

课程实验报告

课 程 名 称： 计算机系统

实验项目名称： 课程实验四

专 业 班 级： 计科1907

姓 名： 杨杰

学 号： 201908010705

指 导 教 师： 黄丽达、赵欢

完 成 时 间： 2021 年 5 月 25 日

信息科学与工程学院

|  |  |
| --- | --- |
| **实验题目：**LAB4-buflab | |
| **实验目的：**本实验将帮助您详细了解IA-32调用约定和堆栈组织。它涉及到对buflab目录中的可执行文件bufbomb应用一系列缓冲区溢出攻击。注意:在本实验中，您将获得使用操作系统和网络服务器中的安全漏洞的常用方法之一的第一手经验。我们的目的是帮助您了解程序的运行时操作，并理解这种形式的安全缺陷的性质，以便您在编写系统代码时可以避免它。 | |
| 实验环境：个人电脑、ubuntu16.04 (32位)环境 | |
| **实验内容及操作步骤：**  **4.1 实验内容**  解压得到包含三个可执行文件的名为buflab-handout的目录：  Bufbomb：用来攻击的缓冲区炸弹程序;  makecookie：根据用户id生成一个“cookie” ;  hex2raw: 用来进行十六进制向二进制串转换的工具(可能所需输入并不是ASCII的可打印字符，借助此工具进行转换)；  使用命令行./bufbomb -u IDname运行bufbomb，程序会根据输入的IDname生成特定的cookie。实验过程中需根据实验要求，构造能够实现特定功能的输入序列，完成对应的五个任务。  **4.2 操作步骤** Level0 **（1）问题描述：** Level0任务是让bufbomb在getbuf执行其返回语句时执行smoke代码，而不是返回test。请注意，您的攻击字符串也可能损坏堆栈中与此阶段没有直接关系的部分，但这不会导致问题，因为smoke会导致程序直接退出。  **（2）分析：** 在bufbomb中存在函数test(),其调用getbuf()函数读取输入，并通过uniqueval()函数进行堆栈是否被破坏的检查,之后根据读取后的情况进行相应的输出。      同时，bufbomb文件中还存在函数smoke().  level 0即**改变程序控制流，使得test函数调用getbuf()后，在getbuf()返回时直接调用smoke()函数，而不是返回函数test().**  使用objdump -d,根据反汇编，查看getbuf反汇编代码：    查看getbuf函数，由“lea -0x28(%ebp),%eax”和“mov %eax,(%esp)”两句可知，getbuf()将%ebp栈基址往下0x28=40个字节的地址作为参数，调用Gets函数。Gets函数将以这个地址为起点，向上存储字符。因此只要输入的字符串将getbuf()的返回地址覆盖为smoke()函数第一条语句的地址，在函数返回时就能进入并执行smoke()函数。在汇编代码中查到smoke()函数入口地址为08048e0a.  由于0x0a是（\n），而Gets是通过换行符\n界定输入终止，所以还不能用0x0a。因此，可以使用08048e0b(push %ebp这步是没有必要的，因为smoke函数直接退出进程，不需要保存调用者栈帧)，所以读入的字符串可以为：    注意，虽然test中有检查堆栈破环的canary，但任务目的是在getbuf结束后直接调转至另一函数，而没有执行后续的堆栈是否被破坏的检查，所以可直接构造超出数组长度的字符串来覆盖返回地址，使其指向目标函数的地址。  **（3）结果验证：**    程序输出结果为“You called smoke()”，说明getbuf执行其返回语句时执行smoke代码，而不是返回测试，通过测试。  **Level1**  **(1)问题描述：**  Level1的任务是让bufbomb执行fizz的代码，而不是返回test。并且将参数值设置为cookie以下是fizz函数:    **（2）分析：**  (1)对于有参数的被调函数，函数调用之前会将参数按从右至左的顺序入栈，之后在被调函数中通过%ebp+8、%ebp+12等地址获得函数调用的实参。  (2)函数调用指令call会将函数的返回地址入栈，被调函数会将原函数的帧指针即存储在%ebp中的值入栈，并更新%esp，从而使%ebp指向被调函数的栈帧，这样新%ebp指向的地址与函数的参数存放处间隔保存着调用函数%ebp和返回地址，故可以通过%ebp+8获得函数的第一参数。               左侧箭头标识%esp位置。函数返回时，首先将%ebp值赋值给%esp,则栈顶为位置(1).之后pop %ebp，将%ebp还原，%esp在位置(2)。最后ret指令恢复返回地址，%esp指向位置(3)；由于需返回fizz函数，故返回地址已被修改为fizz的地址。注意这里没有call指令，没有返回地址入栈。fizz函数按正常流程执行，其栈自红线处开始。先将%ebp入栈，新的%ebp位置如图所示。fizz函数正常按照%ebp+8的位置取其参数，故图示栈中的位置(1)应被覆盖为cookie值；  故构造的输入字符串应为：44个填充字节 + fizz函数起始地址 + 4个填充字节 + cookie值.  fizz函数的起始地址可使用gdb查看:    知道fizz函数起始地址为0x08048daf.  cookie值为bufbomb生成的值，这里注意使用小端法书写即可。    **（3）结果验证：**    程序输出结果为“You called fizz(0x5f0bf1f7)”，说明getbuf执行其返回语句时执行fizz代码，并且将val的值设置为cookie，通过测试。  **Level2**  **（1）问题描述：**  Level2的任务是让bufbomb执行bang的代码，而不是返回test。在此之前，必须将全局变量global\_value设置为用户id的cookie。在堆栈上设置bang的地址，然后执行ret指令，以跳转到bang的代码。以下是bang函数代码：    **（2）分析：**  任务的关键在于**如何构造机器代码，使得程序跳转至输入的机器代码处执行。**  构造输入字符串的过程：   1. 全局变量global\_value在程序执行的过程中逻辑地址不发生变化，可直接在gdb中得到其地址，使用mov指令对其进行赋值；     (2)将getbuf函数的返回地址修改，指向构造的机器代码的开始处，这里即buf数组的起始地址；  (3)由于getbuf函数的返回地址已经被用于指向输入的机器代码，故跳转至bang函数的实现需要使用额外的指令。这里由于程序是已经编译好的，所以bang函数的逻辑地址不变，故可以直接使用逻辑地址调用。使用push 将bang函数地址入栈，再使用ret指令进行跳转。(push指令将数据放置在栈顶，ret取栈顶的数据并将其作为地址进行跳转)；  (4)这里需要注意的是，之前的level 0与level 1，机器代码存放在代码段，由PC指示，数据操作在栈上，由%esp指示。level 2中第一次跳转后，正在执行的机器代码位于栈上的缓冲区中，由PC指示，数据操作也在栈上，由%esp指示，这里需要注意两者的区别，前者是用于执行的，后者是用于操作的。图示为getbuf函数ret指令之后%esp和PC的位置。    构造输入字符串：可执行的机器代码 + 填充字符 + 指向输入机器代码的地址。  通过gdb得到bang函数的起始地址为0x08048d52。    同样在bang函数的反汇编中，将0x0804d104与0x0804d10c处的值进行了比较，查看地址0x0804d104，  发现存放的是cookie，则0x0804d10c处即为全局变量global\_value的值。  在getbuf函数内部设置断点，并运行至函数内部，得到buf数组的起始地址为0x55683ac8.  (这里注意要运行至getbuf内部是由于需要使得%ebp指向的是getbuf的栈帧，这样%ebp-40才是buf数组的首地址，否则直接输出%ebp-40可能指向的是其他地方)  可以使用layout regs命令同时查看汇编代码和寄存器值，使用ni逐条汇编代码执行。      构造的可执行代码为：    可以将上诉汇编指令进行编译, 再使用objdump来得到所需的机器代码的十六进制表示。    实际使用的输入字符串如图所示，其中可执行代码（16bytes）+ 填充字符（28bytes）+ 数组首地址（4bytes）    **（3）结果验证：**    程序输出结果为“You set global\_value to 0x5f0bf1f7”，说明getbuf返回时执行了我设置的机器代码，并且将global\_value的值设置为cookie，通过测试。  **Level3**  **(1)问题描述：**  Level3的任务是要修改getbuf()函数的返回值(正常状态为0x1)为你的cookie值,然后让函数正常返回到test.   1. **分析：**   需要注意以下几点：  (1)构造的机器指令是存放在getbuf的缓冲区中，想要执行输入的构造代码，只有修改getbuf函数返回时的地址，注意当跳转至构造的代码处执行时，getbuf是已经结束了，返回值1存放在寄存器%eax中；(正是结束时的ret指令才跳转至修改后的地址处)  (2)回想函数调用过程，call指令调用函数时将返回地址放置在栈顶，进入函数后的第一步为保存%ebp，这样在覆盖修改返回地址时必将保存的%ebp也覆盖掉了。在getbuf函数结束时，会将%esp的值赋值为getbuf栈帧指针%ebp的值(mov指令)，之后将保存的%ebp值赋值给寄存器%ebp。前面所述，保存的%ebp在覆盖返回地址时已经被覆盖，故此时%ebp会是一个废值；  (3)由于题目的要求是正常返回test函数，而该函数存在一定的对缓冲区覆盖的检查(uniqueval函数),故可能需要注意恢复保存的%ebp；  如上所述，**构造的字符串应完成的功能为：**（1）修改存放返回值的寄存器%eax； （2）恢复寄存器%ebp的值为正常值，这里即test函数的栈帧； (3)将getbuf正常返回地址放置在栈顶，并通过ret指令返回test函数。  通过gdb调试，在getbuf函数内部设置断点，查看保存的返回地址、保存的%ebp等信息。p $ebp获得getbuf栈帧指针的信息，再使用x /2xw $ebp获得地址%ebp处开始的连续两个4字节空间的值（回忆一下getbuf的栈结构，这两个空间存放的即为保存的%ebp和返回地址）。得到保存的%ebp为0x55683b20,return address为0x08048e50.      构造的可执行代码为：    可以将上述汇编指令进行编译, 再使用objdump来得到所需的机器代码的十六进制表示。    构造的字符串序序列如下，其中构造代码（11字节） + 填充字符（29字节） +保持保存的%ebp不变（4字节）+ 修改的返回地址，即buf数组起始地址（4字节）    **（3）结果验证：**    程序输出结果为“getbuf returned 0x5f0bf1f7”，说明getbuf返回时执行了设置的代码，返回cookie到test。  **Level4**  **（1）问题描述：**  当在Nitro模式下运行时，bufbomb需要您提供字符串5次，并且它将执行getbufn 5次，每次使用不同的堆栈偏移量。你的攻击字符串必须让它每次都返回你的cookie。我们的任务是对于5次getbufn函数调用，都能使得getbufn函数返回cookie到testn。  **（2）分析：**  对于一个给定的程序而言，程序每次运行时尤其是被不同用户运行时使用的栈位置是不同的。造成栈位置变化的原因有很多，其中一个是由于程序在运行时，所有必要的环境变量都以字符串的形式被放置在栈的底部(高地址单元)。对于不同值的环境变量，其所需要的栈空间自然不同，从而使得栈位置变化，对于不同用户而言这一点更为显著。相应的，程序自然运行与在gdb环境下运行的栈位置也可能不同，因为gdb本身运行所需的部分数据被放置在了栈中。  getbuf函数内置了使栈空间稳定的特性，从而使得进行缓冲区攻击时能够直接获得固定的所需要的地址数据，并采用直接利用的方式写入机器代码中，这也大大降低了实现难度。而这在实际应用情况下是过分理想的。在level 4环节，用户需要在启动bufbomb时使用 -n 选项，从而使得栈空间不再稳定，并在此基础上进行基于缓冲区溢出原理的实验。  程序运行时启用了 -n 选项时，程序在读取输入时会启用 getbufn函数(而不是前面的getbuf)。getbufn函数有与getbuf相似的功能，但前者输入数组的长度为512字节。    调用getbufn函数之前，程序会先在栈上分配一个随机长度的空间，从而使得getbufn函数的栈空间在不同调用情况下不再是固定的，实际上%ebp的差值达到±240。在应用 -n 选项的情况下，程序会要求提交输入字符串 5 次，5次输入会面对5个不同的栈空间，并要求每次都成功返回cookie值。level 4的任务与level 3一致，即要求getbufn函数返回调用函数testn时返回cookie值，而不是常规的1.  程序的运行过程加入了栈随机化的操作，即在程序调用之前，先分配一个随机大小的空间，这个空间程序并不使用，但是长度不定，从而使得每次运行时的栈空间地址产生变化(主要是在栈相对结构不变的情况下，各个栈中元素的地址发生了变化)。这一操作的显著影响是之前所采用的使用固定的返回地址覆盖getbuf返回地址的方法受到限制。由于每次栈空间不同，则输入的机器代码的起始位置也不同(回忆上文，每次均是将机器代码放在输入字符串的开始位置，这样每次修改返回地址为输入数组的起始地址即可执行构造的代码，其中输入数组起始地址是固定的)，相应的直接指定出构造代码的地址变得不可行。  这里对于栈随机化的破解可以借助“空操作雪橇”(nop sled)的技巧。所谓nop sled是在构造的机器代码之前加入nop指令（no operation的缩写，机器码为 0x90），其作用为仅将PC增加而不执行任何操作。在这种情况下，只要覆盖的地址能够指向nop序列所处的任意一个地址，就可以顺序执行nop指令，直到遇到真正构造的机器代码，这样的情况下，对于用于覆盖的返回地址的要求就降低了。  即构造出的字符串为：nop指令串 + 构造的机器代码 + 返回地址。    如图所示，由于随机分配的地址空间的存在，栈上各个元素的地址会发生变化，从而使得用于覆盖的返回地址难以确定。使用空操作雪橇时，会在构造的代码之前填入nop指令。题中的缓冲区有512个字节，同时%ebp的差值为±240。正常情况如上图，则存在一个区间，只要返回地址为该区间内的地址，则总可以通过nop指令向上“滑行”至真正执行的构造代码处，从而实现攻击。  查看getbufn函数的实现，可知数组的分配的长度为520个字节(0x208)，覆盖返回地址需要填充 520（数组长度）+ 4（保存的%ebp）+ 4（返回地址）= 528个字节。    通过 layout regs查看%ebp的值，通过 x /2xw $ebp 查看保存的%ebp和返回地址的值。      解题思路如下：  (1)为达到能返回cookie值至testn函数的目的，同样需要getbufn修改返回地址使其执行构造的代码，完成包括修改返回值、恢复%ebp、返回testn函数这三个步骤；  (2)在步骤（1）中，修改返回值即%eax与返回testn函数的操作与level 3是一样的。总是将返回值修改为cookie，返回testn函数的地址也总是不变的（注意这里程序应用的是栈随机化的操作，影响的是栈空间上的地址，可执行代码是存放在代码段，在题设环境下是不受影响的）；  (3)关于如何恢复被覆盖%ebp的问题。栈随机化是在栈上分配一段不定长的内存空间使得栈中元素的地址发生变化。但是，由于程序总是执行相同的操作，使得在不同的执行情况下，程序所使用的栈中元素的相对位置(距离)不发生变化，可尝试在此前提下恢复%ebp。恢复过程是由输入的构造代码执行的，此时%ebp已经被赋予了“废值”(见level 3分析)，但%esp是有效的值，可以通过%esp推出被覆盖的保存的%ebp的值。从上面获得的保存的testn函数%ebp的值(0x55683b20)和getbufn%ebp的值(%0x55683af0)，在构造代码执行时，testn%esp应为getbufn%ebp+8 = 0x55683af0 + 8 = 0x55683af8 ，则可以看到差值为 0x55683b20 - 0x55683af8 = 0x28.上述地址在不同运行情况下是会改变的，但其相对差值不变，故总是可以通过执行%esp + 0x28得到原有的被破坏的%ebp值。  以下是借助gdb调试程序的过程中两次运行时的栈空间的变化。可以看到，在两次运行中，%ebp和保存的%ebp改变了，而返回地址没有改变，这是由于返回地址指向的是位于代码段的固定位置处的代码，不受栈随机化的影响，但位于栈上的数据则受到了影响。      查看五次buf数组首地址：      得到的最大地址为0x55683988，所以可以考虑将buf地址0x55683988作为跳转地址，将有效机器代码置于跳转地址之前，并将其他所有字符用作nop指令，此时所有五个buf地址的写入都能在满足跳转到0x55683988后到达机器代码。  构造的可执行代码为：    可以将上述汇编指令进行编译, 再使用objdump来得到所需的机器代码的十六进制表示。    构造的输入字符串为 ：nop指令串（506字节） + 构造指令（18字节） + 用于覆盖的新地址（4字节）    **（3）结果验证：**    在-n模式下对于5次getbuf函数调用，都能使得getbuf函数返回cookie到testn，5组函数调用均通过了测试。  **实验结果及分析：**  5个任务全部通过：    **收获与体会：**  1.这次实验我个人感觉Level4比较麻烦。字符数组扩大，nop指令运用以及数组起始位置与我们设计的函数起始位置关系等都是难点。这次实验让我感到新奇的是我们可以自己写一个简单的汇编代码来攻击程序，还是比较有意思的。  2.通过本次实验对gdb调试的使用方法有了进一步了解，调试能力得到了锻炼。尤其学会了使用layout这个工具，能够同时查看汇编代码和寄存器值。  3.本次实验对函数调用有了更深一步的理解。  4.学习到了溢出攻击可以将特定的机器指令插入到可执行程序中从而修改程序执行方式的新方法。 | |
| 实  验成绩 |  |

实验报告撰写说明

1．实验题目和目的

请从实验指导资料中获取。

2．实验步骤和内容

包括：

（1）本次实验的要求；

（2）源程序清单或者主要伪代码；

（3）预期结果；

（4）上机执行或调试结果：包括原始数据、相应的运行结果和必要的说明（截图）；

3．实验体会

调试中遇到的问题及解决办法；若最终未完成调试，要试着分析原因；调试程序的心得与体会；对课程及实验的建议等。