```
    int intlen(long x)
    x in %rdi
1   intlen:
2     subq    $40, %rsp
3     movq    %rdi, 24(%rsp)
4     leaq    24(%rsp), %rsi
5     movq    %rsp, %rdi
6     call    iptoa
```

a）不带保护者

```
    int intlen(long x)
    x in %rdi
1   intlen:
2     subq    $56, %rsp
3     movq    %fs:40, %rax
4     movq    %rax, 40(%rsp)
5     xorl    %eax, %eax
6     movq    %rdi, 8(%rsp)
7     leaq    8(%rsp), %rsi
8     leaq    16(%rsp), %rdi
9     call    iptoa
```

b）带保护者

A. 对于两个版本：buf、v 和金丝雀值(如果有的话)分别在栈帧中的什么位置？

B. 在有保护的代码中，对局部变量重新排列如何提供更好的安全性来对抗缓冲区越界攻击？

3. 限制可执行代码区域

最后一招是消除攻击者向系统中插入可执行代码的能力。一种方法是限制哪些内存区域能够存放可执行代码。在典型的程序中，只有保存编译器产生的代码的那部分内存才需要是可执行的。其他部分可以被限制为只允许读和写。正如第 9 章中会看到的，虚拟内存空间在逻辑上被分成了页(page)，典型的每页是 2048 或者 4096 个字节。硬件支持多种形式的内存保护，能够指明用户程序和操作系统内核所允许的访问形式。许多系统允许控制三种访问形式：读(从内存读数据)、写(存储数据到内存)和执行(将内存的内容看作机器级代码)。以前，x86 体系结构将读和执行访问控制合并成一个 1 位的标志，这样任何被标记为可读的页也都是可执行的。栈必须是既可读又可写的，因而栈上的字节也都是可执行的。已经实现的很多机制，能够限制一些页是可读但是不可执行的，然而这些机制通常会带来严重的性能损失。

最近，AMD 为它的 64 位处理器的内存保护引入了“NX”(No-Execute，不执行)位，将读和执行访问模式分开，Intel 也跟进了。有了这个特性，栈可以被标记为可读和可写，但是不可执行，而检查页是否可执行由硬件来完成，效率上没有损失。

有些类型的程序要求动态产生和执行代码的能力。例如，“即时(just-in-time)”编译技术为解释语言(例如 Java)编写的程序动态地产生代码，以提高执行性能。是否能够将可执行代码限制在由编译器在创建原始程序时产生的那个部分中，取决于语言和操作系统。

我们讲到的这些技术——随机化、栈保护和限制哪部分内存可以存储可执行代码——是用于最小化程序缓冲区溢出攻击漏洞三种最常见的机制。它们都具有这样的属性，即不需要程序员做任何特殊的努力，带来的性能代价都非常小，甚至没有。单独每一种机制都降低了漏洞的等级，而组合起来，它们变得更加有效。不幸的是，仍然有方法能够攻击计算机[85，97]，因而蠕虫和病毒继续危害着许多机器的完整性。

3.10.5　支持变长栈帧

到目前为止，我们已经检查了各种函数的机器级代码，但它们有一个共同点，即编译器能够预先确定需要为栈帧分配多少空间。但是有些函数，需要的局部存储是变长的。例如，当函数调用 alloca 时就会发生这种情况。alloca 是一个标准库函数，可以在栈上分配任意字节数量的存储。当代码声明一个局部变长数组时，也会发生这种情况。

虽然本节介绍的内容实际上是如何实现过程的一部分，但我们还是把它推迟到现在才