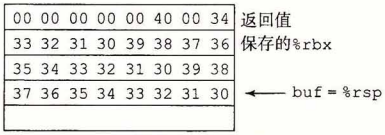
B. 执行了第 5 行后的栈：



C. 这个程序试图返回到地址 0x040034。低位 2 字节被字符'4'和结尾的空(null)字符覆盖了。

D. 寄存器%rbx 的保存值被设置为 0x3332313039383736。在 get_line 返回前，这个值会被加载回这个寄存器中。

E. 对 malloc 的调用应该以 strlen(buf)+ 1 作为它的参数，而且代码还应该检查返回值是否为 NULL。

3.47　A. 这对应于大约 2^{13} 个地址的范围。

B. 每次尝试，一个 128 字节的空操作 sled 会覆盖 2^7 个地址，因此我们只需要 $2^6=64$ 次尝试。

这个例子明确地表明了这个版本的 Linux 中的随机化程度只能很小地阻挡溢出攻击。

3.48　这道题让你看看 x86-64 代码如何管理栈，也让你更好地理解如何防卫缓冲区溢出攻击。

A. 对于没有保护的代码，第 4 行和第 5 行计算 v 和 buf 的地址为相对于%rsp 偏移量为 24 和 0。在有保护的代码中，金丝雀被存放在偏移量为 40 的地方(第 4 行)，而 v 和 buf 在偏移量为 8 和 16 的地方(第 7 行和第 8 行)。

B. 在有保护的代码中，局部变量 v 比 buf 更靠近栈顶，因此 buf 溢出就不会破坏 v 的值。

3.49　这段代码中包含许多我们已经见到过的执行位级运算的技巧。要仔细研究才能看得懂。

A. 第 5 行的 leaq 指令计算值 $8n+22$，然后第 6 行的 andq 指令把它向下舍入到最接近的 16 的倍数。当 n 是奇数时，结果值会是 $8n+8$，当 n 是偶数时，结果值会是 $8n+16$，这个值减去 s_1 就得到 s_2。

B. 该序列中的三条指令将 s_2 舍入到最近的 8 的倍数。它们利用了 2.3.7 节中实现除以 2 的幂用到的偏移和移位的组合。

C. 这两个例子可以看做最小化和最大化 e_1 和 e_2 的情况。

n	s_1	s_2	p	e_1	e_2
5	2065	2017	2024	1	7
6	2064	2000	2000	16	0

D. 可以看到 s_2 的计算方式会保留 s_1 的偏移量为最接近的 16 的倍数。还可以看到 p 会以 8 的倍数对齐，正是对 8 字节元素数组建议使用的。

3.50　这道题要求你仔细检查代码，小心留意使用的转换和数据传送指令。可以看到取出的值和转换的情况如下：

- 取出位于 dp 的值，转换成 int(第 4 行)，再存储到 ip。因此可以推断出 val1 是 d。
- 取出位于 ip 的值，转换成 float(第 6 行)，再存储到 fp。因此可以推断出 val2 是 i。
- 1 的值被转换成 double(第 8 行)，并存储在 dp。因此可以推断出 val3 是 l。
- 第 3 行上取出位于 fp 的值。第 10 和 11 行的两条指令把它转换为双精度，值通过寄存器%xmm0 返回。因此可以推断出 val4 是 f。

3.51　可以通过从图 3-47 和图 3-48 中选择适当的条目或者使用在浮点格式间转换的代码序列来处理这些情况。

T_x	T_y	指令
long	double	vcvtsi2sdq %rdi,%xmm0,%xmm0
double	int	vcvttsd2si %xmm0,%eax
double	float	vunpcklpd %xmm0,%xmm0,%xmm0 vcvtpd2ps %xmm0,%xmm0
long	float	vctsi2ssq %rdi,%xmm0,%xmm0
float	long	vcvttss2siq %xmm0,%rax