Stack Safety

尹嘉权 3120000419 计科1205

问题描述

GCC能在函数调用的时候插入代码做堆栈安全检查，尝试如何编译（包括代码和编译选项）能影响这个代码的产生， 对编译结果的汇编代码做出分析，指出这个安全检查的实现机制。

分析过程

首先我们可以编写代码：

test.c

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

int main() {
    int i = 12;
    char buffer[4] = {0};

    buffer[0] = '0';
    buffer[1] = '1';
    buffer[2] = '2';
    buffer[3] = '3';
    buffer[4] = '4';
    buffer[5] = '5';
    buffer[6] = '6';
    buffer[7] = '7';
    buffer[8] = '8';

  //  gets(buffer);
    printf("%d\n",i);
    printf("%d\n",(int)strlen(buffer));
    return 0;
}
```

可以发现，这里面我们对buffer的char数组操作的时候溢出了，但是在编译的过程中并未检查到下标越界，能够顺利通过编译，但是运行结果如下：

```
parallels@ubuntu: ~/Desktop/PPL
parallels@ubuntu:~/Desktop/PPL$ gcc test.c
parallels@ubuntu:~/Desktop/PPL$ ./a.out
12
16
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
parallels@ubuntu:~/Desktop/PPL$
```

说明在运行的过程中，gcc编译器检查到了堆栈缓冲区溢出（stack smashing detected）。事实上，gcc在编译的时候已经为我们加入了堆栈安全检查的编译选项，具体编译选项为：-fstack-protector

```
parallels@ubuntu:~/Desktop/PPL$ gcc test.c
parallels@ubuntu:~/Desktop/PPL$ ./a.out
12
16
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
parallels@ubuntu:~/Desktop/PPL$ gcc test.c -S -o test.s
parallels@ubuntu:~/Desktop/PPL$ gcc test.c -S -o test_pro.s -fstack-protector
parallels@ubuntu:~/Desktop/PPL$ diff test.s test_pro.s
parallels@ubuntu:~/Desktop/PPL$ diff -u test.s test_pro.s
parallels@ubuntu:~/Desktop/PPL$
```

说明gcc是默认加上了堆栈安全检查的编译选项-fstack-protector，具体汇编代码会在下文分析。

然后，我们试试如果去掉这个编译选项会怎样：（加入编译选项-fno-stack-protector）

```
parallels@ubuntu:~/Desktop/PPL$ gcc test.c -fno-stack-protector
parallels@ubuntu:~/Desktop/PPL$ ./a.out
12
9
parallels@ubuntu:~/Desktop/PPL$
```

可以发现，在运行的过程中并没有报错，并没有像之前加入了堆栈安全编译选项一样会提示stack smashing detected，和terminated Aborted(core dumped)

于是，我们输出这两个编译选项下的汇编文件，对比分析，看看具体安全检查的机制：

```
parallels@ubuntu:~/Desktop/PPL$ ./a.out
12
9
parallels@ubuntu:~/Desktop/PPL$ gcc test.c -fno-stack-protector -S -o test_non_p
ro.s
parallels@ubuntu:~/Desktop/PPL$
```

打开test_pro.s 和 test_non_pro.s

test_pro.s 内容如下：

```
        .file   "test.c"
        .section        .rodata
.LC0:
        .string "%d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movl    $12, -20(%rbp)
        movl    $0, -16(%rbp)
        movb    $48, -16(%rbp)
        movb    $49, -15(%rbp)
        movb    $50, -14(%rbp)
        movb    $51, -13(%rbp)
        movb    $52, -12(%rbp)
        movb    $53, -11(%rbp)
        movb    $54, -10(%rbp)
        movb    $55, -9(%rbp)
```

```
        movb   $56, -8(%rbp)
        movl   -20(%rbp), %eax
        movl   %eax, %esi
        movl   $.LC0, %edi
        movl   $0, %eax
        call   printf
        leaq   -16(%rbp), %rax
        movq   %rax, %rdi
        call   strlen
        movl   %eax, %esi
        movl   $.LC0, %edi
        movl   $0, %eax
        call   printf
        movl   $0, %eax
        movq   -8(%rbp), %rdx
        xorq   %fs:40, %rdx
        je     .L3
        call   __stack_chk_fail
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size  main, .-main
        .ident "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
        .section       .note.GNU-stack,"",@progbits
```

test_non_pro.s 内容如下:

```
        .file  "test.c"
        .section       .rodata
.LC0:
        .string "%d\n"
        .text
        .globl main
        .type  main, @function
main:
.LFB0:
        .cfi_startproc
        pushq  %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq   %rsp, %rbp
        .cfi_def_cfa_register 6
        subq   $16, %rsp
        movl   $12, -4(%rbp)
        movl   $0, -16(%rbp)
        movb   $48, -16(%rbp)
        movb   $49, -15(%rbp)
        movb   $50, -14(%rbp)
        movb   $51, -13(%rbp)
        movb   $52, -12(%rbp)
        movb   $53, -11(%rbp)
        movb   $54, -10(%rbp)
```

```
        movb   $55, -9(%rbp)
        movb   $56, -8(%rbp)
        movl   -4(%rbp), %eax
        movl   %eax, %esi
        movl   $.LC0, %edi
        movl   $0, %eax
        call   printf
        leaq   -16(%rbp), %rax
        movq   %rax, %rdi
        call   strlen
        movl   %eax, %esi
        movl   $.LC0, %edi
        movl   $0, %eax
        call   printf
        movl   $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
        .section        .note.GNU-stack,"",@progbits
```

对比发现，主要区别在这几行：

在test_pro.s中

```
subq    $32, %rsp
movq    %fs:40, %rax
movq    %rax, -8(%rbp)
xorl    %eax, %eax
movl    $12, -20(%rbp)
movl    $0, -16(%rbp)
movb    $48, -16(%rbp)
movb    $49, -15(%rbp)
movb    $50, -14(%rbp)
movb    $51, -13(%rbp)
movb    $52, -12(%rbp)
movb    $53, -11(%rbp)
movb    $54, -10(%rbp)
movb    $55, -9(%rbp)
movb    $56, -8(%rbp)
movl    -20(%rbp), %eax
movl    %eax, %esi
movl    $.LC0, %edi
movl    $0, %eax
call    printf
```

test_non_pro.s中

```
subq    $16, %rsp
movl    $12, -4(%rbp)
movl    $0, -16(%rbp)
movb    $48, -16(%rbp)
movb    $49, -15(%rbp)
movb    $50, -14(%rbp)
movb    $51, -13(%rbp)
movb    $52, -12(%rbp)
movb    $53, -11(%rbp)
movb    $54, -10(%rbp)
movb    $55, -9(%rbp)
movb    $56, -8(%rbp)
movl    -4(%rbp), %eax
movl    %eax, %esi
movl    $.LC0, %edi
movl    $0, %eax
call    printf
```

其中，在带有堆栈安全检查的汇编代码中，变量i=12这一行在被执行的时候，局部变量i被存放在-20(%rbp)中，而buffer被存放在-16(%rbp)中，正常来说，由于定义的时候buffer的char数组大小是4，所以对buffer的操作应该在-16(%rbp)到-13(%rbp)，由于现在我们人为地对buffer做了溢出下标的操作，所以在汇编代码里面同样会相对应的赋值，只是顺着-13(%rbp)之后的地址区域进行赋值/操作。

在不带有堆栈安全检查的汇编代码中，变量i被放在-4(%rbp)中，而buffer同样是被放在-16(%rbp)中，说明两者最大的不同就是，带有安全检查的汇编代码，字符数组放在其他变量之后，堆栈高地址存放int,float这类的变量，低地址存放字符数组；而在不带堆栈安全检查的汇编代码中，堆栈高地址存放字符数组，堆栈低地址存放int,float这类非字符数组变量。

另外，值得指出的是，在带有堆栈安全保护的汇编代码中，最后函数结束之前(leave和ret之前)，会有堆栈安全检查（在-fno-stack-protector中没有）：

```
        movq    -8(%rbp), %rdx
        xorq    %fs:40, %rdx
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

加上前面的分析，我们可以看到，在带有堆栈安全保护编译选项的代码中：

开始的三行：

```
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
```

和最后的四行：

```
        movq    -8(%rbp), %rdx
        xorq    %fs:40, %rdx
        je      .L3
        call    __stack_chk_fail
```

这几行就是堆栈保护的关键所在，或者是是SSP堆栈保护（stack smashing protection），通过这几行代码在函数栈框中插入了一个canary word，之后通过检查这个canary word来检查函数栈是否被破坏。

其中%fs:40中保存的是一个随机数，开始的三行把这个随机数放入到-8(%rbp)中，最后的四行把这个存在-8(%rbp)中的随机数取出来，与一开始的随机数比较。如果不相等，说明函数执行过程中函数堆栈被破坏，这时候就会转跳到__stack_chk_fail 来输出错误信息，中止程序的执行（即Aborted），如果相等，那么就继续执行后面的代码。

这里面i的值没有变，说明溢出的堆栈还没有影响到变量i存放的地址的值，我们可以修改之前的代码，来观察如果不加入堆栈安全保护的代码中，变量i的值；同时修改为函数，在该函数执行完毕之后，在main()之后做一定的输出

修改代码如下：

test.c

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

int gao() {
    int i = 12;
    char buffer[4] = {0};
/*
    buffer[0] = '0';
    buffer[1] = '1';
    buffer[2] = '2';
    buffer[3] = '3';
    buffer[4] = '4';
    buffer[5] = '5';
    buffer[6] = '6';
    buffer[7] = '7';
    buffer[8] = '8';
*/
    gets(buffer);
    printf("%d\n",i);
    printf("%d\n",(int)strlen(buffer));
    return 0;
}

int main(){
    printf("before gao()\n");
    gao();
    printf("after gao()\n");
    return 0;
}
```

执行结果如下：

parallels@ubuntu: ~/Desktop/PPL

```
parallels@ubuntu:~/Desktop/PPL$ gcc test.c -fno-stack-protector
test.c: In function ‘gao’:
test.c:18:5: warning: ‘gets’ is deprecated (declared at /usr/include/stdio.h:638
) [-Wdeprecated-declarations]
     gets(buffer);
     ^
/tmp/cczhWjV1.o: In function `gao':
test.c:(.text+0x1e): warning: the `gets' function is dangerous and should not be
 used.
parallels@ubuntu:~/Desktop/PPL$ ./a.out
before gao()
BrokenBuffer
0
12
after gao()
parallels@ubuntu:~/Desktop/PPL$ gcc test.c
test.c: In function ‘gao’:
test.c:18:5: warning: ‘gets’ is deprecated (declared at /usr/include/stdio.h:638
) [-Wdeprecated-declarations]
     gets(buffer);
     ^
/tmp/ccL5nZKL.o: In function `gao':
test.c:(.text+0x2d): warning: the `gets' function is dangerous and should not be
 used.
parallels@ubuntu:~/Desktop/PPL$ ./a.out
before gao()
BrokenBuffer
12
12
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
parallels@ubuntu:~/Desktop/PPL$
```

可以看到几个事实：
0、编译的时候，编译器提示gets是一个不安全的函数，它可能会导致堆栈溢出等安全问题（事实上我们就是要让它溢出）
1、在-fno-stack-protector下，程序并没有检查到堆栈溢出，而是在gets()了非法长度的字符串之后继续执行，同时也成功执行完了整个main()
2、在-fno-stack-protector下，变量i的值由12变成了0，看到输入BrokenBuffer之后的第一个输出
3、在-fstack-protector(gcc默认开启)下，变量i的值并没有因为buffer字符数组堆栈溢出而修改，输出的第一个数字是12，说明真正执行了堆栈安全保护，另外，我们也看不到执行完gao()之后的main()下的输出，即after gao()，而是在离开gao()这个函数之后检查到堆栈安全问题，之后直接中止了程序，同时输出错误信息，所以按照之前分析的，”after gao()”这个输出并没有打印到terminal上，符合分析。