CC_STACKPROTECTOR 防止内核 stack 溢出补丁分析

by wzt <wzt.wzt@gmail.com>

CC_STACKPROTECT 补丁是 Tejun Heo 在 09 年给主线 kernel 提交的一个用来防止内核堆栈溢出的补丁。默认的 config 是将这个选项关闭的，可以在编译内核的时候，
修改.config 文件为 CONFIG_CC_STACKPROTECTOR=y 来启用。未来飞天内核可以将这个选项开启来防止利用内核 stack 溢出的 0day 攻击。
这个补丁的防溢出原理是： 在进程启动的时候， 在每个 buffer 的后面放置一个预先设置好的 stack canary，你可以
把它理解成一个哨兵， 当 buffer 发生缓冲区溢出的时候， 肯定会破坏 stack canary 的值， 当 stack canary 的值被破坏的时候， 内核就会直接当机。那么是怎么判断 stack canary
被覆盖了呢？ 其实这个事情是 gcc 来做的，内核在编译的时候给 gcc 加了个-fstack-protector 参数， 我们先来研究下这个参数是做什么用的。

先写个简单的有溢出的程序：

```
[wzt@localhost csaw]$ cat test.c
#include <stdio.h>
#include <stdlib.h>

void test(void)
{
        char buff[64];

        memset(buff, 0x41, 128);        //向 64 大小的 buffer 拷贝 128 字节， 肯定会发生缓冲区溢出。
}

int main(void)
{
        test();

        return 0;
}
[wzt@localhost csaw]$ gcc -o test test.c
[wzt@localhost csaw]$ ./test
段错误
```

反汇编看看：

```
[wzt@localhost csaw]$ objdump -d test > hex

08048384 <test>:
 8048384:       55                      push   %ebp
 8048385:       89 e5                   mov    %esp,%ebp
 8048387:       83 ec 58                sub    $0x58,%esp
 804838a:       c7 44 24 08 80 00 00    movl   $0x80,0x8(%esp)
 8048391:       00
 8048392:       c7 44 24 04 41 00 00    movl   $0x41,0x4(%esp)
 8048399:       00
```

```
 804839a:       8d 45 c0                lea    0xffffffc0(%ebp),%eax
 804839d:       89 04 24                mov    %eax,(%esp)
 80483a0:       e8 e3 fe ff ff          call   8048288 <memset@plt>
 80483a5:       c9                      leave
 80483a6:       c3                      ret
```

没什么特别的，我们在加上-fstack-protector 参数看看：

```
[wzt@localhost csaw]$ gcc -o test test.c -fstack-protector
[wzt@localhost csaw]$ ./test
*** stack smashing detected ***: ./test terminated
已放弃
```

这次程序打印了一条堆栈被溢出的信息，然后就自动退出了。

在反汇编看下：

```
[wzt@localhost csaw]$ objdump -d test > hex1
```

```
080483d4 <test>:
 80483d4:       55                      push   %ebp
 80483d5:       89 e5                   mov    %esp,%ebp
 80483d7:       83 ec 68                sub    $0x68,%esp
 80483da:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 80483e0:       89 45 fc                mov    %eax,0xfffffffc(%ebp)
 80483e3:       31 c0                   xor    %eax,%eax
 80483e5:       c7 44 24 08 80 00 00    movl   $0x80,0x8(%esp)
 80483ec:       00
 80483ed:       c7 44 24 04 41 00 00    movl   $0x41,0x4(%esp)
 80483f4:       00
 80483f5:       8d 45 bc                lea    0xffffffbc(%ebp),%eax
 80483f8:       89 04 24                mov    %eax,(%esp)
 80483fb:       e8 cc fe ff ff          call   80482cc <memset@plt>
 8048400:       8b 45 fc                mov    0xfffffffc(%ebp),%eax
 8048403:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 804840a:       74 05                   je     8048411 <test+0x3d>
 804840c:       e8 db fe ff ff          call   80482ec <__stack_chk_fail@plt>
 8048411:       c9                      leave
 8048412:       c3                      ret
```

使用-fstack-protector 参数后， gcc 在函数的开头放置了几条汇编代码：

```
 80483d7:       83 ec 68                sub    $0x68,%esp
 80483da:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 80483e0:       89 45 fc                mov    %eax,0xfffffffc(%ebp)
```

将代码段 gs 偏移 0x14 内存处的值赋值给了 ebp-4， 也就是第一个变量值的后面。

在 call 完 memeset 后，有如下汇编代码：

```
 80483fb:       e8 cc fe ff ff          call   80482cc <memset@plt>
 8048400:       8b 45 fc                mov    0xfffffffc(%ebp),%eax
 8048403:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 804840a:       74 05                   je     8048411 <test+0x3d>
 804840c:       e8 db fe ff ff          call   80482ec <__stack_chk_fail@plt>
```

在 memset 后，gcc 要检查这个操作是否发生了堆栈溢出，将保存在 ebp-4 的这个值与原来的值

对比一下，
如果不相同， 说明堆栈发生了溢出，那么就会执行__stack_chk_fail 这个函数， 这个函数是 glibc 实现的，
打印出上面看到的信息， 然后进程退出。

从这个例子中我们可以看出 gcc 使用了-fstack-protector 参数后，会自动检查堆栈是否发生了溢出， 但是有一个前提就是
内核要给每个进程提前设置好一个检测值放置在%gs:0x14 位置处， 这个值称之为 stack canary。所以我们可以看到防止
堆栈溢出是由内核和 gcc 共同来完成的。

gcc 的任务就是放置几条汇编代码， 然后和%gs:0x14 位置处的值进行对比即可。 主要任务还是内核如何来设置 stack canary， 也是
CC_STACKPROTECTOR 补丁要实现的目的， 下面我们仔细来看下这个补丁是如何实现的。

既然 gcc 硬性规定了 stack canary 必须在%gs 的某个偏移位置处，那么内核也必须按着这个规定来设置。

对于 32 位和 64 位内核， gs 寄存器有着不同的功能。

64 位内核 gcc 要求 stack canary 是放置在 gs 段的 40 偏移处， 并且 gs 寄存器在每 cpu 变量中是共享的，每 cpu 变量 irq_stack_union 的结构如下：

arch/x86/include/asm/processor.h

```
union irq_stack_union {
        char irq_stack[IRQ_STACK_SIZE];
        /*
         * GCC hardcodes the stack canary as %gs:40.  Since the
         * irq_stack is the object at %gs:0, we reserve the bottom
         * 48 bytes of the irq stack for the canary.
         */
        struct {
                char gs_base[40];
                unsigned long stack_canary;
        };
};

DECLARE_PER_CPU_FIRST(union irq_stack_union, irq_stack_union);
```

gs_base 只是一个 40 字节的站位空间， stack_canary 就紧挨其后。
并且在应用程序进出内核的时候，内核会使用 swapgs 指令自动更换 gs 寄存器的内容。

32 位下就稍微有点复杂了。由于某些处理器在加载不同的段寄存器时很慢， 所以内核使用 fs 段寄存器替换了
gs 寄存器。 但是 gcc 在使用-fstack-protector 的时候， 还要用到 gs 段寄存器， 所以内核还要管理 gs 寄存器，
我们要把 CONFIG_X86_32_LAZY_GS 选项关闭， gs 也只在进程切换的时候才改变。 32 位用每 cpu 变量 stack_canary 保存 stack canary。

```
struct stack_canary {
        char __pad[20];         /* canary at %gs:20 */
        unsigned long canary;
};
DECLARE_PER_CPU_ALIGNED(struct stack_canary, stack_canary);
```

内核是处于保护模式的， 因此 gs 寄存器就变成了保护模式下的段选子，在 GDT 表中也要有相应的设置：

```
diff --git a/arch/x86/include/asm/segment.h b/arch/x86/include/asm/segment.h
index 1dc1b51..14e0ed8 100644 (file)
--- a/arch/x86/include/asm/segment.h
+++ b/arch/x86/include/asm/segment.h
@@ -61,7 +61,7 @@
  *
  *  26 - ESPFIX small SS
  *  27 - per-cpu                        [ offset to per-cpu data area ]
- *  28 - unused
+ *  28 - stack_canary-20                [ for stack protector ]
  *  29 - unused
  *  30 - unused
  *  31 - TSS for double fault handler
@@ -95,6 +95,13 @@
 #define __KERNEL_PERCPU 0
 #endif

+#define GDT_ENTRY_STACK_CANARY          (GDT_ENTRY_KERNEL_BASE + 16)
+#ifdef CONFIG_CC_STACKPROTECTOR
+#define __KERNEL_STACK_CANARY           (GDT_ENTRY_STACK_CANARY * 8)
+#else
+#define __KERNEL_STACK_CANARY           0
+#endif
+
 #define GDT_ENTRY_DOUBLEFAULT_TSS       31
```

GDT 表中的第 28 个表项用来定为 stack canary 所在的段。

```
#define GDT_STACK_CANARY_INIT                                                   \
        [GDT_ENTRY_STACK_CANARY] = GDT_ENTRY_INIT(0x4090, 0, 0x18),
```

GDT_STACK_CANARY_INIT 在刚进入保护模式的时候被调用， 这个段描述符项被设置为基地址为 0， 段大小设为 24，因为只在基地址为 0， 偏移为 0x14 处放置一个 4bytes 的 stack canary， 所以 24 字节正好。
不理解的同学可以看看 intel 保护模式的手册， 对着段描述符结构一个个看就行了。

在进入保护模式后， start_kernel()会调用 boot_init_stack_canary()来初始话一个 stack canary。

```
/*
 * Initialize the stackprotector canary value.
```

```
 *
 * NOTE: this must only be called from functions that never return,
 * and it must always be inlined.
 */
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
        /*
         * We both use the random pool and the current TSC as a source
         * of randomness. The TSC only matters for very early init,
         * there it already has some randomness on most systems. Later
         * on during the bootup the random pool has true entropy too.
         */
        get_random_bytes(&canary, sizeof(canary));
        tsc = __native_read_tsc();
        canary += tsc + (tsc << 32UL);

        current->stack_canary = canary;
#ifdef CONFIG_X86_64
        percpu_write(irq_stack_union.stack_canary, canary);
#else
        percpu_write(stack_canary.canary, canary);
#endif
}
```

随机出了一个值赋值给每 cpu 变量， 32 位是 stack_canary, 64 位是 irq_stack_union。

内核在进一步初始化 cpu 的时候，会调用 setup_stack_canary_segment()来设置每个 cpu 的 GDT 的 stack canary 描述符项：

start_kernel()->setup_per_cpu_areas()->setup_stack_canary_segment：

```
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
}
```

在内核刚进入保护模式的时候, stack canary 描述符的基地址被初始化为 0， 现在在 cpu 初始化

的时候要重新设置为每 cpu 变量 stack_canary 的地址， 而不是变量保存的值。通过这些设置当内核代码在访问 %gs:0x14 的时候， 就会访问 stack canry 保存的值。注意：setup_stack_canary_segment 是针对 32 位内核做设置， 因为 64 位内核中的 irq_stack_union 是每个 cpu 共享的， 不用针对每个 cpu 单独设置。 然后就可以调用 switch_to_new_gdt(cpu);来加载 GDT 表和加载 gs 寄存器。

经过上述初始化过程，在内核代码里访问%gs:0x14 就可以定位 stack canary 的值了， 那么每个进程的 stack canary 是什么时候设置的呢？

在内核启动一个进程的时候， 会把 gs 寄存器的值设为__KERNEL_STACK_CANARY

```
--- a/arch/x86/kernel/process_32.c
+++ b/arch/x86/kernel/process_32.c
@@ -212,6 +212,7 @@ int kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
             regs.ds = __USER_DS;
             regs.es = __USER_DS;
             regs.fs = __KERNEL_PERCPU;
+            regs.gs = __KERNEL_STACK_CANARY;
             regs.orig_ax = -1;
             regs.ip = (unsigned long) kernel_thread_helper;
             regs.cs = __KERNEL_CS | get_kernel_rpl();
```

内核在 fork 一个进程的时候， 有如下操作：

```
static struct task_struct *dup_task_struct(struct task_struct *orig)
{
#ifdef CONFIG_CC_STACKPROTECTOR
            tsk->stack_canary = get_random_int();
#endif
}
```

随机初始化了一个 stack_canary 保存在 task_struct 结构中的 stack_canary 变量中。当进程在切换的时候， 通过 switch 宏把新进程的 stack canary 保存在每 cpu 变量 stack_canary 中， 当前进程的 stack_canary 也保存在一个每 cpu 变量中，完成 stack canary 的切换。

```
diff --git a/arch/x86/include/asm/system.h b/arch/x86/include/asm/system.h
index 79b98e5..2692ee8 100644 (file)
--- a/arch/x86/include/asm/system.h
+++ b/arch/x86/include/asm/system.h
@@ -23,6 +23,22 @@ struct task_struct *__switch_to(struct task_struct *prev,

  #ifdef CONFIG_X86_32

+#ifdef CONFIG_CC_STACKPROTECTOR
+#define __switch_canary                                                  \
+            "movl "__percpu_arg([current_task])",%%ebx\n\t"              \
+            "movl %P[task_canary](%%ebx),%%ebx\n\t"                      \
+            "movl %%ebx,"__percpu_arg([stack_canary])"\n\t"
+#define __switch_canary_oparam                                           \
```

```
+            , [stack_canary] "=m" (per_cpu_var(stack_canary))
+#define __switch_canary_iparam                                          \
+            , [current_task] "m" (per_cpu_var(current_task))                    \
+            , [task_canary] "i" (offsetof(struct task_struct, stack_canary))
+#else    /* CC_STACKPROTECTOR */
+#define __switch_canary
+#define __switch_canary_oparam
+#define __switch_canary_iparam
+#endif /* CC_STACKPROTECTOR */
+
 /*
   * Saving eflags is important. It switches not only IOPL between tasks,
   * it also protects other tasks from NT leaking through sysenter etc.
@@ -46,6 +62,7 @@ do
{                                                                       \
                "pushl %[next_ip]\n\t"          /* restore EIP      */      \
                "jmp __switch_to\n"             /* regparm call     */      \
                "1:\t"                                                      \
+               __switch_canary                                             \
                "popl %%ebp\n\t"                /* restore EBP      */      \
                "popfl\n"                       /* restore flags */         \
                                                                            \
@@ -58,6 +75,8 @@ do
{                                                                       \
                    "=b" (ebx), "=c" (ecx), "=d" (edx),                     \
                    "=S" (esi), "=D" (edi)                                  \
                                                                            \
+                   __switch_canary_oparam                                  \
+                                                                           \
                    /* input parameters: */                                 \
                  : [next_sp]  "m" (next->thread.sp),                       \
                    [next_ip]  "m" (next->thread.ip),                       \
@@ -66,6 +85,8 @@ do
{                                                                       \
                    [prev]       "a" (prev),                                \
                    [next]       "d" (next)                                 \
                                                                            \
+                   __switch_canary_iparam                                  \
+                                                                           \
                  : /* reloaded segment registers */                        \
                      "memory");                                            \
 } while (0)
```

前面讲过当 gcc 检测到堆栈溢出的时候， 会调用 glibc 的__stack_chk_fail 函数， 但是当内核堆栈发生溢出的时候，
不能调用 glibc 的函数，所以内核自己实现了一个__stack_chk_fail 函数：

kernel/panic.c

```
#ifdef CONFIG_CC_STACKPROTECTOR

/*
 * Called when gcc's -fstack-protector feature is used, and
 * gcc detects corruption of the on-stack canary value
 */
void __stack_chk_fail(void)
{
        panic("stack-protector: Kernel stack is corrupted in: %p\n",
                __builtin_return_address(0));
}
EXPORT_SYMBOL(__stack_chk_fail);

#endif
```

当内核堆栈发生溢出的时候，就会执行__stack_chk_fail 函数， 内核当机。 这就是这个补丁的原理，不懂的同学请参考：
http://git.kernel.org/?p=linux/kernel/git/next/linux-next.git;a=commitdiff;h=60a5317ff0f42dd313094b88f809f63041568b08