing arg\n");
19         return 1;
20     }
21 }

[ threads ]
[#0] Id 1, Name: "bof-x32", stopped, reason: BREAKPOINT

[ trace ]
[#0] RetAddr: 0xb04854f, Name: main(argc=0x1,argv=0xffffd444,envp=0xffffdd4c)

```

```
gef> tmux-setup
```

trace-run 命令

它将跟踪并存储执行流程中 `$pc` 所取的所有值，从当前值到作为参数提供的值。

```
gef> trace-run <address_of_last_instruction_to_trace>
```

```
gef> trace-run 0x0000000000400642
[+] Tracing from 0x400600 to 0x400642 (max depth=1)
You suck bro!!
[+] Done, logfile stored as './gef-trace-0x400600-0x400642.txt'
[+] Hint: import logfile with `ida_color_gdb_trace.py` script in IDA to visualize path
gef>
```

通过在生成的文本文件上使用脚本 `ida_color_gdb_trace.py`，它将为所采用的路径着色：

```

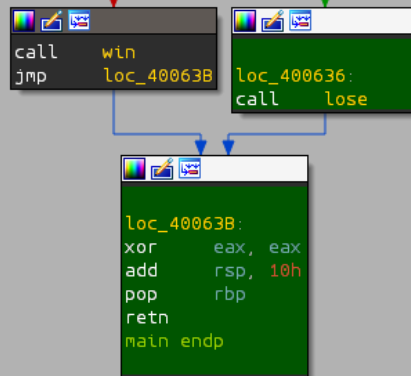
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

var_10= qword ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], 0
mov     [rbp+var_8], edi
mov     [rbp+var_10], rsi
mov     rsi, [rbp+var_10]
mov     rdi, [rsi+8]
call    sub_400560
cmp     eax, 0
jz      loc_400636

```



unicorn-emulate 命令

如果您已经安装了 [unicorn](#) 仿真引擎及其Python绑定，`gef` 会集成一个新命令来模拟当前调试环境的指令！

这个命令 `unicorn-emulate` (或它的别名 `emu`) 将为您复制当前的内存映射 (包括页面权限)，默认情况下 (即没有任何附加参数)，它将模拟指令的执行显示即将执行的 (即 `$pc` 指向的那个) 并显示哪个寄存器被它修改了。

使用 `-h` 寻求帮助

```
gef> emu -h
```

例如，以下命令将仅执行接下来的2条指令：

```
gef> emu -n 2
```

并显示：

```
$elp 0x080484db $cs 0x00000023 $ss 0x0000002b $ds 0x0000002b
$es 0x0000002b $fs 0x00000000 $gs 0x00000063 $eflags [ SF IF ]
Flags: [ carry parity adjust zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification ]
[stack]
0xffffd570 +0x00: 0xffffd7fc -> "AAAA" ← $sp
0xffffd574 +0x04: 0x08048592 -> "gef_rulez"
0xffffd578 +0x08: 0xffffffff
0xffffd57c +0x0c: 0xffffd624 -> 0xffffd7e3 -> "/home/hugsy/code/gef/wln"
0xffffd580 +0x10: 0x2
0xffffd584 +0x14: 0x0
0xffffd588 +0x18: 0x0
0xffffd58c +0x1c: 0xf7e1870e -> <__libc_start_main+222>: add esp,0x10
0xffffd590 +0x20: 0x2
0xffffd594 +0x24: 0xffffd624 -> 0xffffd7e3 -> "/home/hugsy/code/gef/wln"
[code:1386]
0x80484cc <main+60> call 0x8048450 <win>
0x80484d1 <main+65> jmp 0x80484db <main+75>
0x80484d6 <main+70> call 0x8048470 <lose>
0x80484db <main+75> xor eax,eax ← $pc
0x80484dd <main+77> add esp,0x18
0x80484e0 <main+80> pop ebp
0x80484e1 <main+81> ret
0x80484e2 xchg ax,ax
0x80484e4 xchg ax,ax
0x80484e6 xchg ax,ax
0x80484e8 xchg ax,ax
0x80484ea xchg ax,ax
[trace]
#0 0x080484db in main ()
Breakpoint 5, 0x080484db in main ()
gef> emu -n 2
[+] Starting emulation: 0x80484db -> 0x80484e0
[+] Emulation ended, showing tainted registers:
[+] $eax : old=0x000000000000000f || new=0x0000000000000000
[+] $esp : old=0x00000000ffffd570 || new=0x00000000ffffd588
[+] $eip : old=0x00000000080484db || new=0x00000000080484e0
[+] $eflags : old=[ carry parity adjust zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification ]
new=[ carry PARITY adjust zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification ]
gef> |
```

在这个例子中，我们可以看到执行后的结果

```
0x80484db <main+75> xor eax,eax
0x80484dd <main+77> add esp,0x18
```

寄存器 `eax` 和 `esp` 被修改。

一个方便的选项是 `-o /path/to/file.py`，它将生成一个嵌入当前执行上下文的纯Python脚本，可以在 `gef` 之外重用！这对于处理混淆或解决使用SMT搭建的Crackme非常有用。

vmmap 命令

`vmmap` 显示整个内存空间映射。

```
0xffffdca0: 0x1
0xffffdca4: 0xffffddc4 -> 0xffffdeb3 -> "/bin/ip"
-----[code]
=> 0x14224 <main+4>: ld [ %i1 ], %i5
0x14228 <main+8>: mov 0x2f, %o1
0x1422c <main+12>: call 0x58700 <strchr@plt>
0x14230 <main+16>: mov %i5, %o0
0x14234 <main+20>: add %o0, 1, %g1
0x14238 <main+24>: cmp %o0, 0
-----[trace]
#0 0x00014224 in main ()
Temporary breakpoint 1, 0x00014224 in main ()
gef> vmmap
Start End Offset Perm Path
0x00010000 0x00048000 0x00000000 r-x /bin/ip
0x00058000 0x0005c000 0x00038000 rwx /bin/ip
0xf7dfc000 0xf7f6e000 0x00000000 r-x /lib/sparc-linux-gnu/libc-2.13.so
0xf7f6e000 0xf7f7e000 0x00172000 --- /lib/sparc-linux-gnu/libc-2.13.so
0xf7f7e000 0xf7f80000 0x00172000 r-- /lib/sparc-linux-gnu/libc-2.13.so
0xf7f80000 0xf7f84000 0x00174000 rwx /lib/sparc-linux-gnu/libc-2.13.so
0xf7f84000 0xf7f88000 0x00000000 rwx
0xf7f88000 0xf7f8c000 0x00000000 r-x /lib/sparc-linux-gnu/libdl-2.13.so
```

正如一位聪明的读者可能已经看到的，内存映射从一个架构到另一个架构不同（这是我首先开始使用 GEF 的主要原因之一）。例如，您可以了解到在SPARC体系结构上运行的ELF始终将其 `.data` 和 `heap` 部分设置为读/写/执行。

`vmmap` 接受一个参数，一个字符串来匹配结果：

```
gef> vmmap /bin/ls
Start      End      Offset   Perm Path
0x000055555554000 0x0000555555573000 0x0000000000000000 r-x  /bin/ls
0x00005555555772000 0x00005555555773000 0x0000000000001e000 r--  /bin/ls
0x00005555555773000 0x00005555555774000 0x0000000000001f000 rw-  /bin/ls
gef>
```

xfiles 命令

`xfiles` 是GDB命令的更方便的表示，`info files` 允许您按参数中给出的模式进行过滤。例如，如果您只想显示代码部分（即 `.text`）：

```
gef> xfiles .text
Start      End      Name      File
0x00000000004004c0 0x0000000000400692 .text     /home/ubuntu/malloc-test
0x00007ffff7dd7ac0 0x00007ffff7df5640 .text     /lib64/ld-linux-x86-64.so.2
0x00007ffff7ffa9a0 0x00007ffff7ffaee9 .text     /home/ubuntu/malloc-test
0x00007ffff7a2d8b0 0x00007ffff7b7ff14 .text     /lib/x86_64-linux-gnu/libc.so.6
0x00000000004004c0 0x0000000000400692 .text     /home/ubuntu/malloc-test
0x00007ffff7dd7ac0 0x00007ffff7df5640 .text     /lib64/ld-linux-x86-64.so.2
0x00007ffff7ffa9a0 0x00007ffff7ffaee9 .text     /home/ubuntu/malloc-test
0x00007ffff7a2d8b0 0x00007ffff7b7ff14 .text     /lib/x86_64-linux-gnu/libc.so.6
```

xinfo 命令

`xinfo` 命令显示作为参数给出的特定地址的所有已知信息：

```
=> 0x104dc <main+28>:  mov %o0, %g1
0x104e0 <main+32>:    st  %g1, [ %fp + -4 ]
0x104e4 <main+36>:    sethi %hi(0x10400), %g1
0x104e8 <main+40>:    or  %g1, 0x2e0, %g1      ! 0x106e0
0x104ec <main+44>:    mov  %g1, %o0
0x104f0 <main+48>:    ld   [ %fp + -4 ], %o1
-----[trace]
#0  0x000104dc in main ()
0x000104dc in main ()
gef> xinfo 0x00022008
===== [ xinfo: 0x22008 ] =====
Found 0x00022008
Page: 0x00022000->0x00044000 (size=0x22000)
Permissions: rwx
Pathname: [heap]
Offset (from page): +0x8
Inode: 0
gef> vmmap
Start      End      Offset   Perm Path
0x00010000 0x00012000 0x00000000 r-x  /root/malloc
0x00020000 0x00022000 0x00000000 rwX /root/malloc
0x00022000 0x00044000 0x00000000 rwX [heap]
0xf7e3c000 0xf7fae000 0x00000000 r-x  /lib/sparc-linux-gnu/libc-2.13.so
0xf7fae000 0xf7fbe000 0x00172000 ---  /lib/sparc-linux-gnu/libc-2.13.so
```

重要说明：出于性能原因，`gef` 会缓存某些结果。`gef` 将尝试自动刷新自己的缓存，以避免依赖已调试过程的过时信息。然而，在一些特殊的场景中，`gef` 可能无法检测到一些新事件，使其缓存部分过时。如果您发现内存映射存在不一致，则可能需要通过运行命令 `reset-cache` 强制 `gef` 刷新其缓存并获取全新数据。

xor-memory 命令

此命令用于对内存块进行异或。

它的语法是：

```
xor-memory <display|patch> <address> <size_to_read> <xor_key>
```

第一个参数（`display` 或 `patch`）是要执行的操作：

1. `display` 只显示XOR-ed内存块结果的hexdump，而不写入调试对象的内存。

```
gef> xor display $rsp 16 1337
[+] Displaying XOR-ing 0x7fff589b67f8-0x7fff589b6808 with '1337'
-----[ Original block ]-----
0x00007fff589b67f8      46 4e 40 00 00 00 00 00 00 00 00 00 00 00 00
FN@.....
-----[ XOR-ed block ]-----
0x00007fff589b67f8      55 79 53 37 13 37 13 37 13 37 13 37 13 37
uys7.7.7.7.7.7
```

2. `patch` 将使用xor-ed内容覆盖内存。

```
gef> xor patch $rsp 16 1337
[+] Patching XOR-ing 0x7fff589b67f8-0x7fff589b6808 with '1337'
gef> hexdump byte $rsp 16
0x00007fff589b67f8      55 79 53 37 13 37 13 37 13 37      uys7.7.7.7
```

版权信息

本中文手册基于[GEF - GDB Enhanced Features](#)翻译

原版手册与本中文手册均遵循MIT协议

license MIT

译者：[ERROR404](#)

校对：[phosphorus](#)

PDF下载地址

PDF下载将于全文翻译完毕后开放