40 | 规范，代码长治久安的基础

2019-04-05 范学雷

如果从安全角度去考察，软件是非常脆弱的。今天还是安全的代码，明天可能就有人发现漏洞。安全攻击的问题，大部分出自信息的不对称性；而维护代码安全之所以难，大部分是因为安全问题是不可预见的。那么，该怎么保持代码的长治久安呢？

评审案例

有些函数或者接口，可能在我们刚开始写程序的时候，就已经接触，了解，甚至熟知了它们，比如说C语言的read()函数、Java语言的InputStream.read()方法。我一点都不怀疑，我们熟知这些函数或接口的规范。比如说，C语言的read()函数在什么情况下返回值为0？ InputStream.read()方法在什么情况下返回值为-1？

我知道，我们用错read()的概率很小。但是今天，我要和你讨论一两个不太常见，且非常有趣，的错误的用法。

让我们一起来看几段节选改编的C代码，代码中的socket表示网络连接的套接字文件描述符（file descriptor）。 你能够找到这些代码里潜在的问题吗？

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>

void clientHello(int socket) {
    char buffer[1024];
    char* hello = "Hello from client!";

    send(socket, hello, strlen(hello), 0);
    printf("Hello message sent\n");

    int n = read(socket, buffer, 1024);
    printf("%s\n", buffer);
}
```

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>

void serverHello(int socket) {
    char buffer[1024];
    char* hello = "Hello from server!";

    int n = read(socket, buffer, 1024);
    printf("%s\n", buffer);

    send(socket, hello, strlen(hello), 0);
    printf("Hello message sent\n");
}
```

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>

void serverHello(int socket) {
    char buffer[1024];
    char* hello = "Hello from server!";

    int n = read(socket, buffer, 1024);
    if (n == 0) {
        close(socket);
    } else {
        printf("%s\n", buffer);

        send(socket, hello, strlen(hello), 0);
        printf("Hello message sent\n");
    }
}
```

现在，我们集中寻找read()函数返回值的使用问题。为了方便你分析，我把一个标准的read()函数返回值的规范摘抄如下：

> RETURN VALUES
> If successful, the number of bytes actually read is returned. Upon reading end-of-file, zero is returned. Otherwise, a -1 is returned and the global variable errno is set to indicate the error.

上面三段代码里，read()函数的返回值使用都有什么问题？ 上面的函数能够实现编码者所期望的功能吗？

案例分析

上述代码可以作为教学示范的一部分，它们简洁地展示了套接字文件描述符的一些使用方法。但是，这些代码离真正的工业级产品的质量要求还有很大的一段距离。当然了，如果你把上述的代码运行一万次，那么这一万次可能都不会辜负你的期望；运行一百万次，一百万次也可能都是成功的。但是，不论是理论上还是实际上，这些代码还是有可能出现错误的，它们并不是可靠的代

码。

问题出在哪儿呢？如果我们仔细阅读read()函数返回值规范，可以注意到，read()函数的返回值是实际读取的字节数。一段信息，套接字的底层实现可能会分段传输，分段接收。所以，read()函数并不能保证一次调用就返回完整的一段信息，传送和接收也未必是一一对应的，即使这段信息很短。

在上述的例子中，如果期望接收到的信息是“Hello from server!”，那么一次read()函数的执行，实际接收到的信息可能是完整的信息，也可能是一个开头的字母“H”。套接字的底层实现并不能保证通过调用一次或者两次read()函数，就能够接收到这条完整的信息。

这其实带来了一个不小的麻烦。如果调用read()函数的次数无法确定，那么接收端就要一直读取，直到接收到完整的信息。可是，什么样的信息才是完整的信息呢？接收端似乎并没有办法知道一条信息是否完整。

比如在上面的例子中，对于接收端来说，怎么知道“H”不是一条完整的信息，“Hello”也不是一条完整的信息，而“Hello from server!”就是一条完整的信息呢？无法判断信息的完整性，就会面临信息丢失或者读取阻塞的问题。所以，应用层面的设计，必须考虑如何检验接收消息是否完整。比如，对于HTTP协议而言，请求行必须以“CRLF”结束。那么，接收端读取到“CRLF”，就能够确定请求行的数据传输完整了。

在实际运行中，如果信息足够短，比如上面的“Hello from server!”，那么套接字底层的实现和网络环境，大部分情况下都能够一次传输完整的信息。所以，上述代码运行一万次，可能这一万次都是成功的。即便如此，也不能保证每次传输的都是完整的信息。

这里面还有另一个不太小的麻烦，是关于read()函数的实现的。函数的规范要求，数据传输结束（End-Of-File）后，read()函数应该返回0。那么，read()函数返回0，是不是就表示数据传输结束呢？ 是的。不然的话，应用程序如何判断数据传输结束又是一个大麻烦。

可是，的确存在类似的实现，读取操作返回了0，但是数据传输才刚刚开始。下面我要给你讲的这个例子，**就是这样的一个看似微不足道，但后果却很严重的问题，把互联网协议的重要安全变更，耽搁了整整十年**。

十年的死局

安全套接字协议（ Secure Socket Layer, 简称SSL）是用来确保网络通信和事务安全的最常见的标准。现在只要你使用互联网，几乎就是这个标准的使用者。这个标准最初由网景公司（NetScape）设计并且实现，后来移交给了国际互联网工程任务组（The Internet Engineering Task Force，简称 IETF）管理，并且更名为**传输层安全协议**（Trassport Layer Security，简称TLS）。

我们通过浏览器输入，并且传输到网站的用户名和密码必须只有我们自己知道，不能在传输的过

程中被第三者窃取，也不能传送给指定网站以外的服务器。一般来说，浏览器和服务器之间需要建立安全传输连接。这样，网站的真实性是经过校验的，浏览器和网站之间传输的所有数据都是经过加密的，只有我们自己和网站服务器可以解密、理解传输的数据。

传输层安全协议就是用来满足这些安全需求的。那它是怎么做到呢？传输层安全协议需要使用一系列的密码技术，来保证安全连接的建立。

保证数据的私密性使用的是数据加密技术。其中，影响最大的一类数据加密技术使用的是一种叫作链式加密(Cipher Block Chaining，CBC)的模式。简单地说，就是前一个加密数据的最后一个数据块，被用来作为后一个数据块加密的输入参数。这样，就形成了后一个加密数据依赖前一个加密数据的链条。

1999年1月，传输层安全协议第一版发布，一般简称为TLS 1.0。TLS 1.0使用链式加密模式作为其加密传输数据的一个技术方案。TLS 1.0获得了巨大的成功。我们很难想象如果没有TLS协议，互联网会是一个什么样子。然而，完美的东西，渴求不来也偶遇不到。

2001年9月的密码学进展大会上，一位密码学研究者（Hugo Krawczyk）发表了一篇论文，该论文研究了链式加密的缺陷，以及对于TLS协议的影响。利用链式加密的缺陷，攻击者可以破解出加密密码，使用这个密码，就可以解密加密的传输数据，从而获取传输信息。从此，链式加密，一个有着最广泛影响的技术，开始淡出历史舞台。然而，这个进程非常缓慢，非常缓慢。在新技术替代的过程中，老技术的现有问题，以及新老技术的衔接，会出现很多非常复杂和棘手的问题。原有的技术使用得越多，部署得越广泛，这些问题越复杂。

2002年，OpenSSL，一个被广泛使用的实现传输层安全协议的类库，发布了针对链式加密缺陷的安全补丁和缺陷报告。这个解决方案的目的就是打破链式加密模式的链条，在数据块之间插入随机数据。由于随机数据插到了加密数据链之间，解决了链式加密模式的上述缺陷，这使得链式加密的形式和算法得以保留。

幸运的是，TLS协议的设计恰巧允许这种使用方法，那么TLS协议在理论上仍可以继续使用。既然是随机数据，那就是没有任何意义的数据，不能用于实际的应用，接收端必须忽略这些随机数据。TLS协议通过传输一个空数据段，然后再传输有效数据，就可以达到添加随机数据的目的。在理论上，这是一个很好的解决方案。然而，现实比想象的还要精彩。

该解决方案的真正落地，需要read()函数或者类似的方法有一个好的实现。在接收到空数据段所代表的随机数据时，需要忽略该数据段，继续等待真正有效的数据，不能返回0。为什么不能返回0呢？还记得上面的read()返回值规范吗？返回值为0，代表数据传输结束，应用程序就不应该继续使用该通道了，后续的数据都会被丢弃。可是对于这个解决方案，如果read()返回0，意味着真正的数据传输才刚刚开始，而不是结束。

如果这样的实现存在，那么这个解决方案不但没有解决安全缺陷问题，还直接导致应用程序不能继续使用。

有没有这样一时糊涂的实现呢？ OpenSSL的缺陷报告里，提到了一个这样的糊涂的实现。有这么有一个产品，名字的简写是MSIE。曾经，它是一种特立独行般的存在，到了哪里，哪里就会绽放出不一样的烟火。考虑到MSIE及其相关家族产品巨大的市场使用份额，谁采用该安全缺陷修复解决方案，谁就自绝于市场，自绝于广大的用户。遇到了这种巨大的互操作性问题后，OpenSSL随后缺省关闭了这个安全漏洞修补方案。随后，其他公司，比如Google也曾经尝试在他们的产品中做类似的安全修复，都因为这种灾难性的互操作性问题而放弃。安全诚可贵，自由价更高！

对于这样糊涂的实现而言，这只能算是一个芝麻蒜皮的小问题。修复这样的问题也应该不是多么困难的事情。可是，真正的困难在于，这样的产品已经有了非常广泛的用户群体，以及产品部署，包括个人计算机、自动取款机、商超收银机以及银行柜员机等各种各样的形式。

很多产品的部署形式使得产品的升级非常困难，更别提还有很多产品的实现，是以固件的方式存在的了。比如我们家里用的路由器，部署在计算机房里的交换机，以及每辆汽车里的计算机，这些都是升级非常困难的产品。**用户越广泛，部署越广泛，升级就越困难，安全变更面临的挑战就越大。芝麻蒜皮的小问题，都可能构筑困难的障碍，带来巨大的风险，从而造成严重的损失**。

可能你会有疑问，我换一个浏览器不就没事了吗？如果服务器使用的是这样糊涂的实现，那么一个浏览器是没有办法访问这样的服务器提供的服务的。如果这样的服务器被广泛使用，那么一个浏览器的合理策略，就是不开启这种安全缺陷修复。很多网站不能访问的浏览器，是一个不会有人使用的浏览器。

那么，我自己的服务器是不是可以启动这个安全修复呢？问题又回到了客户端，如果客户端使用了这样糊涂的实现，它也没有办法访问修复了的服务器。如果这样的客户端被广泛使用，比如说最流行的浏览器，那么一个服务器的最合理策略就是不开启这种安全修复。假如一个网站有很多用户不能访问，这实在不是网站设计者和拥有者的初衷。

看起来，这似乎是一个死局！

当时的共识是，针对该漏洞的攻击并不会轻易得手，所以即使不修复该漏洞，估计也不是一个多大的问题。同时，针对该漏洞的升级协议也有条不紊地开始了。

2006年，经过4年的反复敲打，传输层安全协议版本1.1发布，一般简称为TLS 1.1。TLS 1.1的一个重要的任务，就是解决链式加密的缺陷。然而，**任何一个标准从制定到落实，都有一段很长的路要走**。TLS 1.1并没有得到业界及时的响应和应有的重视。携带着安全缺陷的TLS 1.0依然统治着传输安全的世界，似乎大家并没有觉得有太多的不妥之处。 时间来到了十年后，2011年9月。

无奈的少数派

针对链式加密安全漏洞的攻击真的不会轻易得手吗？2010年，一个年轻人（Juliano Rizzo），在印度尼西亚的海滩上阅读了OpenSSL的缺陷报告。在优美的印尼海滩上，他发现了一种可能非常有效的攻击方法。

2011年9月，两位天才般的研究人员（Juliano Rizzo，Thai Duong）表示，给他们几分钟时间，他们就可以利用该漏洞入侵你的支付账户。他们给这个攻击技术取了一个超酷的名字，BEAST。你要是搜索一下“the BEAST attack”，就知道这是一个多么轰动的攻击技术。

他们的研究成果受到了密码学家的高度赞美。但是业界厂商的处境就比较尴尬了。毕竟，这是他们十年前尝试修复，但是最后不得不放弃修复的漏洞。十年后的今天，原来阻碍这个漏洞修复的现实障碍，并没有减少。原计划2011年7月份公开发表论文的日期，不得不推迟。 因为直到7月份，还是没有合适的修复方案。这让人感到有些失望，有些沮丧。

7月20日，事情有了转机。

如果传输空数据段不被接受，那么传输一个字节呢？空数据的read()实现可能返回0，一个字节的read()实现应该毫无例外地返回1。在TLS 1.0的链式加密模式下，传输一个字节时，有足够随机的数据插入链式加密数据块之间，简单有效地打破链式加密模式的链条。基于这个想法，7月20日，一个通常被称为1/n-1分割的解决方案被提出，并且得到了验证。

由于该方法简单有效，主流厂商迅速采纳了这个方案，发布了对应产品的安全补丁。幸运的是，TLS 1.0续命了十年，业界有更多的时间完成产品的升级换代。不幸但也在预料之中的是，该方案也不是一点兼容性影响都没有。

比如我们案例中讨论的代码，就出了大问题。预期收到一条完整的信息“Hello from server!”。 使用了这个安全补丁后，就必须要接收被分割的两条信息，“H”以及“ello from server!”。如果应用不能处理分割的信息，就不能好好工作了。

幸运的是，虽然不能处理分割信息的应用依然存在，但是数量很少。而且，这是应用自身的问题，很难抱怨安全补丁的不是。由于主流的厂商拥抱了1/n-1分割法，而存在问题的应用又是少数派，这些少数派不得不亲手解决他们自身的问题。否则就面临着应用不得不停工的损失，或者承受安全攻击的风险。

对于某一个特定的问题来说，一旦我们成为少数派的一部分，就有可能面临软件安全的风险，以及在兼容性方面做妥协。对于接口规范来说，我们应该严格遵从白名单原则，没有明文规定的行为规范，就不是能依赖的行为规范。

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. **对于应用接口（API）的使用，一定要严格遵守规范，小失误可能造成大麻烦**；

2. **对于应用接口（API）的定义，一定要清晰简单，描述一定要详实周到。如果使用者对规范的理解感到困难或者困惑，可能会带来难以预料的问题；**

3. **对于应用接口（API）的实现，一定要在规范许可范围内自由发挥。越是影响广泛的实现，越不要逾越规范的界限。**

这是一个特殊的案例，我们好像聊了一个故事。对这个案例，你还有什么看法吗？

一起来动手

我们讨论了read()函数返回值的问题，可是上述的案例，还有其他的问题存在。你还发现了什么问题？这些问题该怎么更改？你可以使用Java或者你熟悉的语言来修改。这可并不是一个简单的修改，我知道你一定会遇到很多问题，欢迎留言分享你的修改或者问题。

如果让你给clientHello()或者serverHello()加上规范描述，你会怎么描述？你会用什么样的文字，告诉这个接口的使用者，该怎么正确地使用这个应用接口？这同样不是一个简单的小练习，欢迎分享你的规范描述。

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。

精选留言


hua168

0

像我们开发是直接调用框架函数，如果是安全问题，一般是框架自身的问题吧？

2019-04-08

作者回复

一般不是框架自身的问题。

2019-04-09

我来也

0

我还是只懂c语言。我觉得比较奇怪，buffer的长度是1024，read 1024没问题，但是printf（%s）时，如果最后一个字节不是\0，那输出就会有问题了。

2019-04-06

作者回复

哈哈，buffer不是字符串，printf打印的是字符串，所以问题就来了。

2019-04-07