

18 | 防人之心不可无：检查数据的有效性

2019-09-11 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 10:14 大小 9.38M



你好，我是盛延敏，这里是网络编程实战第 18 讲，欢迎回来。

在前面一讲中，我们仔细分析了引起故障的原因，并且已经知道为了应对可能出现的各种故障，必须在程序中做好防御工作。

在这一讲里，我们继续前面的讨论，看一看为了增强程序的健壮性，我们还需要准备什么。

对端的异常状况

在前面的第 11 讲以及第 17 讲中，我们已经初步接触过一些防范对端异常的方法，比如，通过 read 等调用时，可以通过对 EOF 的判断，随时防范对方程序崩溃。

```
1 int nBytes = recv(connfd, buffer, sizeof(buffer), 0);
2 if (nBytes == -1) {
3     error(1, errno, "error read message");
4 } else if (nBytes == 0) {
5     error(1, 0, "client closed \n");
6 }
```

你可以看到这一个程序中的第 4 行，当调用 read 函数返回 0 字节时，实际上就是操作系统内核返回 EOF 的一种反映。如果是服务器端同时处理多个客户端连接，一般这里会调用 shutdown 关闭连接的这一端。

上一讲也讲到了，不是每种情况都可以通过读操作来感知异常，比如，服务器完全崩溃，或者网络中断的情况下，此时，如果是阻塞套接字，会一直阻塞在 read 等调用上，没有办法感知套接字的异常。

其实有几种办法来解决这个问题。

第一个办法是给套接字的 read 操作设置超时，如果超过了一段时间就认为连接已经不存在。具体的代码片段如下：


```
1 struct timeval tv;
2 tv.tv_sec = 5;
3 tv.tv_usec = 0;
4 setsockopt(connfd, SOL_SOCKET, SO_RCVTIMEO, (const char *) &tv, sizeof tv);
5
6 while (1) {
7     int nBytes = recv(connfd, buffer, sizeof(buffer), 0);
8     if (nBytes == -1) {
9         if (errno == EAGAIN || errno == EWOULDBLOCK) {
10             printf("read timeout\n");
11             onClientTimeout(connfd);
12         } else {
13             error(1, errno, "error read message");
14         }
15     } else if (nBytes == 0) {
16         error(1, 0, "client closed \n");
17     }
18     ...
19 }
```

这个代码片段在第 4 行调用 `setsockopt` 函数，设置了套接字的读操作超时，超时时间为在第 1-3 行设置的 5 秒，当然在这里这个时间值是“拍脑袋”设置的，比较科学的设置方法是通过一定的统计之后得到一个比较合理的值。关键之处在读操作返回异常的第 9-11 行，根据出错信息是 `EAGAIN` 或者 `EWOULDBLOCK`，判断出超时，转而调用 `onClientTimeout` 函数来进行处理。

这个处理方式虽然比较简单，却很实用，很多 FTP 服务器端就是这么设计的。连接这种 FTP 服务器之后，如果 FTP 的客户端没有续传的功能，在碰到网络故障或服务器崩溃时就会挂断。

第二个办法是第 12 讲中提到的办法，添加对连接是否正常的检测。如果连接不正常，需要从当前 `read` 阻塞中返回并处理。

还有一个办法，前面第 12 讲也提到过，那就是利用多路复用技术自带的超时能力，来完成对套接字 I/O 的检查，如果超过了预设的时间，就进入异常处理。

 复制代码

```
1 struct timeval tv;
2 tv.tv_sec = 5;
3 tv.tv_usec = 0;
4
5 FD_ZERO(&allreads);
6 FD_SET(socket_fd, &allreads);
7 for (;;) {
8     readmask = allreads;
9     int rc = select(socket_fd + 1, &readmask, NULL, NULL, &tv);
10    if (rc < 0) {
11        error(1, errno, "select failed");
12    }
13    if (rc == 0) {
14        printf("read timeout\n");
15        onClientTimeout(socket_fd);
16    }
17    ...
18 }
```

这段代码使用了 `select` 多路复用技术来对套接字进行 I/O 事件的轮询，程序的 13 行是到达超时后的处理逻辑，调用 `onClientTimeout` 函数来进行超时后的处理。

缓冲区处理

一个设计良好的网络程序，应该可以在随机输入的情况下表现稳定。不仅是这样，随着互联网的发展，网络安全也愈发重要，我们编写的网络程序能不能在黑客的刻意攻击之下表现稳定，也是一个重要考量因素。


很多黑客程序，会针对性地构建出一定格式的网络协议包，导致网络程序产生诸如缓冲区溢出、指针异常的后果，影响程序的服务能力，严重的甚至可以夺取服务器端的控制权，随心所欲地进行破坏活动，比如著名的 SQL 注入，就是通过针对性地构造出 SQL 语句，完成对数据库敏感信息的窃取。

所以，在网络程序的编写过程中，我们需要时时刻刻提醒自己面对的是各种复杂异常的场景，甚至是别有用心的攻击者，保持“防人之心不可无”的警惕。

那么程序都有可能出现哪几种漏洞呢？

第一个例子


我在文稿中已经放置了一段代码：

 复制代码

```
1 char Response[] = "COMMAND OK";
2 char buffer[128];
3
4 while (1) {
5     int nBytes = recv(connfd, buffer, sizeof(buffer), 0);
6     if (nBytes == -1) {
7         error(1, errno, "error read message");
8     } else if (nBytes == 0) {
9         error(1, 0, "client closed \n");
10    }
11
12    buffer[nBytes] = '\0';
13    if (strcmp(buffer, "quit") == 0) {
14        printf("client quit\n");
15        send(socket, Response, sizeof(Response), 0);
16    }
17
18    printf("received %d bytes: %s\n", nBytes, buffer);
19 }
```

这段代码从连接套接字中获取字节流，并且判断了出差和 EOF 情况，如果对端发送来的字符是 “quit” 就回应 “COMAAND OK” 的字符流，乍看上去一切正常。

但仔细看一下，这段代码很有可能会产生下面的结果。


 复制代码

```
1 char buffer[128];  
2 buffer[128] = '\0';
```

通过 `recv` 读取的字符数为 128 时，就会是文稿中的结果。因为 `buffer` 的大小只有 128 字节，最后的赋值环节，产生了缓冲区溢出的问题。

所谓缓冲区溢出，是指计算机程序中出现的一种内存违规操作。本质是计算机程序向缓冲区填充的数据，超出了原本缓冲区设置的大小限制，导致了数据覆盖了内存栈空间的其他合法数据。这种覆盖破坏了原来程序的完整性，使用过游戏修改器的同学肯定知道，如果不小心修改错游戏数据的内存空间，很可能导致应用程序产生如 “Access violation” 的错误，导致应用程序崩溃。

我们可以对这个程序稍加修改，我把代码贴在了文稿里，主要的想法是留下 `buffer` 里的一个字节，以容纳后面的 `'\0'`。

 复制代码

```
1 int nBytes = recv(connfd, buffer, sizeof(buffer)-1, 0);
```

这个例子里面，还昭示了一个有趣的现象。你会发现我们发送过去的字符串，调用的是 `sizeof`，那也就意味着，`Response` 字符串中的 `'\0'` 是被发送出去的，而我们在接收字符时，则假设没有 `'\0'` 字符的存在。

为了统一，我们可以改成如下的方式，使用 `strlen` 的方式忽略最后一个 `'\0'` 字符。


 复制代码

```
1 send(socket, Response, strlen(Response), 0);
```

第二个例子

第 16 讲中提到了对变长报文解析的两种手段，一个是使用特殊的边界符号，例如 HTTP 使用的回车换行符；另一个是将报文信息的长度编码进入消息。

在实战中，我们也需要对这部分报文长度保持警惕。

 复制代码

```
1 size_t read_message(int fd, char *buffer, size_t length) {
2     u_int32_t msg_length;
3     u_int32_t msg_type;
4     int rc;
5
6     rc = readn(fd, (char *) &msg_length, sizeof(u_int32_t));
7     if (rc != sizeof(u_int32_t))
8         return rc < 0 ? -1 : 0;
9     msg_length = ntohl(msg_length);
10
11    rc = readn(fd, (char *) &msg_type, sizeof(msg_type));
12    if (rc != sizeof(u_int32_t))
13        return rc < 0 ? -1 : 0;
14
15    if (msg_length > length) {
16        return -1;
17    }
18
19    /* Retrieve the record itself */
20    rc = readn(fd, buffer, msg_length);
21    if (rc != msg_length)
22        return rc < 0 ? -1 : 0;
23    return rc;
24 }
```

在进行报文解析时，第 15 行对实际的报文长度`msg_length`和应用程序分配的缓冲区大小进行了比较，如果报文长度过大，导致缓冲区容纳不下，直接返回 -1 表示出错。千万不要小看这部分的判断，试想如果没有这个判断，对方程序发送出来的消息体，可能构建出一个非常大的`msg_length`，而实际发送的报文本体长度却没有这么大，这样后面的读取操作就不会成功，如果应用程序实际缓冲区大小比`msg_length`小，也产生了缓冲区溢出的问题。

```

1 struct {
2     u_int32_t message_length;
3     u_int32_t message_type;
4     char data[128];
5 } message;
6
7 int n = 65535;
8 message.message_length = htonl(n);
9 message.message_type = 1;
10 char buf[128] = "just for fun\0";
11 strncpy(message.data, buf, strlen(buf));
12 if (send(socket_fd, (char *) &message,
13         sizeof(message.message_length) + sizeof(message.message_type) + strlen(message
14         error(1, errno, "send failure");

```

文稿里就是这样一段发送端“不小心”构造的一个程序，消息的长度“不小心”被设置为 65535 长度，实际发送的报文数据为“just for fun”。在去掉实际的报文长度 `msg_length` 和应用程序分配的缓冲区大小做比较之后，服务器端一直阻塞在 `read` 调用上，这是因为服务器端误认为需要接收 65535 大小的字节。

第三个例子

如果我们需要开发一个函数，这个函数假设报文的分界符是换行符（`\n`），一个简单的想法是每次读取一个字符，判断这个字符是不是换行符。

文稿中给出了这样的函数，这个函数的最大问题是工作效率太低，要知道每次调用 `recv` 函数都是一次系统调用，需要从用户空间切换到内核空间，上下文切换的开销对于高性能来说最好是能省则省。

```


1 size_t readline(int fd, char *buffer, size_t length) {
2     char *buf_first = buffer;
3
4     char c;
5     while (length > 0 && recv(fd, &c, 1, 0) == 1) {
6         *buffer++ = c;
7         length--;
8         if (c == '\n') {
9             *buffer = '\0';
10            return buffer - buf_first;
11        }

```



```
12     }
13
14     return -1;
15 }
```

于是，就有了文稿中的第二个版本，这个函数一次性读取最多 512 字节到临时缓冲区，之后将临时缓冲区的字符一个一个拷贝到应用程序最终的缓冲区中，这样的做法明显效率会高很多。


 复制代码

```
1 size_t readline(int fd, char *buffer, size_t length) {
2     char *buf_first = buffer;
3     static char *buffer_pointer;
4     int nleft = 0;
5     static char read_buffer[512];
6     char c;
7
8     while (length-- > 0) {
9         if (nleft <= 0) {
10             int nread = recv(fd, read_buffer, sizeof(read_buffer), 0);
11             if (nread < 0) {
12                 if (errno == EINTR) {
13                     length++;
14                     continue;
15                 }
16                 return -1;
17             }
18             if (nread == 0)
19                 return 0;
20             buffer_pointer = read_buffer;
21             nleft = nread;
22         }
23         c = *buffer_pointer++;
24         *buffer++ = c;
25         nleft--;
26         if (c == '\n') {
27             *buffer = '\0';
28             return buffer - buf_first;
29         }
30     }
31     return -1;
32 }
```


这个程序的主循环在第 8 行，通过对 length 变量的判断，试图解决缓冲区长度溢出问题；第 9 行是判断临时缓冲区的字符有没有被全部拷贝完，如果被全部拷贝完，就会再次尝试读取最多 512 字节；第 20-21 行在读取字符成功之后，重置了临时缓冲区读指针、临时缓冲区待读的字符个数；第 23-25 行则是在拷贝临时缓冲区字符，每次拷贝一个字符，并移动临时缓冲区读指针，对临时缓冲区待读的字符个数进行减 1 操作。在程序的 26-28 行，判断是否读到换行符，如果读到则将应用程序最终缓冲区截断，返回最终读取的字符个数。

这个程序运行起来可能很久都没有问题，但是，它还是有一个微小的瑕疵，这个瑕疵很可能会造成线上故障。


为了讲清这个故障，我们假设这样调用，输入的字符为 012345678\n。

 复制代码

```
1 // 输入字符为：012345678\n
2 char buf[10]
3 readline(fd, buf, 10)
```

当读到最后一个\n 字符时，length 为 1，问题是在第 26 行和 27 行，如果读到了换行符，就会增加一个字符串截止符，这显然越过了应用程序缓冲区的大小。

正确的程序我也附在了文稿中，这里最关键的是需要先对 length 进行处理，再去判断 length 的大小是否可以容纳下字符。

 复制代码

```
1 size_t readline(int fd, char *buffer, size_t length) {
2     char *buf_first = buffer;
3     static char *buffer_pointer;
4     int nleft = 0;
5     static char read_buffer[512];
6     char c;
7
8     while (--length > 0) {
9         if (nleft <= 0) {
10             int nread = recv(fd, read_buffer, sizeof(read_buffer), 0);
11             if (nread < 0) {
12                 if (errno == EINTR) {
13                     length++;
14                     continue;
15                 }
16             }
17         }
18     }
19 }
```

```
16         return -1;
17     }
18     if (nread == 0)
19         return 0;
20     buffer_pointer = read_buffer;
21     nleft = nread;
22 }
23 c = *buffer_pointer++;
24 *buffer++ = c;
25 nleft--;
26 if (c == '\n') {
27     *buffer = '\0';
28     return buffer - buf_first;
29 }
30 }
31 return -1;
32 }
```

总结

今天的内容到这里就结束了。让我们总结一下：在网络编程中，是否做好了对各种异常边界的检测，将决定我们的程序在恶劣情况下的稳定性，所以，我们一定要时刻提醒自己做好应对各种复杂情况的准备，这里的异常情况包括缓冲区溢出、指针错误、连接超时检测等。

思考题

和往常一样，给大家留两道思考题吧。

第一道，我们在读数据的时候，一般都需要给应用程序最终缓冲区分配大小，这个大小有什么讲究吗？

第二道，你能分析一下，我们文章中的例子所分配的缓冲是否可以换成动态分配吗？比如调用 `malloc` 函数来分配缓冲区？

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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。

上一篇 17 | TCP并不总是“可靠”的？

下一篇 19 | 提高篇答疑：如何理解TCP四次挥手？

精选留言 (7)

写留言



yusuf

2019-09-11

- 1、大小一般为2的多少次方
- 2、不能换成动态分配。在read中需要sizeof指明接受数据的最大长度，malloc返回的是一个指针，求指针的sizeof时返回的是指针所占内存大小（32位为4，64位为8），跟实际数据的大小不一致

展开 ∨

3

2



传说中的成大大

2019-09-15

第一问：

不能太小也不能太大 太小了频繁的用户态和内核态切换,太大了读不够容易阻塞,就算不阻塞

也容易浪费

第二问:

如果用malloc频繁的申请和释放也不太好 容易造成碎片

展开 ▾



一周思进

2019-09-13

判断是否换行也可以直接strstr判断吧?

<https://mp.weixin.qq.com/s/YvfZMO2gCjHWmrNRGpdibA>

我觉得这两种方式的问题就是把后面读取的数据丢弃了, 这对于tcp通信可能存在问题吧?
在想后面是不是得换成全局循环缓冲区读写?

展开 ▾



LDxy

2019-09-12

1, 最终缓冲区的大小应该比预计接收的数据大小大一些, 预防缓冲区溢出。2, 完全可以动态分配, 但是要记得在return前释放缓冲区

展开 ▾



徐凯

2019-09-12

第一题。是不是跟结构体字节对齐一样的意思。数据比如是2的倍数 可以方便cpu处理?

第二题 可以是动态内存 有时候应用层分包可以自定义几m一个包 甚至更大 而栈上分配空间是有限的 平均都在2到4m的样子 如果在栈上分配缓冲区 可能你的程序会根据平台不同选择性崩溃, 而在堆上则没有这个问题唯一需要注意的是内存泄漏问题 c++有智能指针可以避免, java应该更方便吧 它的内存回收不是很厉害的嘛

展开 ▾



刘晓林

2019-09-12

老师, 第二个例子中, 及时加上了msg_length和缓冲区length的大小比较, 如果msg_length写得很大(但小于length)而实际数据没有那么大时, 服务器也会阻塞在read上吧?
所以说判断msg_length<=length并不能接read阻塞的问题呀, 只能解内存溢出的问题。

展开 ▾





Steiner

2019-09-11

我觉得不能用动态分配，如果程序崩溃了，内存还没回收会内存溢出吧



4

