

37 | 边界，信任的分水岭

2019-03-29 范学雷 来自北京

《代码精进之路》



边界是信息安全里一个重要的概念。如果不能清晰地界定信任的边界，并且有效地守护好这个边界，那么编写安全的代码几乎就是一项不可能完成的任务。

评审案例

计算机之间的通信，尤其是建立在非可靠连接之上的通信，如果我们能够知道对方是否处于活跃状态，会大幅度地提升通信效率。在传输层安全通信的场景下，这种检测对方活跃状态的协议，叫做心跳协议。

心跳协议的基本原理，就是发起方给对方发送一段检测数据，如果对方能原封不动地把检测数据都送回，就证明对方处于活跃状态。

下面的数据结构，定义的就是包含检测数据的通信消息。

```
1 struct {
2     HeartbeatMessageType type;
3     uint16 payload_length;
4     opaque payload[HeartbeatMessage.payload_length];
5     opaque padding[padding_length];
6 } HeartbeatMessage;
```

[复制代码](#)

其中，type 是一个字节，表明心跳检测的类型；payload_length 使用两个字节，定义的是检测数据的长度；payload 的字节数由 payload_length 确定，它携带的是检测数据；padding 是随机的填充数据，最少 16 个字节。

如果愿意回应心跳请求，接收方就拷贝检测数据（payload_length 和 payload），并把它封装在同样的数据结构里。

下面的这段代码（函数 process_heartbeat，为便于阅读，在源代码基础上有修改），就是接收方处理心跳请求的 C 语言代码。你能看出其中的问题吗？

```
1 int process_heartbeat(
2     unsigned char* request, unsigned int request_length) {
3     unsigned char *p = request, *pl;
4     unsigned short hbtype;
5     unsigned int payload_length;
6     unsigned int padding_length = 16; /* Use minimum padding */
7
8     /* Read type and payload length first */
9     hbtype = *p++;
10    payload_length = ((unsigned int)(*p++)) << 8L |
11                    ((unsigned int)(*p++));
12    pl = p;
13
14    // produce response heartbeat message
15    unsigned char *response, *bp;
16
17    /* Allocate memory for the response, size is 1 bytes
18     * message type, plus 2 bytes payload length, plus
19     * payload, plus padding
20     */
21    response = malloc(1 + 2 + payload_length + padding_length);
22    bp = response;
23}
```

[复制代码](#)


```

24  /* Enter response type, length and copy payload */
25  *bp++ = 1; /* 1: response heartbeat type */
26  *bp++ = (unsigned char)((payload_length >> 8L) & 0xff);
27  *bp++ = (unsigned char)((payload_length      ) & 0xff);
28  memcpy(bp, pl, payload_length);
29  bp += payload_length;
30
31  // snipped
32
33  return 0;
34 }

```

上面这段代码，读取了请求的 `payload_length` 字段，然后按照 `payload_length` 的大小，分配了一段内存。然后，从请求数据的 `payload` 指针开始，拷贝了和 `payload_length` 一样大小的一段数据。这段数据，就是要回应给请求方的检测数据。按照协议，这段数据应该和请求信息的检测数据一模一样。

比如说吧，如果心跳请求的数据是：


 复制代码

```

1  type:           0x01
2  payload_length: 0x00, 0x05 // 5
3  payload:        {0x68, 0x65, 0x6c, 0x6c, 0x6f}; // 'hello'
4  padding:        {0xCF, 0xED, ...};

```

按照协议和上面实现的代码，心跳请求的回应数据应该是：

 复制代码

```

1  type:           0x01
2  payload_length: 0x00, 0x05 // 5
3  payload:        {0x68, 0x65, 0x6c, 0x6c, 0x6f}; // 'hello'
4  padding:        {0x07, 0x91, ...};

```


这看起来很美好，是吧？可是，如果请求方心有图谋，在心跳请求数据上动了手脚，问题就来了。比如说吧，还是类似的心跳请求，但是 `payload_length` 的大小和真实的 `payload` 大

小不相符合。下面的这段请求数据，检测数据还是只有 5 个字节，但是 payload_length 字段使用了一个大于 5 的数字。

 复制代码

```
1 type:          0x01
2 payload_length: 0x04, 0x00                // 1024
3 payload:       {0x68, 0x65, 0x6c, 0x6c, 0x6f}; // hello
4 padding:       {0xCF, 0xED, ...};
```

按照协议的本意，这不是一个合法的心跳请求。上面处理心跳请求的代码，不能识别出这是一个不合法的请求，依旧完成了心跳请求的回应。

 复制代码

```
1 type:          0x01
2 payload_length: 0x04, 0x00                // 1024
3 payload:       {0x68, 0x65, 0x6c, 0x6c, 0x6f, // 'hello
4                0xCF, 0xED, ...                // request padding
5                0x70, 0x72, 0x69, 0x76, 0x69, 0x76, 0x61, 0x74,
6                0x65, 0x20, 0x6b, 0x65, 0x79, 0x20,
7                ... }; // private key "...
8 padding:       {0x07, 0x91, ...};
```

心跳请求的真实检测数据只有 5 个字节，返回检测数据有 1024 个字节，这中间有 1019 个字节的差距。这 1019 个字节从哪儿来呢？由于代码使用了 memcpy() 函数，这 1019 个字节就是从 payload 指针 (pl) 后面的内存中被读取出来的。这些内存中可能包含很多敏感信息，比如密码的私钥，用户的社会保障号等等。

这就是著名的心脏滴血漏洞 (Heartbleed)，这个漏洞出现在 OpenSSL 的代码里。2014 年 4 月 7 日，OpenSSL 发布了这个漏洞的修复版。由于 OpenSSL 的广泛使用，有大批的产品和服务需要升级到修复版，而升级需要时间。修复版刚刚发布，像猎食者一样的黑客抢在产品和服务的升级完成之前，马上就展开了攻击。赛跑立即展开！仅隔一天，2014 年 4 月 8 日，加拿大税务局遭受了长达 6 个小时的攻击，大约有 900 人的社会保障号被泄漏。2014 年 4 月 14 日，英国育儿网站 Mumsnet 有几个用户帐户被劫持，其中包括了其首席执行官的账

户。2014 年 8 月，一家世界 500 强医疗服务机构透露，心脏滴血漏洞公开一周后，他们的系统遭受攻击，导致四百五十万条医疗数据被泄漏。



【图片来自 <http://heartbleed.com/>,
<https://en.wikipedia.org/wiki/Heartbleed#/media/File:Heartbleed.svg>】

案例分析

没有检查和拒绝不合法的请求，是心脏滴血漏洞出现的根本原因。这个漏洞的修复也很简单，增加检查心跳请求的数据结构是否合法的代码就行了。

下面的代码就是修复后的版本。修复后的代码，加入了对心跳请求 payload_length 的检查。

 复制代码

```
1 int process_heartbeat(  
2     unsigned char* request, unsigned int request_length) {  
3     unsigned char *p = request, *pl;  
4     unsigned short hbtype;  
5     unsigned int payload_length;  
6     unsigned int padding_length = 16; /* Use minimum padding */  
7  
8     /* Read type and payload length first */  
9     if (1 + 2 + 16 > request_length) {  
10         /* silently discard */  
11         return 0;  
12     }  
13  
14     hbtype = *p++;  
15     payload_length = ((unsigned int)(*p++)) << 8L |  
16                     ((unsigned int)(*p++));  
17  
18     if (1 + 2 + payload_length + 16 > request_length) {  
19         /* silently discard */  
20         return 0;  
21     }  
22     pl = p;  
23  
24     // produce response heartbeat message  
25     unsigned char *response, *bp;  
26  
27     /* Allocate memory for the response, size is 1 bytes  
28      * message type, plus 2 bytes payload length, plus  
29      * payload, plus padding  
30      */  
31     response = malloc(1 + 2 + payload_length + padding_length);  
32     bp = response;
```

```
33  /* Enter response type, length and copy payload */
34  *bp++ = 1; /* 1: response heartbeat type */
35  *bp++ = (unsigned char)((payload_length >> 8L) & 0xff);
36  *bp++ = (unsigned char)((payload_length      ) & 0xff);
37  memcpy(bp, pl, payload_length);
38  bp += payload_length;
39
40  // snipped
41
42  return 0;
43 }
44
```

如果比较下 `process_heartbeat()` 函数修复前后的实现代码，我们就会发现修复前的危险性主要来自于两点：

1. 没有检查外部数据的合法性（`payload_length` 和 `payload`）；
2. 内存的分配和拷贝依赖于外部的未校验数据（`malloc` 和 `memcpy`）。

这两点都违反了一条基本的安全编码原则，我们在前面提到过这条原则，那就是：[🔗 跨界的数据不可信任](#)。

信任的边界

不知道你有没有这样的疑问：类似于 `memcpy()` 函数，如果 `process_heartbeat()` 函数的传入参数 `request_length` 的数值，大于传入参数 `request` 实际拥有的数据量，这个函数不是还有内存泄漏问题吗？

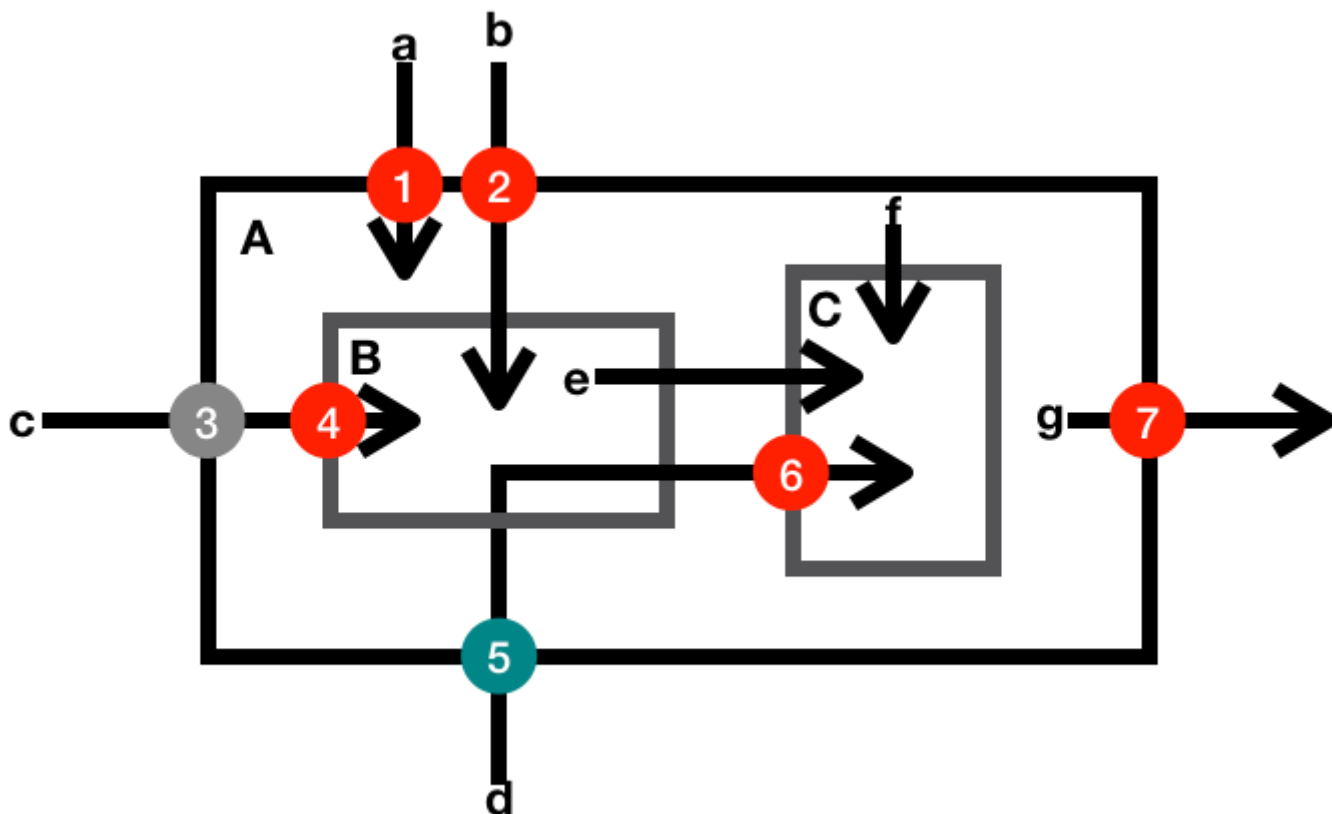
如果独立地看上面的代码，这样的问题是有可能存在的。但是，`process_heartbeat()` 是 OpenSSL 的一个内部函数，它的调用代码，已经检查过 `request` 容量和 `request_length` 的匹配问题。所以，在 `process_heartbeat()` 的实现代码里，我们就不再操心这个匹配的问题了。

对一个函数来说，到底哪些传入参数应该检查，哪些传入参数不需要检查？这的确是一个让人头疼的问题。

一般来说，对于代码内部产生的数据，我们可以信任它们的合法性；而对于外部传入的数据，就不能信任它们的合法性了。外部数据，需要先检验，再使用。

区分内部数据、外部数据的依据，就是数据的最原始来源，而不是数据在代码中的位置。

比如下面的示意图，标明的就是一些典型的数据检查点。其中小写字母代表数据，大写字母标示的方框代表函数或者方法，数字代表检查点，箭头代表数据流向。



1. 数据 a 是一个外部输入数据，函数 A 使用数据 a 之前，需要校验它的合法性（检查点 1）。
2. 数据 b 是一个外部输入数据，函数 A 使用数据 b 之前，完全校验了它的合法性（检查点 2）。函数 A 内部调用的函数 B 在使用数据 b 时，就不再需要检查它的合法性了。
3. 数据 c 是一个外部输入数据，函数 A 使用数据 c 之前，部分校验了它的合法性（检查点 3）。函数 A 只能使用校验了合法性的部分数据。函数 A 内部调用的函数 B 在使用数据 c 时，如果需要使用未被检验部分的数据，还要检查它的未被校验部分的合法性（检查点 4）。

4. 数据 d 是一个外部输入数据，函数 A 使用数据 d 之前，部分校验了它的合法性（检查点 5）。函数 A 内部调用的函数 B，没有使用该数据，但是把该数据传送给了函数 C。函数 C 在使用数据 d 时，如果需要使用未被检验部分的数据，还要检查它的未被校验部分的合法性（检查点 6）。
5. 数据 e 和 f 是一个内部数据，函数 C 使用内部数据时，不需要校验它的合法性。
6. 数据 g 是一个内部数据，由函数 A 产生，并且输出到外部。这时候，不需要检验数据 g 的合法性，但是需要防护输出数据的变化对内部函数 A 状态的影响（防护点 7）。

原则上，对于外部输入数据的合法性，我们要尽早校验，尽量全面校验。但是有时候，只有把数据分解到一定程度之后，我们才有可能完成对数据的全面校验，这时候就比较容易造成数据校验遗漏。

我们上面讨论过的心脏滴血漏洞，就有点像数据 d 的用例，调用关系多了几层，数据校验的遗漏就难以察觉了。

哪些是外部数据？

你是不是还有一个疑问：为什么数据 e 和 f 对函数 C 来说，就不算是外部数据了？它们明明是函数 C 的外部输入数据呀！

当我们说跨界的数据时，这些数据指的是一个系统边界外部产生的数据。如果我们把函数 A、函数 B 和函数 C 看成一个系统，那么数据 e 和数据 f 就是这个系统边界内部产生的数据。内部产生的数据，一般是合法的，要不然就存在代码的逻辑错误；内部产生的数据，一般也是安全的，不会故意嵌入攻击性逻辑。所以，为了编码和运行的效率，我们一般会选择信任内部产生的数据。

一般的编码环境下，我们需要考量四类外部数据：

1. 用户输入数据（配置信息、命令行输入，用户界面输入等）；
2. I/O 输入数据（TCP/UDP 连接，文件 I/O）；
3. 公开接口输入数据；

4. 公开接口输出数据。


我想，前三类外部数据都容易理解。第四类公开接口输出数据，不是内部数据吗？怎么变成需要考量的外部数据了？我们在 [前面的章节](#) 讨论过这个问题。

公开接口的输出数据，其实是把内部数据外部化了。如果输出数据是共享的可变量（比如没有深拷贝的集合和数组），那么外部的代码就可以通过修改输出数据，进而影响原接口的行为。这也算是一种意料之外的“输入”。

需要注意的是，公开接口的规范，要标明可变量的处理方式。要不然，调用者就不清楚可不可以修改可变量。

让调用者猜测公开接口的行为，会埋下兼容性的祸根。

比如下面的例子，就是两个 Java 核心类库的公开方法。这两个方法，对于传入、传出的可变量（数组）都做了拷贝，并且在接口规范里声明了变量拷贝。

 复制代码

```
1 package javax.net.ssl;
2
3 // snipped
4 public class SSLParameters {
5     private String[] applicationProtocols = new String[0];
6
7     // snipped
8     /**
9      * Returns a prioritized array of application-layer protocol names
10     * that can be negotiated over the SSL/TLS/DTLS protocols.
11     * <snipped>
12     * This method will return a new array each time it is invoked.
13     *
14     * @return a non-null, possibly zero-length array of application
15     *         protocol {@code String}s. The array is ordered based
16     *         on protocol preference, with {@code protocols[0]}
17     *         being the most preferred.
18     * @see #setApplicationProtocols
19     * @since 9
20     */
21     public String[] getApplicationProtocols() {
```

```

22         return applicationProtocols.clone();
23     }
24
25     /**
26      * Sets the prioritized array of application-layer protocol names
27      * that can be negotiated over the SSL/TLS/DTLS protocols.
28      * <snipped>
29      * @implSpec
30      * This method will make a copy of the {@code protocols} array.
31      * <snipped>
32      * @see #getApplicationProtocols
33      * @since 9
34      */
35     public void setApplicationProtocols(String[] protocols) {
36         if (protocols == null) {
37             throw new IllegalArgumentException("protocols was null");
38         }
39
40         String[] tempProtocols = protocols.clone();
41         for (String p : tempProtocols) {
42             if (p == null || p.isEmpty()) {
43                 throw new IllegalArgumentException(
44                     "An element of protocols was null/empty");
45             }
46         }
47
48         applicationProtocols = tempProtocols;
49     }
50 }

```

从上面的例子中，我们也可以体会到，公开接口的编码要比内部接口的编码复杂得多。因为我们无法预料接口的使用者会怎么创造性地使用这些接口。公开接口的实现一般要慎重地考虑安全防护措施，这让公开接口的设计、规范和实现都变得很复杂。从这个意义上来说，我们也需要遵守在第二部分“经济的代码”里谈到的原则：[🔗 接口要简单直观](#)。

小结

通过对这个案例的讨论，我想和你分享下面两点个人看法。


1. **外部输入数据，需要检查数据的合法性；**
2. **公开接口的输入和输出数据，还要考虑可变量的传递带来的危害。**

一起来动手

外部数据的合法性问题，是信息安全里的一大类问题，也是安全攻击者经常利用的一类安全漏洞。

区分内部数据、外部数据的依据，是数据的最原始来源，而不是数据在代码中的位置。这一点让外部数据的识别变得有点艰难，特别是代码层数比较多的时候，我们可能没有办法识别一个传入参数，到底是内部数据还是外部数据。在这种情况下，我们需要采取比较保守的姿态，**无法识别来源的数据，不应该是可信任的数据。**

这一次的练习题，我们按照保守的姿态，来分析下面这段代码中的数据可信任性问题。

 复制代码

```
1 import java.util.HashMap;
2 import java.util.Map;
3
4 public class Solution {
5     /**
6      * Given an array of integers, return indices of the two numbers
7      * such that they add up to a specific target.
8      */
9     public int[] twoSum(int[] nums, int target) {
10         Map<Integer, Integer> map = new HashMap<>();
11         for (int i = 0; i < nums.length; i++) {
12             int complement = target - nums[i];
13             if (map.containsKey(complement)) {
14                 return new int[] { map.get(complement), i };
15             }
16             map.put(nums[i], i);
17         }
18         throw new IllegalArgumentException("No two sum solution");
19     }
20 }
```

欢迎你把你的看法写在留言区，我们一起来学习、思考、精进！

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。

代码精进之路

你写的每一行代码都是你的名片

范学雷

Oracle 首席软件工程师
Java SE 安全组成员
OpenJDK 评审成员



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (8)



彩色的沙漠

2019-06-04

malloc 开辟的空间可能包含释放了内存空间但是没有清除里面数据，所以里面的敏感信息有泄露的风险。释放内存空间的同时清空内存数据也是一个好习惯，避免很多不必要的麻烦

作者回复: 是的，一个完整的方案要做到两个方面：清空敏感数据以及防范内存泄漏。



8



天佑

2019-04-14

老师，文件IO怎么也算外部数据呢？

作者回复: 文件修改不在代码可以控制的范围里呀。



2



2022-01-26

公开接口输出的数据，有个典型的例子，`java.enum.values()`

// 为了避免输出的values array 被无意修改，jdk 的实现是每次都拷贝一个 array
// 虽然性能上有点妥协，但最大程度上保证了代码的正确性



1



DasonCheng

2019-06-13

作为开发人员，有哪些途径即时获取各种漏洞信息以便修补漏洞，避免损失呢？

作者回复：主要有：订阅相关的安全威胁情报，查看依赖软件的安全修复发布版，跟踪NIST提供的全漏洞数据库。



1



空知

2019-04-01

之前的整数溢出也算是边界问题的一种吧

作者回复：是的。



1



天佑

2019-03-29

我看防御式编程会在边界处，专门构建一些类进行外部输入过滤，穿越进边界内不，可以完全信任，这在实际场景当中可操作性更强些吧，避免个人开发的遗漏。

另外，我看到有些例子对外部输入有标准化归一化处理，比如String normalized = Normalizer.normalize(xxx, Normalizer.Form.NFKC);道理也很好理解，觉得这样做会更好些，但是我咨询了开发，他们并不会经常用到，这是为什么，还是有特定场景才会使用？还有nfdc这玩意儿我一直没参透明白，希望老师解惑，谢谢。

作者回复：专门有一个过滤层，这种办法也能有作用，但是局限性很大。个例还可以，不是一个普遍的解决方案。因为，在边界处，如果处理了所有的数据，过滤层就和内部的代码没什么重大区别；如果处理不了所有的数据，遗漏的数据还是不可信任。就像我们文中边界那一部分的数据d一样。另外，加一层做所有的过滤损害效率，增大代码复杂度，破坏代码逻辑。我很少看到这种用法。如果过滤层能想到检查，没有过滤层，常规代码里也能做到。先想到，才能做到；想到了，怎么做就有很多

选择了。

我也不懂NFKC是什么。



1



ifelse 

2022-07-31 来自浙江

无法识别来源的数据，不应该是可信任的数据。--记下来



丁丁历险记

2019-10-15

涨了不少见识

