**Linux操作系统栈溢出保护机制**

**0x00 概述**

栈溢出保护是一种缓冲区溢出攻击缓解手段，当函数存在缓冲区溢出攻击漏洞时，攻击者可以覆盖栈上的返回地址来让shellcode能够得到执行。当启用栈保护后，函数开始执行的时候会先往栈里插入cookie信息，当函数真正返回的时候会验证cookie信息是否合法，如果不合法就停止程序运行。攻击者在覆盖返回地址的时候往往也会将cookie信息给覆盖掉，导致栈保护检查失败而阻止shellcode的执行。在Linux中我们将cookie信息称为canary(*以下统一使用canary*)。

gcc在4.2版本中添加了-fstack-protector和-fstack-protector-all编译参数以支持栈保护功能，4.9新增了-fstack-protector-strong编译参数让保护的范围更广。以下是-fstack-protector和-fstack-protector-strong的区别：

|  |  |  |
| --- | --- | --- |
| 参数 | gcc支持版本 | 说明 |
| -fstack-protector | 4.2 | 只为局部变量中包含长度超过8-byte(含)的char数组的函数插入保护代码 |
| -fstack-protector-strong | 4.9 | 满足以下三个条件都会插入保护代码：  1.局部变量的地址作为赋值语句的右值或函数参数；  2.局部变量包含数组类型的局部变量，不管数组的长度；  3.带register声明的局部变量 |

**Linux系统中存在着三种类型的栈：**

1.应用程序栈：由应用程序来维护；

2.内核进程上下文栈：由内核在创建线程的时候创建；

3.内核中断上下文栈：在内核初始化的时候给每个CPU核心创建一个。

**0x01 应用程序栈保护**

**1. 栈保护工作原理**

下面是一个包含栈溢出的例子：

/\* test.c \*/  
#include <stdio.h>  
#include <string.h>  
int main(int argc, char \*\*argv)  
{  
 char buf[16];  
 scanf("%s", buf);  
 printf("%s\n", buf);  
 return 0;  
}

我们先禁用栈保护功能看看执行的结果

[root@localhost stackp]# gcc -o test test.c -fno-stack-protector  
[root@localhost stackp]# python -c "print 'A'\*24" | ./test  
AAAAAAAAAAAAAAAAAAAAAAAA  
Segmentation fault <- RIP腐败，导致异常

当返回地址被覆盖后产生了一个段错误，因为现在的返回地址已经无效了，所以现在执行的是CPU的异常处理流程。我们打开栈保护后再看看结果：

[root@localhost stackp]# gcc -o test test.c -fstack-protector  
[root@localhost stackp]# python -c "print 'A'\*25" | ./test  
AAAAAAAAAAAAAAAAAAAAAAAAA  
\*\*\* stack smashing detected \*\*\*: ./test terminated <- 提示检查到栈溢出

这时触发的就不是段错误了，而是栈保护的处理流程，我们反汇编看看做了哪些事情：

0000000000400610 <main>:  
 400610: 55 push %rbp  
 400611: 48 89 e5 mov %rsp,%rbp  
 400614: 48 83 ec 30 sub $0x30,%rsp  
 400618: 89 7d dc mov %edi,-0x24(%rbp)  
 40061b: 48 89 75 d0 mov %rsi,-0x30(%rbp)  
 40061f: 64 48 8b 04 25 28 00 mov %fs:0x28,%rax <- 插入canary值  
 400626: 00 00  
 400628: 48 89 45 f8 mov %rax,-0x8(%rbp)  
 40062c: 31 c0 xor %eax,%eax  
 40062e: 48 8d 45 e0 lea -0x20(%rbp),%rax  
 400632: 48 89 c6 mov %rax,%rsi  
 400635: bf 00 07 40 00 mov $0x400700,%edi  
 40063a: b8 00 00 00 00 mov $0x0,%eax  
 40063f: e8 cc fe ff ff callq 400510 <\_\_isoc99\_scanf@plt>  
 400644: 48 8d 45 e0 lea -0x20(%rbp),%rax  
 400648: 48 89 c7 mov %rax,%rdi  
 40064b: e8 80 fe ff ff callq 4004d0 <puts@plt>  
 400650: b8 00 00 00 00 mov $0x0,%eax  
 400655: 48 8b 55 f8 mov -0x8(%rbp),%rdx <- 检查canary值  
 400659: 64 48 33 14 25 28 00 xor %fs:0x28,%rdx  
 400660: 00 00  
 400662: 74 05 je 400669 <main+0x59> # 0x400669  
 400664: e8 77 fe ff ff callq 4004e0 <\_\_stack\_chk\_fail@plt>  
 400669: c9 leaveq  
 40066a: c3 retq

我们看到函数开头(地址：0x40061f)处gcc编译时在栈帧的返回地址和临时变量之间插入了一个canary值，该值是从%fs:0x28里取的，栈帧的布局如下：

stack:  
| ...... |  
| orig\_return |  
| orig\_rbp | <- %rbp  
| canary | <- -0x8(%rpb), 既 %fs:0x28  
| local variables |  
| | <- %rsp

在函数即将返回时(地址：0x400655)检查栈中的值是否和原来的相等，如果不相等就调用glibc的\_*stack*chk\_fail函数，并终止进程。

**2. canary值的产生**

这里以x64平台为例，canary是从%fs:0x28偏移位置获取的，%fs寄存器被glibc定义为存放tls信息的，我们需要查看glibc的源代码：

typedef struct  
{  
 void \*tcb; /\* Pointer to the TCB. Not necessarily the  
 thread descriptor used by libpthread. \*/  
 dtv\_t \*dtv;  
 void \*self; /\* Pointer to the thread descriptor. \*/  
 int multiple\_threads;  
 int gscope\_flag;  
 uintptr\_t sysinfo;  
 uintptr\_t stack\_guard; <- canary值，偏移位置0x28处  
 uintptr\_t pointer\_guard;  
 ......  
} tcbhead\_t;

结构体tcbhead*t就是用来描述tls的也就是%fs寄存器指向的位置，其中+0x28偏移位置的成员变量stack*guard就是canary值。另外通过**strace ./test**看到在进程加载的过程中会调用arch\_prctl系统调用来设置%fs的值，

root@localhost stackp]# strace ./test  
execve("./test", ["./test"], [/\* 24 vars \*/]) = 0  
......  
arch\_prctl(ARCH\_SET\_FS, 0x7f985a041740) = 0  
......

产生canary值的代码在glibc的*dl*main和\_*libc*start\_main函数中：

/\* Set up the stack checker's canary. \*/  
 uintptr\_t stack\_chk\_guard = \_dl\_setup\_stack\_chk\_guard (\_dl\_random);  
# ifdef THREAD\_SET\_STACK\_GUARD  
 THREAD\_SET\_STACK\_GUARD (stack\_chk\_guard);  
# else  
 \_\_stack\_chk\_guard = stack\_chk\_guard;  
# endif

*dl*random是一个随机数，它由*dl*sysdep*start函数从内核获取的。*dl*setup*stack*chk*guard函数负责生成canary值，THREAD*SET*STACK\_GUARD宏将canary设置到%fs:0x28位置。

**在应用程序栈保护中，进程的%fs寄存器是由glibc来管理的，并不涉及到内核提供的功能。**

**3. x32应用程序栈保护**

解读完了x64的实现，我们来看看x32下面的情况，我们还是使用上面例子的代码在x32的机器上编译，得到下面的代码：

08048464 <main>:  
 ......  
 8048474: 65 a1 14 00 00 00 mov %gs:0x14,%eax # 插入canary值  
 804847a: 89 44 24 3c mov %eax,0x3c(%esp)  
 ......  
 80484aa: 65 33 15 14 00 00 00 xor %gs:0x14,%edx # 检查canary值  
 80484b1: 74 05 je 80484b8 <main+0x54> # 0x80484b8  
 80484b3: e8 c0 fe ff ff call 8048378 <\_\_stack\_chk\_fail@plt>  
 80484b8: c9 leave  
 80484b9: c3 ret

在x32下的实现和x64是一样的，只不过canary值保存在%gs:0x14中，glibc使用%gs寄存器来保存TLS信息。

**0x02 内核态栈保护**

Linux的CC*STACKPROTECTOR补丁提供了对内核栈溢出保护功能，该补丁是Tejun Heo在09年给主线kernel提交的。  
– 2.6.24：首次出现CONFIG*CC\_STACKPROTECTOR编译选项并实现了x64平台的进程上下文栈保护支持；  
– 2.6.30：新增对内核中断上下文的栈保护和对x32平台进程上下文的栈保护支持;  
– 3.14：对该功能进行了一次升级以支持gcc的-fstack-protector-strong参数，提供更大范围的栈保护。

**1. 栈保护工作原理**

我们参照前面的代码写了一个可加载模块并反汇编来看看是怎么样的：

[root@localhost stackpk]# objdump -r test.ko  
RELOCATION RECORDS FOR [.text]:  
OFFSET TYPE VALUE  
0000000000000061 R\_X86\_64\_PC32 \_\_stack\_chk\_fail-0x0000000000000004  
......  
  
[root@localhost stackpk]# objdump -d test.ko  
0000000000000000 <foo>:  
 ......  
 # 函数开头，往栈里插入canary  
 24: 65 48 8b 04 25 28 00 mov %gs:0x28,%rax  
 2d: 48 89 45 f8 mov %rax,-0x8(%rbp)  
 ......  
 # 函数返回，检查canary  
 4a: 48 8b 45 f8 mov -0x8(%rbp),%rax  
 4e: 65 48 33 04 25 28 00 xor %gs:0x28,%rax  
 57: 75 02 jne 5b <foo+0x5b>  
 59: c9 leaveq  
 5a: c3 retq  
 5b: 0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)  
 60: e8 00 00 00 00 callq 65 <foo+0x65> # \_\_stack\_chk\_fail

内核函数的栈保护工作原理和应用程序的栈保护是一样的，只不过canary是从内核%gs:0x28位置取的，并且检查失败时调用内核的\_*stack*chk\_fail函数并且产生panic。

**2. 中断上下文canary值的产生**

**x64平台**

硬件中断(hardware interrupt)和软中断(softirq)都使用中断栈(Interrupt stack)。

当内核刚进入64位模式的时候，startup*64函数为内核的初始化工作设置好了%gs寄存器和分配栈空间，代码在arch/x86/kernel/head*64.S中，下面是startup\_64函数片段：

/\* Setup a boot time stack \*/  
movq stack\_start(%rip), %rsp  
  
/\* Set up %gs \*/  
movl $MSR\_GS\_BASE,%ecx  
movl initial\_gs(%rip),%eax  
movl initial\_gs+4(%rip),%edx  
wrmsr   
  
GLOBAL(initial\_gs)  
.quad INIT\_PER\_CPU\_VAR(irq\_stack\_union)  
GLOBAL(stack\_start)  
.quad init\_thread\_union+THREAD\_SIZE-8  
.word 0

其中%gs被定义为percpu变量，可用irq*stack*union联合体表示：

union irq\_stack\_union {  
 char irq\_stack[IRQ\_STACK\_SIZE];  
 struct {  
 char gs\_base[40];  
 unsigned long stack\_canary; <- GS+40偏移位置  
 };  
};

startup*64的末尾会跳转到start*kernel函数，该函数也是canary产生的地方。start*kernel调用了boot*init*stack*canary，它的作用就是产生一个随机的canary并且应用到当前%gs:0x28位置：

static \_\_always\_inline void boot\_init\_stack\_canary(void)  
{  
 u64 canary;  
 u64 tsc;  
  
#ifdef CONFIG\_X86\_64  
 BUILD\_BUG\_ON(offsetof(union irq\_stack\_union, stack\_canary) != 40);  
#endif  
  
 /\* 同时使用随机数和TSC，让随机性更强 \*/  
 get\_random\_bytes(&canary, sizeof(canary));  
 tsc = rdtsc();  
 canary += tsc + (tsc << 32UL);  
  
 /\* PID为0, 此时的进程名叫swapper,之后成为idle进程 \*/  
 current->stack\_canary = canary; <- canary信息保存在这里  
#ifdef CONFIG\_X86\_64  
 this\_cpu\_write(irq\_stack\_union.stack\_canary, canary);  
#else  
 this\_cpu\_write(stack\_canary.canary, canary);  
#endif  
}

start*kernel函数是由boot CPU执行的，在多核心的情况下还会在每个CPU核心初始化的时候分别调用boot*init*stack*canary来产生canary值。我们发现中断上下文的canary值保存在每个核心的idle进程task*struct->stack*canary中，这样当上下文切换的时候就不会丢失了。

**x32平台**

在x32平台上的percpu区域是保存在%fs中的，内核初始化的时候会为每个cpu核心产生canary值存放在percpu区域的stack\_canary成员中备用。

struct stack\_canary {  
 char \_\_pad[20]; /\* canary at %gs:20 \*/  
 unsigned long canary;  
};  
DECLARE\_PER\_CPU\_ALIGNED(struct stack\_canary, stack\_canary);

但是gcc要求canary值必须从%gs:0x14偏移的位置获取，因此必须让%gs:0x14指向percpu变量stack\_canary.canary才行。首先内核选用GDT第28项描述的数据段来存放canary信息：

/\* ./arch/x86/include/asm/segment.h \*/  
#define GDT\_ENTRY\_STACK\_CANARY 28 <- canary数据段索引  
  
#ifdef CONFIG\_CC\_STACKPROTECTOR  
# define \_\_KERNEL\_STACK\_CANARY (GDT\_ENTRY\_STACK\_CANARY\*8)  
#else  
# define \_\_KERNEL\_STACK\_CANARY 0  
#endif  
  
/\* ./arch/x86/include/asm/stackprotector.h \*/  
#define GDT\_ENTRY\_INIT(flags, base, limit) { { { \  
 .a = ((limit) & 0xffff) | (((base) & 0xffff) << 16), \  
 .b = (((base) & 0xff0000) >> 16) | (((flags) & 0xf0ff) << 8) | \  
 ((limit) & 0xf0000) | ((base) & 0xff000000), \  
 } } }  
  
/\* 段描述符 limit=0x18,s=1,dpl=0,p=1,avl=0,l=0,d=1,g=0 \*/  
#define GDT\_STACK\_CANARY\_INIT \  
 [GDT\_ENTRY\_STACK\_CANARY] = GDT\_ENTRY\_INIT(0x4090, 0, 0x18),  
  
/\* 将percpu变量stack\_canary的地址设置到段描述符base \*/  
static inline void setup\_stack\_canary\_segment(int cpu)  
{  
#ifdef CONFIG\_X86\_32  
 unsigned long canary = (unsigned long)&per\_cpu(stack\_canary, cpu);  
 struct desc\_struct \*gdt\_table = get\_cpu\_gdt\_table(cpu);  
 struct desc\_struct desc;  
  
 desc = gdt\_table[GDT\_ENTRY\_STACK\_CANARY];  
 set\_desc\_base(&desc, canary);  
 write\_gdt\_entry(gdt\_table, GDT\_ENTRY\_STACK\_CANARY, &desc, DESCTYPE\_S);  
#endif  
}

接下来只要设置%gs寄存器来索引该段描述符就可以了：

/\* ./arch/x86/include/asm/stackprotector.h \*/  
static inline void load\_stack\_canary\_segment(void)  
{  
#ifdef CONFIG\_X86\_32  
 asm("mov %0, %%gs" : : "r" (\_\_KERNEL\_STACK\_CANARY) : "memory");  
#endif  
}  
  
/\* ./arch/x86/kernel/cpu/common.c \*/  
void load\_percpu\_segment(int cpu)  
{  
#ifdef CONFIG\_X86\_32  
 loadsegment(fs, \_\_KERNEL\_PERCPU);  
#else  
 loadsegment(gs, 0);  
 wrmsrl(MSR\_GS\_BASE, (unsigned long)per\_cpu(irq\_stack\_union.gs\_base, cpu));  
#endif  
 load\_stack\_canary\_segment();  
}

到这里x32平台的%gs:0x14偏移的canary值已经初始化完成了。

**2. 进程上下文canary值的产生**

无论是内核线程还是用户线程，当一个线程创建的时候，内核给线程生成一个canary值存放在task*struct结构体的stack*canary成员变量中，见dup*task*struct函数：

/\* ./kernel/fork.c \*/  
  
 static struct task\_struct \*dup\_task\_struct(struct task\_struct \*orig)  
 {  
 ......  
 #ifdef CONFIG\_CC\_STACKPROTECTOR  
 tsk->stack\_canary = get\_random\_int();  
 #endif  
 ......  
 }

接下来内核要做的事情就是当发生线程切换的时候让该canary值设置到%gs:0x28偏移处，这个是在switch\_to宏中完成的：

/\* ./arch/x86/include/asm/switch\_to.h \*/  
  
#define \_\_switch\_canary \  
 "movq %P[task\_canary](%%rsi),%%r8\n\t" \  
 "movq %%r8,"\_\_percpu\_arg([gs\_canary])"\n\t"  
#define \_\_switch\_canary\_oparam \  
 , [gs\_canary] "=m" (irq\_stack\_union.stack\_canary)  
#define \_\_switch\_canary\_iparam \  
 , [task\_canary] "i" (offsetof(struct task\_struct, stack\_canary))  
  
#define switch\_to(prev, next, last) \  
 asm volatile(SAVE\_CONTEXT \  
 ......  
 \_\_switch\_canary \  
 ......  
 : "=a" (last) \  
 \_\_switch\_canary\_oparam \  
 : [next] "S" (next), [prev] "D" (prev), \  
 ......  
 \_\_switch\_canary\_iparam \  
 : "memory", "cc" \_\_EXTRA\_CLOBBER)

上面是x64平台上的代码并且忽略了与canary值切换无关的部分，它把task*struct->stack*canary赋值到percpu变量irq*stack*union.stack*canary，这样我们代码中找到的canary值就是当前上下文的了。在x32平台上也是类似的只不过percpu变量是stack*canary.canary。

总结：  
在gcc、glibc和内核的共同支持下，Linux对所有的可能发生缓冲区溢出的栈返回地址都进行了保护：  
1. 在应用进程上下文，canary值由glibc产生并保存在tcbhead*t中，当canary检查失败时执行glibc的stack*chk*fail，并终止进程；  
2. 在内核进程上下文，canary值在内核copy*process时产生并保存在task*struct中，当canary检查失败时执行内核的stack*chk*fail，并产生panic；  
3. 在内核中断上下文，canary值在start*kernel以及每个CPU核心初始化的时候产生并保存在每个CPU核心的idle进程task*struct中，当canary检查失败时执行内核的stack*chk\_fail，并产生panic。

**0x03 参考资料**

* http://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git/commit/?id=60a5317ff0f42dd313094b88f809f63041568b08
* https://lwn.net/Articles/584278/
* https://lwn.net/Articles/318565/
* http://blog.aliyun.com/1126
* https://outflux.net/blog/archives/2014/01/27/fstack-protector-strong/?utm*source=tuicool&utm*medium=referral
* http://www.ibm.com/developerworks/cn/linux/l-overflow/
* https://github.com/wishstudio/flinux/wiki/Difference-between-Linux-and-Windows
* https://xorl.wordpress.com/2010/10/14/linux-glibc-stack-canary-values/