Security Programming

Lab 2.2

Wang Haoyuan

Step 1/2: 打开Ubuntu并创建debug_me.c:

那么先通过vi创建这个文件：

```
why@why:~/SP$ mkdir SP2.2
why@why:~/SP$ cd SP2.2
why@why:~/SP/SP2.2$ vi debug_me.c
```

向文件中写入示例代码：

```
#include <stdio.h>

/* print a given string and a number if a pre-determined format. */
void
print_string(int num, char* string)
{
    printf("String '%d' - '%s'\n", num, string);
}

int
main(int argc, char* argv[])
{
    int i;

    /* check for command line arguments */
    if (argc < 2) { /* 2 - 1 for program name (argv[0]) and one for a param. */
        printf("Usage: %s [ ...]\n", argv[0]);
        exit(1);
    }

    /* loop over all strings, print them one by one */
    for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
        print_string(i, argv[0]);  /* function call */
    }

    printf("Total number of strings: %d\n", i);

    return 0;
}
-- INSERT --
```

保存并退出：

```
why@why: ~/SP/SP2.2
#include <stdio.h>

/* print a given string and a number if a pre-determined format. */
void
print_string(int num, char* string)
{
    printf("String '%d' - '%s'\n", num, string);
}

int
main(int argc, char* argv[])
{
    int i;

    /* check for command line arguments */
    if (argc < 2) { /* 2 - 1 for program name (argv[0]) and one for a param. */
        printf("Usage: %s [ ...]\n", argv[0]);
        exit(1);
    }

    /* loop over all strings, print them one by one */
    for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
        print_string(i, argv[0]);  /* function call */
    }

    printf("Total number of strings: %d\n", i);

    return 0;
}
:wq
```

Step 3: 通过gcc编译该文件后，以gdb打开该文件

由于gcc与gdb在《数据库系统》课程中已经被配置到Ubuntu中，因此这里直接进行相应的操作：

1. 用gcc编译：

```
why@why:~/SP/SP2.2$ gcc -g debug_me.c -o debug_me
why@why:~/SP/SP2.2$ ls
debug_me  debug_me.c
```

用ls后发现生成了`debug_me`的可执行文件。

> 在编译时，发现代码中exit(1)是在"stdlib.h"文件中出现的，此时更改代码为如下形式后再次编译发现不再有warning：

```
why@why:~/SP/SP2.2$ gcc -g debug_me.c -o debug_me
debug_me.c: In function ‘main’:
debug_me.c:18:9: warning: implicit declaration of function ‘exit’ [-Wimplicit-function-declaration]
   18 |         exit(1);
      |         ^~~~
debug_me.c:2:1: note: include ‘<stdlib.h>’ or provide a declaration of ‘exit’
    1 | #include <stdio.h>
  +++ |+#include <stdlib.h>
    2 |
debug_me.c:18:9: warning: incompatible implicit declaration of built-in function ‘exit’ [-Wbuiltin-declaration-mismatch]
   18 |         exit(1);
      |         ^~~~
debug_me.c:18:9: note: include ‘<stdlib.h>’ or provide a declaration of ‘exit’
```

why@why: ~/SP/SP2.2

```c
#include <stdio.h>
#include <stdlib.h>

/* print a given string and a number if a pre-determined format. */
void
print_string(int num, char* string)
{
    printf("String '%d' - '%s'\n", num, string);
}

int
main(int argc, char* argv[])
{
    int i;

    /* check for command line arguments */
    if (argc < 2) { /* 2 - 1 for program name (argv[0]) and one for a param. */
        printf("Usage: %s [ ...]\n", argv[0]);
        exit(1);
    }

    /* loop over all strings, print them one by one */
    for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
        print_string(i, argv[0]);  /* function call */
    }

    printf("Total number of strings: %d\n", i);

    return 0;
"debug_me.c" 30L, 682B
```

Step 4: 用gdb对代码进行调试：

run/r

运行`gdb debug_me`即可调试：

```
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from debug_me...
(gdb) l
warning: Source file is more recent than executable.
3
4       /* print a given string and a number if a pre-determined format. */
5       void
6       print_string(int num, char* string)
7       {
8           printf("String '%d' - '%s'\n", num, string);
9       }
10
11      int
12      main(int argc, char* argv[])
(gdb) l
13      {
14          int i;
15
16          /* check for command line arguments */
17          if (argc < 2) { /* 2 - 1 for program name (argv[0]) and one for a param. */
18              printf("Usage: %s [ ...]\n", argv[0]);
19              exit(1);
20          }
21
22          /* loop over all strings, print them one by one */
(gdb)
```

先进行整体的运行（`r "hello, world" "goodbye, world"`）：

```
(gdb) r "hello, world" "goodbye, world"
Starting program: /home/why/SP/SP2.2/debug_me "hello, world" "goodbye, world"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
String '1' - 'hello, world'
String '2' - 'goodbye, world'
Total number of strings: 3
[Inferior 1 (process 11113) exited normally]
```

其中包括该程序调用的库（在Linux中，程序库设置在.so文件中），运行的结果，以及程序正常退出的状态。

break/b

我们可以通过设置断点让程序在某处中止，从而查验具体信息：

输入`b 17`，则代码在当前文件的第十七行中止。

> 也可输入b debug_me.c: 17,则代码在指定文件的第17行中止。

```
(gdb) b 17
Breakpoint 1 at 0x5555555551af: file debug_me.c, line 17.
```

输入`r`尝试运行，发现停在了第17行：

```
(gdb) r
Starting program: /home/why/SP/SP2.2/debug_me "hello, world" "goodbye, world"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=3, argv=0x7fffffffdef8) at debug_me.c:17
17          if (argc < 2) { /* 2 - 1 for program name (argv[0]) and one for a param. */
```

另外也可对函数进行断点设置，则每次调用函数时进行中止程序（相当于在函数的第一行设置断点）：

输入`b print_string`,在上述情况基础上输入`c`，发现停在了函数所在行：

```
(gdb) b print_string
Breakpoint 2 at 0x55555555517c: file debug_me.c, line 8.
(gdb) c
Continuing.

Breakpoint 2, print_string (num=1, string=0x7fffffffe194 "hello, world") at debug_me.c:8
8           printf("String '%d' - '%s'\n", num, string);
(gdb)
```

next/n && step/s

这两个操作符目的都是从断点处向下运行程序，但区别在于：

next为当前函数的逐行运行，step为整个程序的逐行运行。

换言之，如果某函数调用了其他函数，则next不会进入其他函数中，只是在所在函数中逐行运行，而step会进入其他函数逐行运行。

下面我们以示例为例进行测试：

设置断点为23并运行：

```
(gdb) b 23
Breakpoint 3 at 0x5555555551dd: file debug_me.c, line 23.
(gdb) d 1
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/why/SP/SP2.2/debug_me "hello, world" "goodbye, world"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 3, main (argc=3, argv=0x7fffffffdef8) at debug_me.c:23
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb)
```

接下来先尝试next方法，发现它只在主函数内部逐行操作，并没有跳转到`print_string`函数内部：

```
Breakpoint 3, main (argc=3, argv=0x7fffffffdef8) at debug_me.c:23
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb) n
24              print_string(i, argv[0]);  /* function call */
(gdb)
String '1' - 'hello, world'
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb)
24              print_string(i, argv[0]);  /* function call */
(gdb)
String '2' - 'goodbye, world'
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb)
27          printf("Total number of strings: %d\n", i);
(gdb)
Total number of strings: 3
29          return 0;
(gdb)
```

再尝试step方法，发现它进入了print_string函数内部：

```
Breakpoint 3, main (argc=3, argv=0x7fffffffdef8) at debug_me.c:23
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb) s
24              print_string(i, argv[0]);  /* function call */
(gdb)
print_string (num=1, string=0x7fffffffe194 "hello, world") at debug_me.c:8
8           printf("String '%d' - '%s'\n", num, string);
(gdb)
```

print/p

我们可以通过print方法来对当前状态的变量进行查验：

```
Breakpoint 3, main (argc=3, argv=0x7fffffffdef8) at debug_me.c:23
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb) n
24              print_string(i, argv[0]);  /* function call */
(gdb) p i
$1 = 1
(gdb) p argv[0]
$2 = 0x7fffffffe194 "hello, world"
(gdb) p argv[1]
$3 = 0x7fffffffe1a1 "goodbye, world"
(gdb)
```

where/frame

当一个C语言程序调用函数时，它会将函数相关信息推到函数栈中，以存储调用函数的状态。

我们可以通过where方法查看当前程序执行位置所在的函数栈位置，并且可以通过frame方法查验当前每一层函数的具体信息：

首先在主函数中查验：

```
Breakpoint 1, main (argc=3, argv=0x7fffffffdef8) at debug_me.c:23
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb) where
#0  main (argc=3, argv=0x7fffffffdef8) at debug_me.c:23
(gdb) frame 0
#0  main (argc=3, argv=0x7fffffffdef8) at debug_me.c:23
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb) frame
#0  main (argc=3, argv=0x7fffffffdef8) at debug_me.c:23
23          for (argc--,argv++,i=1 ; argc > 0; argc--,argv++,i++) {
(gdb) frame 1
No frame at level 1.
(gdb)
```

可以看到，当前函数栈中只有一层（主函数main）

然后进入print_string函数中查验：

```
(gdb) n
24              print_string(i, argv[0]);  /* function call */
(gdb) s
print_string (num=1, string=0x7fffffffe196 "hello world") at debug_me.c:8
8           printf("String '%d' - '%s'\n", num, string);
(gdb) where
#0  print_string (num=1, string=0x7fffffffe196 "hello world") at debug_me.c:8
#1  0x0000555555555203 in main (argc=2, argv=0x7fffffffdf00) at debug_me.c:24
(gdb) frame 0
#0  print_string (num=1, string=0x7fffffffe196 "hello world") at debug_me.c:8
8           printf("String '%d' - '%s'\n", num, string);
(gdb) frame 1
#1  0x0000555555555203 in main (argc=2, argv=0x7fffffffdf00) at debug_me.c:24
24              print_string(i, argv[0]);  /* function call */
```

我们可以看到，上层函数栈变更为了print_string函数。而当前语句是在0层（main）与1层（print_string）的结构位置上的。

我们也可以通过frame方法结合print方法查看不同层的变量：

```
(gdb) frame 0
#0  print_string (num=1, string=0x7fffffffe196 "hello world") at debug_me.c:8
8           printf("String '%d' - '%s'\n", num, string);
(gdb) print num
$1 = 1
(gdb) frame 1
#1  0x0000555555555203 in main (argc=2, argv=0x7fffffffdf00) at debug_me.c:24
24              print_string(i, argv[0]);  /* function call */
(gdb) print i
$2 = 1
```

需要注意的是，局部变量无法被跨函数访问。

```
(gdb) frame 0
#0  print_string (num=1, string=0x7fffffffe196 "hello world") at debug_me.c:8
8           printf("String '%d' - '%s'\n", num, string);
(gdb) print i
No symbol "i" in current context.
```

小结

本实验中，我们在Linux环境下对C语言程序进行了编写，并利用gcc与gdb编译与调试。初步掌握了gcc与gdb的使用方式，能够更高效地开发C语言程序。