GDB调试使用简介

gdb（GNU Debugger）是GNU计划开发的一款功能强大的交互式调试器，其代码受到通用公共许可证（GPL）的保护。这个调试工具的主要运行环境是字符模式（命令行界面），这使得它可以在没有图形界面的环境中进行高效的调试工作。

gdb主要的功能

- **监视程序中变量的值的变化**： 在程序执行过程中，可以实时查看和跟踪程序中变量的值，这对于理解程序的行为和查找错误非常有用。
- **设置断点**： 可以在程序的特定位置设置断点，当程序执行到这些位置时，会自动暂停执行。这样，开发者可以检查程序在断点处的数据和状态，便于进行调试。
- **单步执行代码**： gdb支持单步执行代码，即每次只执行一行代码或一条指令。这对于跟踪程序的执行流程、查找错误位置非常有帮助。
- **分析崩溃程序产生的core文件**： 当程序崩溃时，操作系统通常会生成一个core文件，其中包含了程序崩溃时的内存映像和相关信息。gdb可以加载这些core文件，并进行深入分析，帮助开发者找到导致程序崩溃的原因。

1. gdb-dashboard 安装

由于原始 GDB 命令行过于简陋，为方便调试，我们需要下载 gdb-dashboard 插件。以下是安装步骤：

步骤 1：备份原有配置

为避免新安装的配置文件覆盖原有的 .gdbinit 文件，我们需要先对其进行重命名备份。在终端中执行以下命令：

```
mv ~/.gdbinit ~/.gdbinit.bak
```

如果执行该命令时提示没有 `.gdbinit` 文件，这属于正常情况，无需担心，可以直接继续下一步操作。

步骤 2：下载配置文件

使用 wget 命令将 .gdbinit 配置文件下载到用户的家目录下。在终端中输入以下命令：

```
wget -P ~ https://gitee.com/ftutorials/gdb-dashboard/raw/master/.gdbinit
```

步骤 3：配置 GDB 信任所有目录的 .gdbinit

为了让 GDB 能够信任所有目录下的 .gdbinit 文件，我们需要进行相应的配置。下面的命令会使用 grep 工具检查 ~/.gdbinit 文件中是否已经存在 "set auto - load safe - path /" 这行配置，如果不存在，则将其追加到文件末尾。在终端中执行：

```
grep -qxF 'set auto-load safe-path /' ~/.gdbinit || echo 'set auto-load
safe-path /' >> ~/.gdbinit
```

步骤 4：（可选）启用语法高亮

若你希望在 GDB 中启用汇编和 C 语言的语法高亮功能，需要下载并安装 Pygments 工具。需要注意的是，**如果使用的是远程实验平台，平台已经预先安装好了 Pygments，你无需进行此步骤**。若是你自己的本地实验环境，需要自行安装，可在终端中执行以下命令：

```
pip install --no-cache-dir pygments
```

2. 举个栗子：在GDB中查看执行getbuf函数时的栈帧

2.1 启动GDB并加载程序

打开终端，然后启动GDB并加载你的程序。

```
gdb bufbomb
```

GDB会加载你的程序并等待你的命令。

2.2 基础命令速查表

- 基础命令

命令	功能说明	实战场景示例
break *0x401100	在地址 0x401100 设断点	拦截 phase_1 的密码检查逻辑
break phase_2	在phase_2 设断点	在运行phase_2前暂停

break phase_2 if $rax == 5	条件断点（当%rax=5时触发）	只在第5次循环时暂停
nexti (ni)	执行1条指令，跳过函数调用	快速通过 scanf 等辅助函数
stepi (si)	执行1条指令，进入函数调用	深入分析 strings_not_equal
set $rax=0	修改寄存器的值（危险但好用！）	强制让检查函数返回“通过” ✅
continue (c)	继续运行到下一个断点	快速跳转到下一个phase的检查点

- `x命令` 数据查看：

```
x/[数量][格式][单位] <地址>
```

常用格式：

格式符	说明	示例
x	十六进制	x/xw 0x402400
d	十进制	x/dw 0x402400
s	字符串	x/s 0x402400
i	汇编指令	x/5i 0x401100

常用单位：

单位符	说明	示例
b	字节（1字节）	x/8xb $rsp
h	半字（2字节）	x/4xh 0x402400
w	字（4字节）	x/3wd $rbp-0x8

g	巨字（8字节）	x/2gx 0x6032d0

- 逆向工程专用命令:

命令	功能说明	实战场景示例
disas phase_1	反汇编指定函数	分析 phase_1 的密码检查逻辑
backtrace	查看函数调用栈	定位程序崩溃时的调用路径
info registers	查看所有寄存器值	检查输入是否存入了 %rdi
info frame	查看当前栈帧详细信息	分析局部变量和参数布局
info functions regex	搜索匹配函数名	找隐藏函数 secret_phase
info break	查看所有断点	管理多个断点

- 退出命令：

命令/快捷键	功能说明	场景示例
quit (q)	正常退出GDB	完成调试后安全退出
Ctrl + D	快速强制退出	紧急终止卡死的调试会话
Ctrl + C → q	中断程序后退出	程序卡在输入时强制退出

2.2 设置断点

在getbuf函数上设置一个断点，这样当程序执行到这个函数时，GDB会暂停执行。

```
(gdb) break getbuf
Breakpoint 1 at 0x...
```

0x...是getbuf函数的地址

2.3 运行程序

使用run命令启动你的程序。

```
(gdb) run -u [userid] #学号
Starting program: /path/to/myprogram

Breakpoint 1, 0x... in getbuf ()
```

程序将开始执行，当遇到getbuf函数的断点时，GDB会暂停执行。

2.4 查看栈帧

当程序暂停在断点上时，你可以使用backtrace（或简写为bt）命令查看当前栈帧。

```
(gdb) bt
#0  0x... in getbuf ()
#1  0x... in test ()
#2  0x... in launch ()
#3  0x... in launcher ()
#4  0x... in main ()
```

这里，#0表示当前的栈帧，即getbuf函数。

2.5 查看栈帧中的详细信息

```
(gdb) info f
Stack level 0, frame at 0x556734f0:
 rip = 0x401c15 in getbuf; saved rip = 0x4014bb
 called by frame at 0x55673510
 Arglist at 0x556734e0, args:
 Locals at 0x556734e0, Previous frame's sp is 0x556734f0
 Saved registers:
  rbp at 0x556734e0, rip at 0x556734e8
```

可以看到当前rbp的内存地址是0x556734e0。

2.6 查看test()返回地址

test()返回地址保存在getbuf的rbp+8地址上。下面查看0x556734e8内存地址。

```
(gdb) x/10xb 0x556734e8
0x556734e8 <_reserved+1037544>: 0xbb    0x14    0x40    0x00    0x00
0x00    0x00    0x00
0x556734f0 <_reserved+1037552>: 0x00    0x00
```

可以看到test()返回地址是0x00000000004014bb。

对比查看bufbomb的反汇编程序，得知0x00000000004014bb正好是test()调用getbuf()函数后的下一条指令的地址。

```
000000000040149c <test>:
  40149c: 55                    push   %rbp
  40149d: 48 89 e5              mov    %rsp,%rbp
  4014a0: 48 83 ec 10           sub    $0x10,%rsp
  4014a4: b8 00 00 00 00        mov    $0x0,%eax
  4014a9: e8 07 05 00 00        call   4019b5 <uniqueval>
  4014ae: 89 45 f8              mov    %eax,-0x8(%rbp)
  4014b1: b8 00 00 00 00        mov    $0x0,%eax
  4014b6: e8 56 07 00 00        call   401c11 <getbuf>
  4014bb: 89 45 fc              mov    %eax,-0x4(%rbp)
  4014be: b8 00 00 00 00        mov    $0x0,%eax
  4014c3: e8 ed 04 00 00        call   4019b5 <uniqueval>
  4014c8: 8b 55 f8              mov    -0x8(%rbp),%edx
  4014cb: 39 d0                 cmp    %edx,%eax
```

2.7 继续执行或退出GDB

你可以使用continue（或简写为c）命令让程序继续执行，直到遇到下一个断点或程序结束。

```
(gdb) c
Continuing.
```

或者，你可以使用quit命令退出GDB。

```
(gdb) quit
```

这就是在GDB中查看执行getbuf函数时的栈帧的基本步骤。通过GDB的栈帧管理功能，你可以深入了解程序在运行时的调用栈情况，这对于调试和理解程序的行为非常有帮助。

3. GDB常用命令简介

以下是GDB常用命令的详细解释，供同学们参考：

3.1 启动GDB并加载程序

要使用gdb进行调试，可以在命令行中输入以下命令：

```
gdb filename
```

其中filename是你要调试的程序的文件名（不包括扩展名，如果有的话）。这将启动gdb并加载指定的程序，然后你就可以开始使用gdb的各种命令和功能进行调试了。

3.2 设置断点

断点是在程序执行过程中暂停的地方，允许你检查程序的状态。你可以使用break命令（可简写为b）来设置断点：

下面提供几种设置断点的方法：

3.2.1 在函数入口设置断点

```
(gdb) break main  # 在main函数处设置断点
```

3.2.2 在特定行号设置断点

如果你知道想在哪一行代码处暂停，你可以直接指定行号：

```
(gdb) break test.c:10  # 在test.c文件的第10行
```

3.2.3 设置条件断点

条件断点允许程序仅在满足特定条件时暂停。这对于触发某些罕见或难以重现的错误情况非常有用。

```
(gdb) break 1 if i>9
```

这里 1 是断点的编号，i>9 是想要设置的条件。

3.2.4 查看所有断点信息

在加入断点之后，可通过info break命令来查看设置断点的信息，包括编号、类型、地址等。这对于管理和理解复杂的断点设置非常有帮助。

```
(gdb) info break
```

3.3 删除、清除和禁用断点

3.3.1 删除断点

使用delete（或简写为d）命令可以删除一个或多个断点。

- 在delete命令后加入断点编号可以删除指定的断点。不指定断点号则删除所有的断点

```
(gdb) d 1        # 删除编号为1的断点
```

- delete命令后面可以跟一个范围

```
(gdb) d 1-6      # 删除编号为1~6的断点
```

3.3.2 清除断点

也可以使用clear来删除指定代码行上的断点。这对于当你想清除某一行代码上的所有断点，而不管它们的具体编号时非常有用。

```
(gdb) clear 9        # 删除第9行上的所有断点
```

同样，你也可以使用clear命令后跟一个范围来清除多个行上的断点。

3.3.3 禁用和启用断点

可以使用disable命令使某个断点暂时失效。disable命令可加断点号来禁用指定断点，否则会将所有断点禁用。断点在禁用之后可以用enable命令来恢复使用。

```
(gdb) disable 1        # 禁用编号为1的断点
(gdb) disable 1-6      # 禁用编号为1到6的断点

(gdb) enable 1         # 启用编号为1的断点
(gdb) enable 1-6       # 启用编号为1到6的断点
```

3.4 执行程序

gdb 命令 run（或简写为 r）用于启动被调试的程序。当程序开始执行时，如果之前设置了断点，那么程序会在到达第一个断点处自动暂停。这样，开发者就有机会检查程序的状态，比如变量的值、内存的内容等。

此外，run 命令后面还可以跟随程序的参数，这些参数会传递给被调试的程序。这对于调试那些需要命令行参数的程序非常有用。例如：

```
(gdb) r arg1 arg2 arg3
```

在这个例子中，arg1、arg2 和 arg3 就是传递给程序的参数。

run命令后面还可以跟随文件名：

```
(gdb) run inputfile.txt
```

inputfile.txt就是传递给程序的参数，程序会在启动后读取这个文件。

在本实验中可以这样运行bufbomb：

```
linux>gdb bufbomb
(gdb) run -u [userid] < solution-raw.txt
```

3.5 列出源代码

程序在载入到gdb之后，就可以通过列出源代码list命令（可简写为l）来查看源代码信息。

```
(gdb) list
```

默认情况下，list命令会列出当前位置附近的10行代码。如果再次输入list命令，它会继续列出接下来的10行代码（如果有的话）。

你也可以通过提供参数来指定list命令列出源文件的特定部分。例如，要列出example1.c文件的第3行到第5行，你可以这样做：

```
(gdb) list example1.c:3,5
```

注意，你需要指定文件名（如果当前不在该文件上）以及行号范围。

如果当前已经在example1.c文件中，可以不用输入文件名

```
(gdb) list 9
```

另外，如果你只提供一个行号给list命令，它会列出该行代码以及它前后的各5行代码（总共11行）。例如，要列出example1.c文件中第9行及其周围的代码，你可以这样做：

```
(gdb) list example1.c:9
```

3.6 逐步执行代码

当程序在断点处暂停时，你可以使用step或next命令来逐步执行代码。step会进入函数，而next会执行下一行代码（不进入函数）。

当step命令（通常简写为s）进入函数时，gdb通常会显示函数的入口参数、函数定义所在的源文件名以及起始行号。这有助于你理解函数调用时的上下文。

next命令（通常简写为n）也会执行当前行的代码，但如果当前行是一个函数调用，next不会进入函数内部，而是将函数调用当作一条单独的语句执行，并继续执行调用之后的下一行代

码。

```
(gdb) step  # 进入当前行的函数或执行下一行代码
(gdb) next  # 执行下一行代码（不进入函数）
```

next命令后面还可以跟一个参数，指定要执行多少次next操作。这在需要跳过多行代码时非常有用。

例如，跳过接下来的5行代码：

```
(gdb) next 5
```

3.7 检查变量和表达式

3.7.1 显示变量和表达式的值

print（或简写为p）是显示数据最常用的命令，这个命令的功能非常强大，并不限于简单地显示一个整数值。

在程序暂停时，它可以接受指定格式和变量列表作为参数，类似于C语言中的printf函数。

```
(gdb) print variable_name  # 打印变量的值
(gdb) p expression  # 计算并打印表达式的值
```

你可以指定格式来控制输出，例如：

```
(gdb) p/x variable_name    # 以十六进制格式打印变量variable_name的值
(gdb) p/d expression       # 以十进制格式打
```

3.7.2 内存检查

gdb提供了x命令来检查内存内容。x命令后面通常跟着格式说明符和地址。例如：

```
(gdb) x/10xb address      # 以十六进制格式显示从address开始的10个字节的内容
(gdb) x/4iw address       # 以4个字的整数格式显示从address开始的内容
```

格式说明符用于指定如何解释和显示内存内容，如b代表字节，w代表字，i代表整数等。

3.7.3 查看寄存器

查看单个寄存器：在print命令后加上寄存器名称（前加“$”符号）

```
(gdb) p $eip # 打印指令指针寄存器的值
```

3.7.4 查看汇编代码

gdb提供了disassemble命令来查看程序的汇编代码。你可以指定函数名来查看特定函数的汇编代码。例如：

```
(gdb) disassemble printinfo # 显示函数printinfo的汇编代码。
```

如果不指定函数名，gdb通常会显示当前停止点附近的汇编代码。

3.8 设置和使用观察点

观察点也称为数据断点，用来监测某个变量或表达式的值是否有变化，如果有变化，则暂停程序的运行，这在调试程序的过程中是非常有用

3.8.1 设置观察点

设置观察点的命令为watch，后面跟着你想要监视的变量或表达式的名称

```
(gdb) watch tol     # 设置变量tol的观察点
```

上面的命令设置了一个观察点来监视变量tol的值。当tol的值改变时，gdb会暂停程序的执行。

3.8.2 观察点触发时的输出

当观察点触发时，gdb会显示一些有用的信息，包括：

- 触发观察点的变量名。
- 变量改变前的旧值。
- 变量改变后的新值。
- 程序暂停时下一条指令的地址和指令内容。

这些信息可以帮助你理解导致变量值改变的原因，以及变量值改变时程序的状态。

3.8.3 删除观察点

如果你不再需要某个观察点，可以使用delete命令来删除它。你需要指定观察点的编号，这个编号可以在gdb的提示信息中找到。

```
(gdb) delete 1
```

上面的命令删除了编号为1的观察点。

注意事项

- 观察点可能会比普通的断点更慢，因为它们需要硬件支持，并且在某些情况下可能需要额外的内存来存储旧值。
  - 不是所有的变量都可以设置观察点，特别是那些存储在寄存器中的变量或者编译器优化掉的变量。
  - 当你退出gdb时，所有的观察点都会被自动删除。

3.9 查看调用栈

在GDB（GNU Debugger）中，调用栈（call stack）是一个重要的概念，它记录了程序执行过程中函数调用的层次关系。每个函数调用都会在调用栈上创建一个新的栈帧（stack frame），包含了函数参数、局部变量以及返回地址等信息。

当你使用GDB调试一个程序时，你可以使用backtrace或bt命令来查看当前的调用栈信息。这会显示一个列表，列出了从当前函数开始，一直到程序入口点的所有函数调用。每个栈帧都会显示函数名、参数和源代码位置（如果可用的话）。

```
(gdb) backtrace  # 显示当前调用栈的信息
```

如果你想要查看调用栈中某个特定栈帧的详细信息，你可以使用frame或f命令，后面跟上栈帧的编号。这将把当前GDB的上下文切换到指定的栈帧，使你可以查看和操作该栈帧的变量和代码。

```
(gdb) f 1
```

info frame或info f命令提供了当前栈帧的详细信息，包括函数参数、局部变量、源代码位置等。

```
(gdb) info f
```

此外，info registers命令用于显示当前CPU寄存器的状态，这可以帮助你理解程序执行时的内部状态。寄存器中可能包含调用函数的地址、被调用函数的地址等信息，但通常它们更直接地关联于CPU的当前状态和执行流程。

```
info registers
```

3.10 修改代码执行流程

在GDB中，你可以使用多种命令来控制程序的执行流程。这些命令允许你暂停、继续、跳过和跳转到代码的不同部分，以便进行调试和分析。

3.10.1 continue 或 c

continue 或简写 c 命令用于继续执行程序，直到遇到下一个断点或程序结束。在程序暂停后（例如，由于遇到断点或异常），你可以使用此命令让程序继续运行。

```
(gdb) continue  # 继续执行程序
```

3.10.2 return

return 命令用于立即退出当前函数，并返回指定的值（如果有的话）。这允许你跳过函数剩余的部分，并返回到调用该函数的地方。如果省略返回值，函数将返回其类型的默认值。

```
(gdb) return
(gdb) return <value>
```

3.10.3 until

until 命令用于执行程序，直到到达指定行号或当前循环的末尾。如果指定了行号，并且执行过程中遇到断点，GDB将停在断点处。

```
(gdb) until <line_number>
```

3.10.4 finish

finish 命令用于执行当前函数中的剩余语句，直到函数返回。然后，控制将返回到调用该函数的地方。这对于查看函数如何结束并返回值特别有用。

```
(gdb) finish
```

3.10.5 jump

jump 命令允许你改变程序的执行流程，直接跳转到指定的行号。然而，需要注意的是，jump 命令不会调整调用栈或执行任何与跳转相关的清理工作。因此，如果从一个函数内部跳转到另一个函数或代码段，可能会导致未定义的行为，包括栈溢出、内存损坏等问题。通常，最好在同一函数内部进行跳转，并且要小心处理所有可能的副作用。

```
(gdb) jump <line_number>
```

在使用这些命令时，请务必谨慎，并理解它们可能对程序状态和行为产生的影响。错误的跳转或返回可能会导致程序崩溃或产生不可预测的结果。始终确保你理解代码的逻辑和GDB的工作原理，以便有效地使用这些控制命令进行调试。

3.11 退出GDB

当你完成调试并想退出gdb时，可以使用quit命令：

```
(gdb) quit  # 退出GDB
```

通过反复使用这些命令，你可以逐步跟踪程序的执行流程，检查变量的值，理解函数调用的层次关系，并最终找到并修复程序中的错误。
