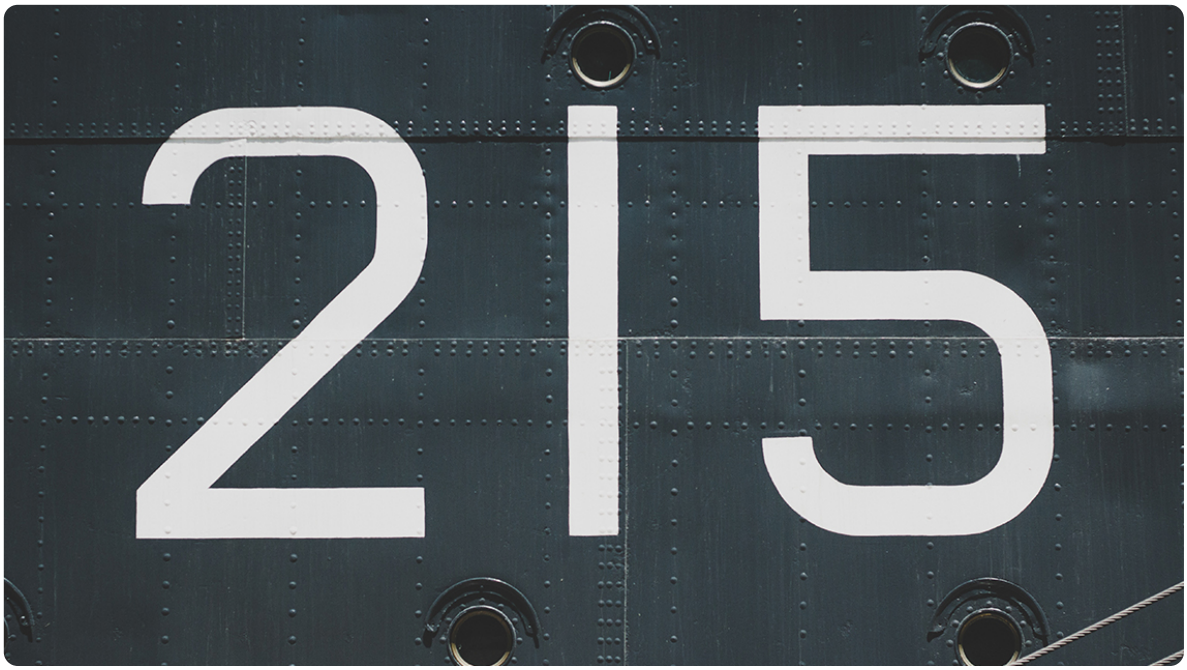
33 | 整数的运算有哪些安全威胁？

范学雷 2019-03-20



00:00 12:17

讲述：刘飞　　大小：28.16M

在我的日常工作中，有一类错误，无论是原理还是后果，我都十分清楚。但是写代码的时候，这类错误曾经还是会反复出现。如果不是代码评审和代码分析环节的校正，我都很难意识到自己的代码中存在这样的缺陷。今天，我想和你聊聊，那些“**道理我都懂，但代码就是写不好**”的老顽固问题。

你不妨先停下来想一想，你有没有类似的经历？ 又是怎么解决的呢？

评审案例

HTTP 连接经常被中断或者取消，如果客户端已经获得一部分数据，再次连接时，应该可以请求获取剩余的数据，而不是再次请求获取所有数据。这个特性背后的支持协议就是 HTTP 范围请求协议（RFC 7233）。

比如下面的例子，客户端请求服务器返回 image.jpg 图像的前 1024 个字节。其中“bytes=0-1023”表示请求传输的数据范围是从 0 到第 1023 位的字节（0-1023），以及“-512”表示请求传输数据的最后 512 个字节（-512）。

复制代码

```
1 GET /image.jpg HTTP/1.1
  Host: www.example.come
```

```
Range: bytes=0-1023,-512

```

如果服务器支持该协议，就会只传输图像的指定数据段。响应消息的代码大致如下所示：

复制代码

```
HTTP/1.1 206 Partial Content
Content-Type: multipart/byteranges; boundary=THIS_STRING_SEPARATES
Content-Length: 2345

--THIS_STRING_SEPARATES
Content-Type: image/jpeg
Content-Range: bytes 0-1023/2048
...
(binary content)

--THIS_STRING_SEPARATES
Content-Type: image/jpeg
Content-Range: bytes 1535-2047/2048
...
(binary content)
--THIS_STRING_SEPARATES--

```

如果服务器端使用下属的代码验证请求数据的指定数据段（C 语言），你来看看可能存在什么严重的问题?

复制代码

```
/**
 * Check if the requested range is valid.
 *
 * start: the first byte position of the range request
 * end: the end byte position of the range request
 * contentLength: the content length of the requested data
 * sum: the sum of bytes of the range request
 */
bool isValid(int start, int end, int contentLength, int* sum )) {
    if (start < end) {
        *sum += end - start;
        if (*sum > contentLength) {
            return false;
        } else {
            return true;
        }
    } else {
        return false;
    }
}

```

案例分析

上面的代码简化自 Nginx HTTP 服务器关于 HTTP 范围请求协议在 2017 年 6 月份的实现版本。由于讨论的需要，我较大幅度地删减和修改了原来的代码。如果有兴趣，你可以阅读并比较2017 年 7 月份的实现版本和2017 年 6 月份的实现版本，看看都有哪些比较有意思的改动。

上面的代码有许多问题，其中有一个致命的魔鬼藏在下面的一行代码中：

复制代码

```
1         *sum += end - start;
2
```

我们假设数据长度为 1024 个字节，HTTP 请求的范围有两段，第一段的请求字节数是一个正常的数据 （比如，1023），而第二段的请求字节数的是一个巨大的数据（比如，INT_MAX - 23），以至于两段数据相加时，发生了整数溢出。本来应该是一个很大的数，可是这个数超出了整数可以表达的范围，结果就发生整数溢出，变成了一个很小的数 （比如，（1023 + INT_MAX - 23），结果是 999 或者 -2147482649。这样就通过了上述的 HTTP 请求范围验证。

第一段数据也许可以正常使用。然而，由于实际的数据长度不足以满足第二段数据范围请求，就有可能出现非常复杂的状况。 比如说，为了加快反应速度，提高服务器效率，很多 HTTP 服务器都使用了缓存技术，Nginx 也不例外。

如果使用缓存技术，数据可能并不是直接从原始数据源读取，而是读取缓存的数据，而缓存的数据是一个临时的大集合，可能包括各种各样的数据，包括敏感数据。

如果发生读取范围溢出，目标数据段之外的缓存数据可能被读取。而目标数据段之外的数据，可能并没有授权给这个用户使用，这样就可能发生敏感信息的泄露。甚至通过设计，攻击者也有可能更改目标数据以外的非授权数据。这样就间接地操纵了服务器。

关于这个安全漏洞的更多描述，请参阅CVE-2017-7529。这是一个通用缺陷评分系统评分为7.5 的严重安全缺陷。由于 Nginx 的广泛部署与使用，该漏洞影响了一大批安装了 Nginx 服务器的系统。

出问题的表达式使用太普遍了，谁能想到会出现这么严重的安全漏洞呢。我们感兴趣的问题是：**整数的加法，如此普遍的运算，如果都这么脆弱，到底该怎么办才好?**

整数的陷阱

一个优秀的语言，一定是便于学习和使用的语言。语言语法的设计通常会考虑我们的现有习惯，降低使用门槛。比如，加法运算通常会设计成我们最熟悉的样子：

复制代码

```
1   a = b + c;
2
```

我们如此熟悉这种表达式，一旦需要加法运算，这种表达式一定首先从我们的脑袋里自动跳出来。我们毫无察觉，也不会有意识地去追究这样一个从学习算数起就开始使用的表达式到底有什么问题。

那么到底有什么问题呢？

第一个问题是，我们使用上述表达式时，每个数字都可以是无限大的；然而在计算机的世界里，数字大小都是有限制的。比如，整数一般使用 32 位（bit），或者 4 个字节（byte）来表示，整数的大小不能超过 32 位能够容纳的范围。

整数运算，可能溢出

如果我们使用一位（bit）来表达一个整数，那么通常就会是下面这种情况：

复制代码

```
0 + 0 = 0
0 + 1 = 1
1 + 1 = 0

```

注意，1 + 1 的结果不是我们习以为常的 2，而是 0。这是为什么呢？因为 2 太大了，一位的空间不够用的，所以就“溢出”了。“溢出”导致这个运算的结果是 0，而不是预想的 2。

再比如说，我们表示时间的时候，如果采用 12 小时制，12 点过一个小时就是 1 点，而不是 13 点；如果采取 24 小时制，24 点过两个小时就是 2 点钟，而不是 26 点。

如果我们对于一位的数，以及 24 小时制还算“清醒”的话，那么对于 32 位或者 64 位的数，可能就没那么重视了。你的代码里有没有涉及到整数的运算？有没有潜在的溢出问题？我是经常会掉入这个陷阱的。

整数溢出的问题，曾经在 1995 年导致火箭的坠落； 在 2016 年导致错误地签发了四千多万美元（最高限额原为 1 万美元）的博彩奖券。还有绵延不绝的，你我知道抑或不知道的软件安全漏洞。

我们更关心的是，该怎么避免这类错误？

首先最重要的，是要借助软件开发的机制，减少代码错误。比如我们在专栏开始讲的借助重重关卡减少错误。虽然我在编写代码的时候会时常忘却这个问题，但在评审代码时，有时候还能够记住这个问题的危害。多一双眼睛，就多了一处关卡。

然后，还要了解一些小技巧，我们看看都有哪些？

1.**比较运算，选择“比较小的数”**。

如果表达式出现在比较运算符的两侧，选择产生较小的数的运算。比如下面这段代码：

复制代码

```
// a, b, c are positive integers
if (a < (b + c)) {    // (b + c) can overflow
    // snipped
}

// a, b, c are positive integers
if ((a - b) < c) {    //  no overflow
    // snipped
}

```

这个例子适用于正数的运算，如果是负数呢？ 如果不确定是整数还是负数，该怎么办？这个问题，我留给你去思考。

2.**限定数的范围，选择冗余的空间。**

如果现实需要 32 位的整数，就选择 64 位的存储空间（也就是说，使用 64 位的整数类型）进行运算。如果现实需要 31 位的整数，就选择 32 位的存储空间。限定了数的范围，一定要记得检查数据的范围，千万不可超越这个范围。

复制代码

```
private static int MAX_DATA_SIZE = 16384;  // 2^14 bytes at mosts
private final ByteBuffer cache =
    ByteBuffer.allocate)(MAX_DATA_SIZE);   // limit the capacity

static int receive(byte[] data) {
    if (data == null || data.length == 0) {    // input check
        // snipped
    } else if (data.length > MAX_DATA_SIZE) {  // check the range
        // throw exception, snipped
    } else {
        if (data.length > cache.remaining()) { // check the add-up
            // throw exception, insufficient space, too much data
        }

        if ((data.length + 1024) > cache.remaining()) {
            // safe '+', as the numbers are limited to 2^14.
            // snipped
        }

        // snipped
    }
    // snipped
}

```

上面例子中的加法运算就是安全的。因为运算涉及的数据都被限定在 14 位范围内，而两个数相加，最多不超过 15 位。由于我们使用了 32 位的整数作为数据的类型，那么 15 位的数据就不会产生溢出问题。

但是，这种方法要求我们时刻绷紧神经，仔细地定义、检查每个数据的限定范围，对我们自身要求相对有点高。所以涉及到比较运算，我还是建议使用“比较小的数”的办法。

复制代码

```
-           if ((data.length + 1024) > cache.remaining()) {
+           if ((data.length - cache.remaining()) > 1024) {

```

3.**检查数据溢出。**

检查数据溢出，虽然代码看起来有点多，但这总是一个有效可行的办法。比如，评审案例中的缺陷修复，就采取了类似如下的修改：

复制代码

```
+       if (*sum > (NGX_MAX_OFF_T_VALUE - (end - start))) {
+           return false;
+       }
        *sum += end - start;

```

从 Java 8 开始，Java 提供了数据溢出保护的运算方法，比如 Math.addExact(int, int)，执行两个整数相加的运算，如果有整数溢出，就会抛出 ArithmeticException 的异常。这些方法也许并不如直接使用运算符直观，但是它们提供了额外的保护机制。如果我们不能确定溢出是否会发生，使用这些方法可以让我们的代码获得更加深度的保护。

复制代码

```
    int sum = Math.addExact(a, b);   // sum = a + b

```

整数溢出的危害是整数太大了，超出了许可边界。如果整数还没有大到溢出的程度，但也足够大，同样是一个值得警惕的风险。

其他的语言也可能有类似的数据溢出保护方法，欢迎你在评论区留言分享。

整数，可能太大

不比数学世界里的整数，软件世界里的整数，大都具有现实的意义。 比如，整数可以代表人民币，可以代表美元，也可以代表文件长度，代表内存空间，代表运算能力。 **一旦抽象的整数被赋予了现实意义，就会有现实的约束。**比如，1 亿元人民币虽然是个小目标，但你要是用来发人手一份的红包，也许就有点大了。再比如，针对 32 位整数，虽然现代计算机已经可以毫无障碍地表达这个数据了，但要是用来分配应用内存，这个数就有点大了。

我们要特别警惕大量内存的动态分配。比如说很多协议的设计都会指定待传输数据的大小，而接收端需要按照指定的大小来接收紧接着的数据流。有时候需要分配内存，来存储、处理接收的数

据。其中有一种实现，接收到指定大小的数据后，接收端再根据指定的大小分配内存，然后把后续的数据存储在该内存里。指定数据接收完毕，再开始处理该数据。你看出其中的问题了吗？

一个比较典型的安全攻击是，攻击者会设置非常大的待传输数据的大小（比如 2^31），但是只传输非常小的数据（比如 1 个字节），然后在很短的时间内，发送多个请求（一个机器或者多个机器）。一个 16G 内存的服务器，如果有 8 个这样的请求，内存就红灯高挂了；有 10 个这样的请求，内存可能就要挂免战牌了。这就破坏了服务器的“可用性”，算是比较严重的安全事故。

好了，这就是今天的内容，算是关于数的问题的敲门砖，更多的、更深入的话题，可以阅读 CWE-190，或者留言与我一起讨论。

小结

通过对这个评审案例的讨论，我想和你分享下面几点个人看法。

1. 数值运算，理论结果可能会超出数值类型许可的空间，进而发生实际结果的溢出。
2. 抽象的数据一旦有了现实意义，便有了具体的现实约束，我们一定要考虑这些约束。
3. 很多问题和我们的习惯并不相符，要通过制度设置来减少由于人的固有缺陷带来的经常性问题。

一起来动手

我们一起讨论了一些整数的问题，你愿不愿意总结下浮点数的问题？ 我们使用了 C 语言和 Java 语言的示例，你了解其他语言关于整数溢出的技术和经验吗？

欢迎你来评审下面的这段 C 语言代码：

复制代码

```
int copy_something(char* buf, int len){
    char kbuf[800];

    if(len > sizeof(kbuf)){
        return -1;
    }

    return memcpy(kbuf, buf, len);
}

```

或者这段 Java 代码：

复制代码

```
public static int mixed(int addOn, int multiplied, int scale) {
    return addOn + (multiplied * scale);
}

```

或者，是下面这段我们已经非常熟悉的 Java 代码：

复制代码

```
import java.util.HashMap;
import java.util.Map;

class Solution {
    /**
     * Given an array of integers, return indices of the two numbers
     * such that they add up to a specific target.
     */
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i };
            }
            map.put(nums[i], i);
        }
        throw new IllegalArgumentException("No two sum solution");
    }
}

```

针对上面三段代码，你有什么改进的建议呢？可以在评论区与我分享你的想法。

另外，分享一个最近（2019 年 3 月）发生的和整数有关的安全事故。

安全起见，一个数字证书的序列号应该至少有 64 位随机数，少一位都不行。如果你对整数足够敏感的话，就会知道 64 位是一个特殊的位数。长整型（long）通常使用 64 位字节来表述。数字证书的序列号能不能使用 64 位的长整型呢？这就是个坑！

为了保证序列号是正数，64 位的长整型，只有 63 位有效的数字。因为，64 位长整型中，有一位是用来表示数据正负的。所以，长整型就不能用做数字证书的序列号。

这个坑，就是有人踩了。有数百万张数字证书仅使用了 63 位的随机数。按照业界规则，这些数字证书需要问题发现 5 天以内撤销，重新签发。这几乎是一项不可能完成的任务。2019 年 3 月和 4 月，很多公司都会面临数字证书更新的问题。

如果你觉得这篇文章有所帮助，欢迎点击“请朋友读”，把它分享给你的朋友或者同事，一起来交流。

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表　　0/2000字　　提交留言

精选留言(2)

hua168

第三段代码

for (int i = 0; i < nums.length; i++) {...}

把nums设置为i的最大值，这样i++就溢出

2019-03-20

hua168

```
public static int mixed(int addOn, int multiplied, int scale) {
    return addOn + (multiplied * scale);
}
```

1. (multiplied * scale)如果相乘大于默认的int类型最大值会溢出
2. addOn + (multiplied * scale)相加大于int最大值也会溢出

2019-03-20