04 | 深入理解栈：从CPU和函数的视角看栈的管理

2021-11-01 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 20:03 大小 18.38M

你好，我是海纳。

上节课，我们讲到，栈被操作系统安排在进程的高地址处，它是向下增长的。但这只是对栈相关知识的“浅尝辄止”。那我们今天这节课，就会跟着前面的脉络，让你可以更深刻地理解栈的运行原理。

栈是每一个程序员都很熟悉的话题，但你敢说你真的完全了解它吗？我相信，你在工作肯定遇到过栈溢出（StackOverflow）的错误，比如在写递归函数的时候，当漏掉退出条件，或者退出条件不小心写错了，就会出现栈溢出错误。我们也经常听说缓冲区溢出带来的严重的安全问题，这在日常的工作中都是要避免的。

所以，今天这节课，我们继续深入探讨一下栈这个话题，我会带你基于“符合人的直观思维”，也就是函数的层面和 CPU 的机器指令层面，多角度来理解栈相关的概念。这样，你以后遇到与栈相关的问题的时候，才知道如何着手进行排查。最后，我们还会通过一个缓冲区溢出攻击栈的案例，看看我们在日常工作中如何提升代码的健壮度和安全性。

函数与栈帧

当我们在调用一个函数的时候，CPU 会在栈空间（这当然是线性空间的一部分）里开辟一小块区域，这个函数的局部变量都在这一小块区域里存活。当函数调用结束的时候，这一小块区域里的局部变量就会被回收。

这一小块区域很像一个框子，所以大家就命名它为 stack frame。frame 本意是框子的意思，在翻译的时候被译为帧，现在它的中文名字就是栈帧了。

所以，我们可以说，**栈帧本质上是一个函数的活动记录。**当某个函数正在执行时，它的活动记录就会存在，当函数执行结束时，活动记录也被销毁。

不过，你要注意的是，在一个函数执行的时候，它可以调用其他函数，这个时候它的栈帧还是存在的。例如，A 函数调用 B 函数，此时 A 的栈帧不会被销毁，而是会在 A 栈帧的下方，再去创建 B 函数的栈帧。只有当 B 函数执行完了，B 的栈帧也被销毁了，CPU 才会回到 A 的栈帧里继续执行。

我们举个例子说明一下，就很好理解了。你可以看一下这个代码：

复制代码

```
#include <stdio.h>

void swap(int a, int b) {
    int t = a;
    a = b;
    b = t;
}

void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
    printf("a is %d, b is %d\n", a, b);
}
```

你可以看到，在 swap 函数中，a 和 b 的值做了一次交换，但是在 main 函数里，打印 a 和 b 的值，a 还是 2，b 还是 3。这是为什么呢？从栈帧的角度，这个问题就非常容易理解：

main	
a	2
b	3

main	
a	2
b	3
swap	
a	3
b	2
t	2

在 main 函数执行的时候，main 的栈帧里存在变量 a 和 b。当 main 在调用 swap 方法的时候，会在 main 的帧下面新建 swap 的栈帧。swap 的帧里也有局部变量 a 和 b，但是明显这个 a、b 与 main 函数里的 a、b 没有任何关系，不管对 swap 的帧里的 a/b 变量做任何操作都不会影响 main 函数的栈帧。

接下来，我们再通过一个递归的例子来加深对栈的理解。由于递归执行的过程会出现函数自己调用自己的情况，也就是说，一个函数会对应多个同时活跃的记录（即栈帧）。所以，理解了递归函数的执行过程，我们就能更加深刻地理解栈帧与函数的关系。

当我们在谈递归时，我们在谈什么

我们先看一下最经典的递归问题：**汉诺塔**。汉诺塔问题是这样描述的：有三根柱子，记为 A、B、C，其中 A 柱子上有 n 个盘子，从上到下的编号依次为 1 到 n，且上面的盘子一定比下面的盘子小。要求一次只能移动一只盘子，且大的盘子不能压在小的盘子上，那么将所有盘子从 A 移到 C 总共需要多少步？

这道题的详细分析过程是一种递归推导的过程，不是我们这节课的重点，如果你对解法感兴趣的话，可以自己查找相关资料。我们这里，重点来讲解递归程序执行的过程中，栈是怎么样变化的，这样可以帮助我们理解栈的基本工作原理。

你先看一下汉诺塔问题的求解程序：

复制代码

```
1  #include <stdio.h>
2
3  void move(char src, char dst, int n) {
4      printf("move plate %d form %c to %c\n", n, src, dst);
5  }
6
7  void hanoi(char src, char dst, char aux, int n) {
8      if (n == 1) {
9          move(src, dst, 1);
10         return;
11     }
12
13     hanoi(src, aux, dst, n-1);
14     move(src, dst, n);
15     hanoi(aux, dst, src, n-1);
16 }
17
18 int main() {
19     hanoi('A', 'C', 'B', 5);
20 }
```

这段代码可以打印出借由 B 柱子将 5 个盘子从 A 搬移到 C 的所有步骤。这个的核心是 hanoi 函数，在深入分析代码的执行过程之前，我们可以先从符合直观思维的角度尝试理解 hanoi 函数。

hanoi 函数有四个参数。第一个 src 代表要搬的起始柱子（开始时是 A），第二个代表目标柱子（开始时是 C），第三个代表可以借用的中间的那个柱子（开始时是 B），第四个参数代表一共要搬的盘子总数（开始时是 5）。

代码的第 13 行的意义是，如果要从 A 搬 5 个盘子到 C，可以先将 4 个盘子搬到 B 上，然后第 14 行代表将第 5 个盘子从 A 搬到 C，第 15 行代表把 B 上面的 4 个盘子搬到 C 上去。第 8 行的判断是说当只搬一个盘子的时候，就可以直接调用 move 方法。

以上就是递归程序的设计思路。下面我们再具体分析这个代码的执行过程。为了简便起见，我们选择 n=3 进行分析。

hanoi(A, C, B, 3)	
src	A
dst	C
aux	B
n	3
line	13

(a)

hanoi(A, C, B, 3)	
src	A
dst	C
aux	B
n	3
line	13
hanoi(A, B, C, 2)	
src	A
dst	B
aux	C
n	2
line	13

(b)

hanoi(A, C, B, 3)	
src	A
dst	C
aux	B
n	3
line	13
hanoi(A, B, C, 2)	
src	A
dst	B
aux	C
n	2
line	13
hanoi(A, C, B,1)	
src	A
dst	C
aux	B
n	1
line	9

(c)

可以看到，当程序在执行 hanoi(A, C, B, 3) 时，CPU 会为其创建一个栈帧，这一帧里记录着变量 src、dst、aux 和 n。

此时 n 为 3，所以，代码可以执行到第 13 行，然后就会调用执行 hanoi(A, B, C, 2)。这代表着将 2 个盘子从 A 搬到 B，同样 CPU 也会为这次调用创建一个栈帧；当这一次调用执行到第 13 行时，会再调用执行 hanoi(A, C, B, 1)，代表把一个盘子从 A 搬到 C。不过，由于这一次调用 n 为 1，所以会直接调用 move 函数，打印第一个步骤“把盘子 1 从 A 搬到 C”。

接下来，程序就会回到 hanoi(A, B, C, 2) 的栈帧，继续执行第 14 行，打印第二个步骤”把盘子 2 从 A 搬到 B”。然后再执行第 15 行，也就是执行 hanoi(C, B, A, 1)。这一步的栈帧变化，你可以看下面这张图。

hanoi(A, C, B, 3)	
src	A
dst	C
aux	B
n	3
line	13
hanoi(A, B, C, 2)	
src	A
dst	B
aux	C
n	2
line	15
hanoi(C, B, A, 1)	
src	C
dst	B
aux	A
n	1
line	9

(a)

hanoi(A, C, B, 3)	
src	A
dst	C
aux	B
n	3
line	15

(b)

hanoi(A, C, B, 3)	
src	A
dst	C
aux	B
n	3
line	15
hanoi(B, C, A, 2)	
src	B
dst	C
aux	A
n	2
line	13
hanoi(B, A, C, 1)	
src	B
dst	A
aux	C
n	1
line	9

(c)

极客时间

我们看到，在调用 hanoi(C, B, A, 1) 的时候，由于 n 等于 1，所以就会打印第三个步骤“把盘子 1 从 C 搬到 B”，此时 hanoi(C, B, A, 1) 就执行完了。

那么接下来，程序就退回到 hanoi(A, B, C, 2) 的第 15 行的下一行继续执行，也就是函数的结束，这就意味着 hanoi(A, B, C, 2) 也执行完了。这个时候，程序就会回退到最高的一层 hanoi(A, C, B, 3) 的第 14 行继续执行。这一次就打印了第四个步骤“把盘子 3 从 A 搬到 C”，此时的栈帧如上图 (b) 所示。

然后，程序会执行第 15 行，再次进入递归调用，创建 hanoi(B, C, A, 2) 的栈帧。当它执行到第 13 行时，就会再创建 hanoi(B, A, C, 1) 的栈帧，此时栈的结构如上图（c）所示。由于 n 等于 1，这一次调用就会打印第五个步骤“把盘子 1 从 B 搬到 A”。

再接着就开始退栈了，回到 hanoi(B, C, A, 2) 的栈帧，继续执行第 14 行，打印第六个步骤“把盘子 2 从 B 搬到 C”。然后执行第 15 行，也就是 hanoi(A, C, B, 1)，此时 n 等于 1，直接打印第七个步骤“把盘子 1 从 A 搬到 C”。接下来就执行退栈，这一次每一个栈帧都执行到了最后一行，所以会一直退到 main 函数的栈帧中去。退栈的过程比较简单，你自己思考一下就好了。

这样我们就完成了一次汉诺塔的求解过程。在这个过程中呢，我们观察到，**先创建的帧最后才销毁，后创建的帧最先被销毁**，这就是**先入后出**的规律，也是程序执行时的活跃记录

要被叫做栈的原因。

那么在这里呢，我还想让你做一个小练习。我想让你试着用我们上面分析栈变化的方法，来分析使用深度优先算法打印全排列的程序，这会让你更加深入地理解栈的运行规律，同时掌握深度优先算法的递归写法。

复制代码

```
res = []

def make(n, level):
    if n == level:
        print(res)
        return

    for i in range(1, n+1):
        if i not in res:
            res.append(i)
            make(n, level+1)
            res.pop()

make(3, 0)
```

从指令的角度理解栈

好了，前面递归的例子，是从人的直观思维的角度去理解栈，但是在 CPU 层面，机器指令又是怎样去理解栈的呢？我们还是通过一个例子来考察一下：

复制代码

```
int fac(int n) {
    return n == 1 ? 1 : n * fac(n-1);
}
```

这是一个使用递归的写法求阶乘的例子，源码是比较简单的，我们可以使用 gcc 对其进行编译，然后使用 objdump 对其反编译，观察它编译后的机器码。

复制代码

```
# gcc -o fac fac.c
# objdump -d fac
```

然后你可以得到以下输出：

复制代码

```
1   40052d:       55                      push   %rbp
2   40052e:       48 89 e5                mov    %rsp,%rbp
3   400531:       48 83 ec 10             sub    $0x10,%rsp
4   400535:       89 7d fc                mov    %edi,-0x4(%rbp)
5   400538:       83 7d fc 01             cmpl   $0x1,-0x4(%rbp)
6   40053c:       74 13                   je     400551 <fac+0x24>
7   40053e:       8b 45 fc                mov    -0x4(%rbp),%eax
8   400541:       83 e8 01                sub    $0x1,%eax
9   400544:       89 c7                   mov    %eax,%edi
10  400546:       e8 e2 ff ff ff          callq  40052d <fac>
11  40054b:       0f af 45 fc             imul   -0x4(%rbp),%eax
12  40054f:       eb 05                   jmp    400556 <fac+0x29>
13  400551:       b8 01 00 00 00          mov    $0x1,%eax
14  400556:       c9                      leaveq
15  400557:       c3                      retq
```

我们来分析一下这段汇编代码。

第 1 行是将当前栈基址指针存到栈顶，第 2 行是把栈指针保存到栈基址寄存器，这两行的作用是把当前函数的栈帧创建在调用者的栈帧之下。保存调用者的栈基址是为了在 return 时可以恢复这个寄存器。

第 3 行的作用呢，是把栈向下增长 0x10，这是为了给局部变量预留空间。从这里，你可以看出来运行 fac 函数要是消耗栈空间的。

试想一下，如果我们不加 n==1 的判断，那么 fac 函数将无法正常返回，会出现一直递归调用回不来的情况，这样栈上就会出现很多 fac 的帧栈，会造成栈空间耗尽，出现 StackOverflow。这里的原理是，操作系统会在栈空间的尾部设置一个禁止读写的页，一旦栈增长到尾部，操作系统就可以通过中断探知程序在访问栈末端。

第 4 行是把变量 n 存到栈上。其中变量 n 一开始是存储在寄存器 edi 中的，存储的目标地址是栈基址加上 0x4 的位置，也就是这个函数栈帧的第一个局部变量的位置。变量 n 在寄存器 edi 中是 X86 的 ABI 决定的，第一个整型参数一定要使用 edi 来传递。

第 5 行将变量 n 与常量 0x1 进行比较。在第 6 行，如果比较的结果是相等的，那么程序就会跳转到 0x400551 位置继续执行。我们看到，在这块代码里，0x400551 是第 13 行，它把 0x1 送到寄存器 eax 中，然后返回，就是说当 n==1 时，返回值为 1。

如果第 5 行的比较结果是不相等的，又会怎么办呢？那第 6 行就不会跳转，而是继续执行第 7 行。7、8、9 这三行的作用，就是把 n-1 送到 edi 寄存器中，也就是说以 n-1 为参数调用 fac 函数。这个时候，调用的返回值在 eax 中，第 11 行会把返回值与变量 n 相乘，结果仍然存储在 eax 中。然后程序就可以跳转到 0x400556 处结束这次调用。

理解了 fac 函数的汇编指令以后，我们再重点讨论 callq 指令。

执行 callq 指令时，CPU 会把 rip 寄存器中的内容，也就是 call 的下一条指令的地址放到栈上（在这个例子中就是 0x40054b），然后跳转到目标函数处执行。当目标函数执行完成后，会执行 ret 指令，这个指令会从栈上找到刚才存的那条指令，然后继续恢复执行。

栈空间中的 rbp、rsp，以及返回时所用的指令都是非常敏感的数据，一旦被破坏就会造成不可估量的损失。

不过，你在重现这个例子一定要注意，我们使用不同的优化等级，产生的汇编代码也是不同的。比如如果你用以下命令进行编译，得到的二进制文件中将不再使用 rbp 寄存器。

复制代码

```
1 # gcc -O1 -o fac fac.c
```

至于这个结果，我这里就不再展示了，我想让你自己动手试一下，然后在留言区和我们分享。

到这里，我们已经从人的大脑的理解角度和机器指令的角度，让你加深了对栈和栈帧的理解。现在，我们就从理论转向实操，举一个通过缓冲区溢出来破坏栈的例子。通过这个例子，你就知道在平时的工作中，应该如何避免写出被黑客攻击的不安全代码。

栈溢出

下面这个测试是我精心构造的例子。因为是演示用的，所以我就把各种无关的代码去掉了，只保留了关键路径上的代码。你先看一下代码：

复制代码

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

#define BUFFER_LEN 24

void bad() {
    printf("Haha, I am hacked.\n");
    exit(1);
}

void copy(char* dst, char* src, int n) {
    int i;
    for (i = 0; i < n; i++) {
        dst[i] = src[i];
    }
}

void test(char* t, int n) {
    char s[16];
    copy(s, t, n);
}

int main() {
    char t[BUFFER_LEN] = {
        'w', 'o', 'l', 'd',
        'a', 'b', 'a', 'b', 'a', 'b',
        'a', 'b', 'a', 'b', 'a', 'b',
    };
    int n = BUFFER_LEN - 8;
    int i = 0;
    for (; i < 8; i++) {
        t[n+i] = (char)((((long)(&bad)) >> (i*8)) & 0xff);
    }

    test(t, BUFFER_LEN);
    printf("hello\n");
}
```

你可以用 gcc 编译器来编译上面这个程序：

复制代码

```
gcc -O1 -o bad bad.c -g -fno-stack-protector
```

执行它，你可以看到，虽然在 main 函数里我们并没有调用 bad 函数，但它却执行了。最后运行结果是“Haha, I am hacked”。

我们首先来分析一下，这个程序为什么会有这样的运行结果。

当我们在调用 test 函数的时候，会把返回地址，也就是 rip 寄存器中的值，放到栈上，然后就进入了 test 的栈帧，CPU 接着就开始执行 test 函数了。

test 函数在执行时，会先在自己的栈帧里创建数组 s，数组 s 的长度是 16。此时，栈上的布局是这样的：

地址	存储的值
16	old eip，由call指令压栈，是攻击的目标
0	变量s，占据16字节空间

通过计算，我们可以知道返回地址是变量 s 的地址 + 16 的地方，**这就是我们要攻击的目标**。我们只要在这个地方把原来的地址替换为函数 bad 的入口地址（第 26 至 34 行所做的事情），就可以改变程序的执行顺序，实现了一次**缓冲区溢出**。

简单地说，数组 s 的长度是 16，理论上我们只能修改以 s 的地址开始、长度为 16 的数据。但是我们现在通过 copy 函数操作了大于 16 的数据，从而破坏了栈上的关键数据。也就是说我们针对函数调用的返回地址发起了一次攻击。所以，test 函数的实现是不安全的。

其实这种缓冲区溢出，就是指通过一定的手段，来达成修改不属于本函数栈帧的变量的目的，而这种手段多是通过往字符串变量或者数组中写入错误的值而造成的。

有两种常见的手段可以对这一类攻击进行防御。

第一，对入参进行检查，尽量使用 strncpy 来代替 strcpy。因为 strcpy 不对参数长度做限制，而 strncpy 则会做检查。比如上述例子中，如果我们对参数 n 做检查，要求它的值必须大于 0 且小于缓冲区长度，就可以阻击缓冲区溢出攻击了。

第二，可以使用 gcc 自带的栈保护机制，这就是 -fstack-protector 选项。你查 gcc 手册（在 Linux 系统使用“man gcc”就能查到）可以看到它的一些相关信息。

当 -fstack-protector 启用时，当其检测到缓冲区溢出时 (例如，缓冲区溢出攻击）时会立即终止正在执行的程序，并提示其检测到缓冲区存在的溢出的问题。这种机制是通过在函数中的易被受到攻击的目标上下文添加保护变量来完成的。这些函数包括使用了 alloca 函数以及缓冲区大小超过 8bytes 的函数。这些保护变量在进入函数的时候进行初始化，当函数退出时进行检测，如果某些变量检测失败，那么会打印出错误提示信息并且终止当前的进程。

从 4.8 版本开始，gcc 中的这个选项是默认打开的。如果我们在编译时，不加 -fno-stack-protector，gcc 就会给可执行程序增加栈保护的功能。这样的话，运行结果就会出现 Segment Fault，导致进程崩溃。不过，你要知道，在遇到攻击时自己崩溃，相比起去执行攻击者的恶意代码，影响可就小多了。

这里，我们为了演示的方便，使用 -fno-stack-protector 关闭了这个选项。不过，在日常开发中，这个选项虽然使得栈的安全大大加强了，但它也有巨大的性能损耗。在一个实际的线上例子中，关闭这个选项可以提升 8% 至 10% 的性能。

当然这个选项也不是万能的，攻击者依然能通过精心构造数据来达成它的目标。所以在写代码的时候，你还是应该对缓冲区安全多加注意。

总结

这节课，我们一起学习了栈帧的作用，并通过汉诺塔程序的求解过程，来分析了栈帧的创建和销毁的过程，以此揭示了函数和栈帧的关系。栈帧就是函数的活动记录，当函数被调用时，栈帧创建，当函数调用结束后，栈帧消失。

在程序的执行过程中，尤其是递归程序的执行过程中，你可以清楚地观察到栈帧的创建销毁，满足后入先出的规律。这也是人们把管理函数的活跃记录的区域称为栈的原因。

除了用人的直观思维来理解栈帧之外，我还带你看了在汇编代码级别，栈帧是怎么真实地被创建和销毁的，或者说栈是怎么增长和收缩的。这会进一步加深你对栈的理解。

这节课的最后，我也通过一个缓冲区溢出的例子说明了，在栈空间内使用缓冲区的时候，你必须要十分小心，要避免恶意的输入对缓冲区进行越界读写，破坏栈的结构，从而导致关键数据被修改。我们演示了一个破坏了调用者返回地址的例子，以此来说明当返回地址被破坏以后，攻击者可以让程序的控制流转向我们不希望的地方。

很多人以为安全和攻击是做安全的同事才应该关心的问题，这个想法是不对的。要想提高软件的整体水平，每一个程序员都应该写出健壮而安全的代码。只有每一块砖都足够坚固，我们才有可能建成一个安全可靠的建筑物。

思考题

我们这节课前面讲的 swap 函数的例子，是很多新手会犯的错误。在 C 语言中，为了使 swap 可以交换 main 函数里的 a/b 两个变量的值，我们可以使用指针：

复制代码

```
#include <stdio.h>

void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

void main() {
    int a = 2;
    int b = 3;
    swap(&a, &b);
    printf("a is %d, b is %d\n", a, b);
}
```

或者在 C++ 中，直接使用引用，**引用可以看成是一个能自动解引用的指针**：

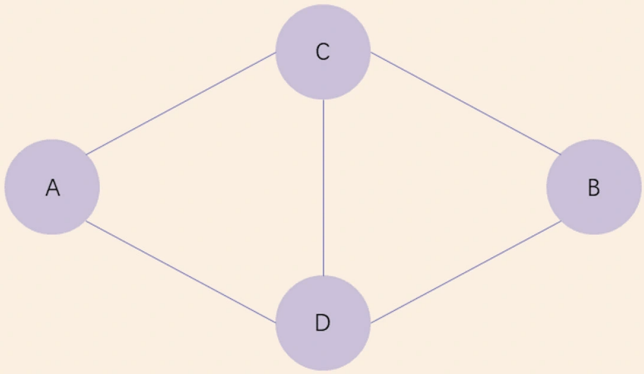
复制代码

```
#include <stdio.h>

void swap(int& a, int& b) {
    int t = a;
    a = b;
    b = t;
}

void main() {
    int a = 2;
    int b = 3;
    swap(a, b);
    printf("a is %d, b is %d\n", a, b);
}
```

那么，我想请你从汇编代码层面思考，这是怎么做到的？（提示：要从“传入栈帧的参数到底是什么”这个角度去思考）另外，如果你对 Java 程序比较熟悉，你也可以思考一下 Java 能不能实现类似的功能？

吊打面试官

- 给你一张地图，上面标注了各个城市之间是否有道路相连，请问从A城市到B城市一共有多少条路？
  要求一条路径中不能多次出现同一个城市。例如下面的图中从A到B一共有ACB、ADB、ACDB、ADCB四条路径，但ACDCB不是一个合法路径，因为这条路径中C出现了两次。



这是一道图的题目，要使用深度优先搜索（Depth First Search, DFS)来做，深度优先搜索的实现方案中，最简单直观的就是递归的写法。

这道题目与普通的DFS相比，有一个关键的技巧就在于，递归调用之前我们要把待搜索城市标成visited，当递归调用结束以后再把它的visited标记清空。要理解整个程序的运行原理，就要深刻地理解栈的运行原理。这个程序的伪代码可以这么写：

```
int count = 0;
bool visited[n] = {false,};

void search(int node) {
  if (node == TARGET) {
    count++;
    return;
  }

  for (ni in neighbors[node]) {
    if (visited[ni])
      continue;

    visited[ni] = true;
    search(ni);
    visited[ni] = false;
  }
}
```

高频面试真题

好，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 03 | 内存布局：应用程序是如何安排数据的？

下一篇 05 | 栈的魔法：从栈切换的角度理解进程和协程

精选留言 (12)

写留言

2021-11-01

老师，出堆后栈空间里的数据还是保留的啊，是不是叫栈空间扩展和收缩形象点

作者回复: 是的。可以这么说。不过我们在说expand这个词的时候，往往用于栈空间不足了，需要对栈进行动态扩展这种场景。你说的很对，rsp指针的上移并没有真的把栈上的数据清空掉，所以我们在使用局部变量的时候一定要初始化，否则就有可能访问到上次释放的栈内存。你掌握的很好！

3

费城的二鹏
2021-11-02

吊打面试官的配图很清晰，点赞！

展开

1 1

威
2021-11-01

老师您好，缓冲区溢出的Segment Fault，是指一个栈桢里溢出，还是栈桢之间的溢出。我理解是按照文章说的保护机制，应该是溢出到了别的栈桢，才会出现Segment Fault。不知道这样的理解正不正确呢？

作者回复: 往往无意识的缓冲区溢出，因为会拿随机值覆盖有效值，所以会带来segment fault，覆盖了本栈帧的关键数据，或者覆盖了其他栈帧的数据都有可能造成segment fault。但是精心构造的缓冲区溢出，就像课程里对test函数的攻击，是可以把控制流导向恶意代码的。保护机制其实是在关键位置，比如栈帧开始处，编译器自动插入变量，函数结束时再检查一下，如果变化了就主动触发fault，以增强安全性，所以不一定是溢出到别的栈帧，两者之间没有必然联系

1

keepgoing
2021-11-04

又看了一遍老师这一课，没太看懂栈溢出攻击这一块的细节，想多请教一下：

执行test函数后，字符串数组s中从0元素到15元素在栈中存储地址是从高向低的吗？随后call copy方法后，会压栈下一条指令地址到栈上，这条指令存储地址更低。所以最后让栈溢出时多拷贝地址是把地址数值的低位放在内存地址高位、数值的高位放在内存地址低位…

展开

作者回复: 是的。是这样的。就是通过把bad函数的地址写到栈上，然后就使得ret指令跑进bad函数里面运行了。两个要素：一是越界读写，一是覆盖栈上的返回地址。

coder
2021-11-04

C语言思考题：
main函数部分汇编代码：
0000000000400559 <main>:
400559: 55 push %rbp
40055a: 48 89 e5 mov %rsp,%rbp…
展开

作者回复: 是32bit，你注意看，整型变量都是使用movl 来操作的，这个 l 是 long 的缩写，代表4字节。long long类型才会占用8字节，64位，操作指令也是 movq。

keepgoing
2021-11-04

老师想提个小建议，能不能把汇编代码也贴上来比较方便理解，n*(n-1)那个例子因为示例代码只有机器码，只能看着您的文字理解，我们这种刚开始入门的同学看着可能比较抽象，不过这一课又把栈更深入地理解了一遍，谢谢老师
展开

作者回复: 汇编代码往右拖，我也看不懂机器码，哈哈

1

柒
2021-11-03

老师，我觉得你一下用python，一下用c语言，不太好。

作者回复: 其实基本上都是C语言。python呢就当伪代码看吧，你看最后结尾的吊打面试官里，其实也是伪代码。自己转换成可执行的C或者Java代码是个很好的练习哦。

鸮
2021-11-03

老师好，栈溢出的例子，在栈桢上不是先保存基地址rbp，然后分配rsp保存参数和局部变量吗？所以在参数的栈高位应该还有rbp，然后才是rip。但是代码本地运行一下是可以的，通过objdump看，发现没有push %rbp，mov %rsp，%rbp了。这是因为gcc加了-o1的优化参数。这个是不是有点类似方法内敛呢？不加-o1,就还会先保存rbp了，在执行即使段错误。...

展开

作者回复: 对的。这说明你真的动手实践过了。这就掌握得很好了。你的看法都是对的。

Rovebiy

2021-11-02

老师，是不是曾经在知乎写过专栏进击的Java新人？

展开

作者回复: 嗯，是我。

GL

2021-11-02

swap函数在C传入指针或C++的引用 是拿到了操作数的存放地址 所以可以改变对应的值，Java语言的入参如果基本数据类型是没法改变外部变量的值，如果是引用类型是可以改变引用对象内的属性值。

作者回复: Right! Java没有指向栈上的指针，这个设计很重要。

linker

2021-11-01

思考题：反汇编结果显示传递的是地址

000000000040052d <swap>:

40052d: 55 push %rbp

40052e: 48 89 e5 mov %rsp,%rbp

400531: 48 89 7d e8 mov %rdi,-0x18(%rbp)...

展开

作者回复: 请文字描述，哈哈

1

杨军

2021-11-01

这个就是csapp 中 lab2 的内容，好亲切

展开

作者回复: 赞，我不知道还有这个。我去查了一下，应该是lab3，attack。