Internationally collaborative higher education

AttackLab解析

Tel:+86-021-62233586 Fax:+86-021-62606775
E-mail: ecnu@ecnu.com.cn Http: //www.ecnu.edu.cn

背景

缓冲区溢出攻击是一种历史悠久的攻击方式。缓冲区溢出是指程序向固定大小的缓冲区写入数据时，超出其边界并覆盖相邻内存空间（如栈上的返回地址或函数指针），这样我们可以控制程序跳转到本来不应该跳转到的位置（ctarget touch1)。

但是现代的程序提供了许多保护机制，这些保护机制可以在很大程度上避免程序遭受缓冲区溢出攻击。

后来，随着保护机制的加强，一种更加难以防御的，被称为ROP的攻击方式开始兴起，它通过利用程序内存中已存在的gadgets，“拼凑”出一段恶意程序，从而对计算机系统产生威胁。

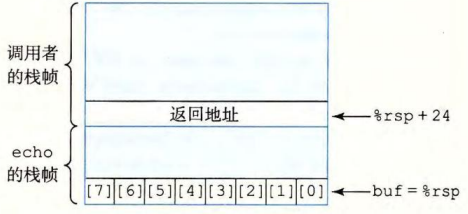


图 3-40 echo 函数的栈组织。字符数组 buf 就在保存的状态下面。对 buf 的越界写会破坏程序的状态

ctarget-缓冲区溢出攻击

ctarget包含三个touch（我们可以认为是三个目标），目标的程序结构已经给出，它们均调用了getbuf函数，这个函数有着显而易见的缓冲区溢出缺陷（gets函数），我们需要通过向其中输入字符串，期望通过缓冲区溢出改写返回地址进行攻击。

```
void test()
{
    int val;
    val = getbuf();
    printf("No exploit.  Getbuf returned 0x%x\n", val);
}
```

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

touch1:warmup

第一个touch非常直观，它不需要任何额外的操作，只需要我们将控制流跳转到touch1函数中即可。

```
void touch1()
{
    vlevel = 1;        /* Part of validation protocol */
    printf("Touch1!: You called touch1()\n");
    validate(1);
    exit(0);
}
```

touch1:warmup

第一个touch非常直观，它不需要任何额外的操作，只需要我们将控制流跳转到touch1函数中即可。

gdb/objdump获得touch1的地址，然后我们就可以设计出一个字符串来执行这样的攻击了，注意大小端序。

```
void touch1()
{
    vlevel = 1;        /* Part of validation protocol */
    printf("Touch1!: You called touch1()\n");
    validate(1);
    exit(0);
}
```

```
(gdb) disas touch1
Dump of assembler code for function touch1:
   0x00000000004017c0 <+0>:     sub    $0x8,%rsp
   0x00000000004017c4 <+4>:     movl   $0x1,0x202d0e(%rip)
   0x00000000004017ce <+14>:    mov    $0x4030c5,%edi
   0x00000000004017d3 <+19>:    call   0x400cc0 <puts@plt>
   0x00000000004017d8 <+24>:    mov    $0x1,%edi
   0x00000000004017dd <+29>:    call   0x401c8d <validate>
   0x00000000004017e2 <+34>:    mov    $0x0,%edi
   0x00000000004017e7 <+39>:    call   0x400e40 <exit@plt>
```

touch2:写入数据

根据write-up中的内容，第二个touch要求让val值等于cookie.txt中给出的cookie值，否则视为失败。
对程序执行流的控制同样是通过getbuf中进行缓冲区溢出操作覆盖返回地址完成的。

```
void touch2(unsigned val)
{
    vlevel = 2; /* Part of validation protocol */
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
        }
    exit(0);
}
```

touch2:写入数据

根据write-up中的内容，第二个touch要求让val值等于cookie.txt中给出的cookie值，否则视为失败。
对程序执行流的控制同样是通过getbuf中进行缓冲区溢出操作覆盖返回地址完成的。
但是问题在于：我们如何给val赋上合适的值呢?

```
void touch2(unsigned val)
{
    vlevel = 2; /* Part of validation protocol */
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
        }
    exit(0);
}
```

touch2:写入数据

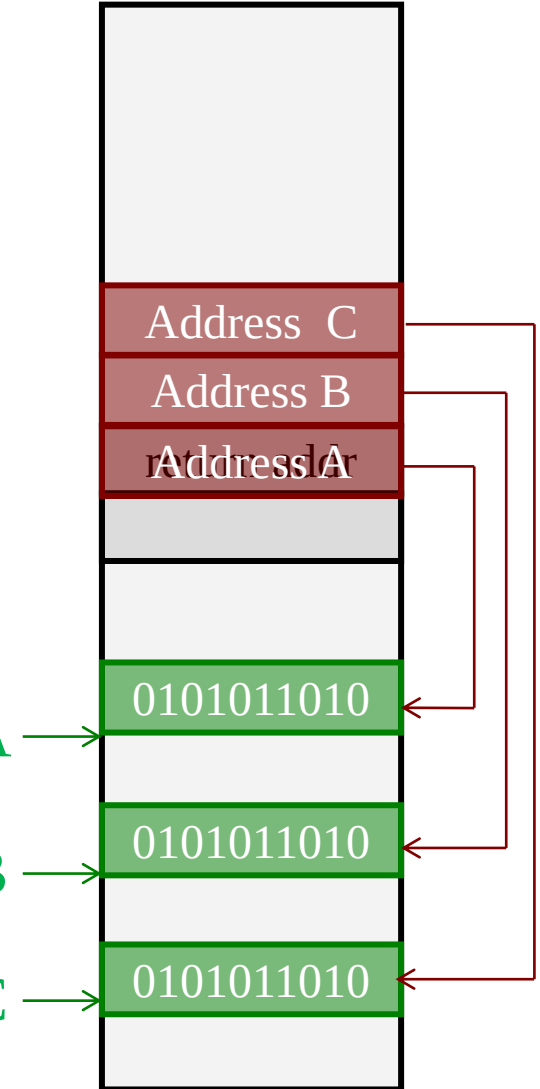
根据write-up中的内容，第二个touch要求让val值等于cookie.txt中给出的cookie值，否则视为失败。
对程序执行流的控制同样是通过getbuf中进行缓冲区溢出操作覆盖返回地址完成的。

```
void touch2(unsigned val)
{
    vlevel = 2; /* Part of validation protocol */
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
        }
    exit(0);
}
```

touch3:写入数据2.0

第三个touch实际上和第二个lab差不多，唯一的区别在于这次touch的参数换成了char *类型，也就是说，这次我们要传入一个字符串进行字符串形式的比较了。
如果第二个touch的思路弄明白了的话，这个touch应该是非常轻松的。

```
void touch3(char *sval)
{
    vlevel = 3; /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
    }
    exit(0);
}
```



rtarget-ROP攻击

在完成touch2和touch3时，我们假定栈中的数据可以作为代码执行。

但事实上，现代的操作系统对内存的权限限制十分严格，通过将一块内存标记为不可执行，操作系统可以很轻松地避免遭受前面我们使用的攻击方式的攻击(DEP)。

但是道高一尺魔高一丈，一种被称为ROP的攻击方式开始出现，它的原理如右图所示，我们在栈中存放了一系列带有ret代码段的地址（这些代码段称为gadget），通过“断章取义”，构建出一个恶意程序，由于我们使用的都是可执行区域内的代码，前面的防御手段被我们巧妙地绕开了。

A. Encodings of `movq` instructions

`movq` S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of `popq` instructions

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

C. Encodings of `movl` instructions

`movl` S, D

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

D. Encodings of 2-byte functional nop instructions

Operation	Register R			
	%al	%cl	%dl	%bl
andb R, R	20 c0	20 c9	20 d2	20 db
orb R, R	08 c0	08 c9	08 d2	08 db
cmpb R, R	38 c0	38 c9	38 d2	38 db
testb R, R	84 c0	84 c9	84 d2	84 db

Figure 3: Byte encodings of instructions. All values are shown in hexadecimal.

现在让我们参考左边的指令以及rtarget和farm.c的内容执行ROP攻击

```
0000000000401994 <start_farm>:
  401994:       b8 01 00 00 00          mov    $0x1,%eax
  401999:       c3                      ret

000000000040199a <getval_142>:
  40199a:       b8 fb 78 90 90          mov    $0x909078fb,%eax
  40199f:       c3                      ret

00000000004019a0 <addval_273>:
  4019a0:       8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
  4019a6:       c3                      ret

00000000004019a7 <addval_219>:
  4019a7:       8d 87 51 73 58 90       lea    -0x6fa78caf(%rdi),%eax
  4019ad:       c3                      ret

00000000004019ae <setval_237>:
  4019ae:       c7 07 48 89 c7 c7       movl   $0xc7c78948,(%rdi)
  4019b4:       c3                      ret

00000000004019b5 <setval_424>:
  4019b5:       c7 07 54 c2 58 92       movl   $0x9258c254,(%rdi)
  4019bb:       c3                      ret

00000000004019bc <setval_470>:
  4019bc:       c7 07 63 48 8d c7       movl   $0xc78d4863,(%rdi)
  4019c2:       c3                      ret

00000000004019c3 <setval_426>:
  4019c3:       c7 07 48 89 c7 90       movl   $0x90c78948,(%rdi)
  4019c9:       c3                      ret

00000000004019ca <getval_280>:
  4019ca:       b8 29 58 90 c3          mov    $0xc3905829,%eax
  4019cf:       c3                      ret

00000000004019d0 <mid_farm>:
  4019d0:       b8 01 00 00 00          mov    $0x1,%eax
  4019d5:       c3                      ret
```


rtarget-touch2

writeup中要求使用start_farm到mid_farm之间的内容完成任务。

我们通过objdump查看相应函数的位置以及二进制内容，左图呈现的就是我们在这里可以使用的所有gadget了，思考如何将它们拼接成一段能够完成任务的程序。

注意我们在栈中保存的跳转位置不一定是每个指令的开头，比如，我们可以跳转到0x4019c3+2=0x4019c5的位置执行48 89 c7，即mov %rax，%rdi指令

rtarget-touch3

我们首先沿用ctarget中的思路：把我们在rtarget-touch2的思路沿用过来，然后把传入的整型值换成传入char *的指针，问题解决了……吗?

```c
#include <stdio.h>
#include <malloc.h>

void main(){
        void *p = (void *)malloc(8);
        char buf[8];
        printf("address(stack &P): %p\n",&p);
        printf("address(heap P): %p\n",p);
}
```

```
jizimo@linuxdemo:~/target1$ ./randomize.out
address(stack &P): 0x7ffd99c12428
address(heap P): 0x5cfdce8af2a0
jizimo@linuxdemo:~/target1$ ./randomize.out
address(stack &P): 0x7ffcaf023128
address(heap P): 0x5b5a863832a0
```

rtarget-touch3

我们首先沿用ctarget中的思路：把我们在rtarget-touch2的思路沿用过来，然后把传入的整型值换成传入char *的指针，问题解决了……吗?

由于加入了栈随机化保护（原理如右图所示），我们无法传递绝对地址以实现攻击，必须使用相对地址。

THANKS

Tel:+86-021-62233586 Fax:+86-021-62606775
E-mail: ecnu@ecnu.com.cn Http: //www.ecnu.edu.cn