TCP 选择性应答的性能权衡

SACK 优化会引起拒绝服务攻击吗？

在最近几个月 Internet 上对 Linux® 开发的日常讨论中，一个重要的话题就是 Linux 的 TCP SACK（Selective Acknowledgment）实现。讨论主要集中在处理某些 SACK 事件时 TCP 协议栈的性能方面，有一些人认为这会带来安全隐患。

我对这些讨论非常感兴趣，但是这些讨论似乎缺乏有力的数据支持。他们讨论的是哪些特定情形？这只是一个小的性能缺陷，还是完全有可能使服务器受到拒绝服务攻击？

我收集了一些有关此主题的引述（参见 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-tcp-sack/#artrelatedtopics) 中这些引述的链接）：

David Miller：“现在，基本上每个 TCP 协议栈都存在此问题，处理无效或恶意创建的 SACK 块会消耗大量 CPU 资源。”

Ilpo Jarvinen [1]：“但是，只要在 sacktag 中存在对 skb 的 fack\_count 的依赖，即使在 RB-tree 的情况下，CPU 也存在处理攻击的空间，因为在慢速道上只能步行”。

NC State University：“我们在这个试验中展示了 SACK 处理的效率。正如我们看到的许多情形，TCP 不能很好地处理大型窗口拖放，尤其是大量数据的缓冲。”

CHC IT: “最后，对 2.4 和 2.6 版提出一个警告：对于 TCP 窗口大于 20 MB 的大型 BDP 路径，可能会遇到 Linux SACK 实现问题。如果 Linux 收到一个 SACK 事件时有大量包在传递，它会花费很长的时间定位经过 SACK 处理的包，而且将会导致 TCP 超时，CWND 将返回到第一个包。”

本文将观察 SACK 实现及其在非理想条件下的性能，我们将从 Linux 2.6.22 入手，它目前是 Ubuntu 7.10 的普通发行版内核。该内核在几个月之前发行，自从它发布以来没有出现什么问题，开发人员一直在为其编写代码。当前的开发内核为 2.6.25，其中包含 Ilpo Jarvinen 提供的一系列补丁，用于处理 SACK 性能。本文最后将查看这些代码如何发挥作用，并简略讨论进一步的更改。

SACK 简介

SACK 由 RFC 2018、2883 和 3517 定义（参见 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-tcp-sack/#artrelatedtopics) 中这些 RFC 的链接）。普通 TCP（即未提供 SACK 特性）应答是严格累积的 — 对 N 的应答意味着字节 N 和所有之前的字节都已经收到。SACK 要解决的问题普通累积式应答的 “全有或全无” 性质。

例如，即使包 2（假设从 0 到 9 的序列）是在传送过程中惟一丢失的包，接收方也只能对包 1 发出一个普通的 ACK，因为这是连续接收到的包中的最后一个。另一方面，SACK 接收方可以发出包 1 的 ACK 和包 3 到包 9 的 SACK 选项。这种附加信息可以帮助发送方确定丢失的包最少，只需重新传送很少的数据。如果没有这种附加信息，它需要重新传送大量的数据，这样会降低传送速率，从而适应高丢包率的网络。

在高延迟的连接中，SACK 对于有效利用所有可用带宽尤其重要。高延迟会导致在任何给定时刻都有大量正在传送的包在等待应答。在 Linux 中，除非得到应答或不再需要，这些包将一直存放在重传队列中。这些包按照序列编号排队，但不存在任何形式的索引。当需要处理一个收到的 SACK 选项时，TCP 协议栈必须在重传队列中找到应用了 SACK 的包。重传队列越长，找到所需的数据就越困难。

每个包中的最多包含 4 个 SACK 选项。

攻击场景

常见的安全漏洞源于这样一个事实：SACK 选项的接收方在收到一个包后可能会被要求做大量工作。这种 N:1 的比例使 SACK 发送方可以打击非常强大的平台上的接收方。

特定于 Linux SACK 处理器的攻击场景要求重传队列中已经持有大量包。然后攻击者发送一个充满 SACK 选项的包，目的是使另一方主机扫描整个队列以处理每个选项。一个指向队列末尾包的选项可能会迫使接收方的 TCP 协议栈遍历整个队列以判断此选项指向哪一个包。显然，攻击客户机可以根据自己的喜好发送任何 SACK 选项，但不太容易看出的一个问题是，它可以轻易控制被攻击方的重传队列的长度。SACK 选项的创建者可以控制接收者对接收到的每个选项执行的工作量，因为它也可以控制队列的大小。

重传队列中包的数目基本上取决于两台主机间的带宽延迟效果（bandwidth delay product，BDP）。一条路径的带宽受到网络的物理属性的限制 — 攻击者在未获得更多基础设施的情况下很难增加带宽。可是，攻击者通过在传输之前使每个应答阻塞一小段时间就可任意增加延迟。为了有效利用高延迟连接的带宽，服务器需要保持大量的包处于传输途中，以便总传输时间等于它得到延迟应答的时间。如果不这样做，则网络在某段时间内不会传输任何包，因此它的带宽未得到充分利用。高延迟路径需要大量在传输中的包以便充分使用网络，而 TCP 发送方会在拥塞控制标准的限制范围内尽可能充分利用带宽。

延迟应答大大增加了重传队列的长度，这为攻击提供了必要条件。例如，如果在相对慢的每秒 10MB 的连接中增加 1750ms 的延迟，在这个期限内会产生超过 12000 个包。稍快的连接会产生更多的包，但这种漏洞部分源于这样一个事实，因为引入了较高的延迟，这种方法只适用于标准家庭宽带连接。

在传输延迟的应答时客户机选择 SACK 选项的值。发送方增加一个指向最新收到的包数据的 SACK 选项（也就是拥有最大序列号的数据）。在这种场景中，此数据开始延迟自己的 ACK，但现在它可以选择性应答，这会造成在已经应答和选择性应答的包之间的距离在重传队列中是最长的。

这种特殊的攻击场景（本文的焦点）被称为 *find-first* 攻击，因为它迫使 TCP 协议栈花费大量时间查找由 SACK 选项指向的第一个字节。

攻击场景评测

Kode Vicious：“采用特殊评测！”。

了解这些背景知识后，让我们来判断这是否是一个严重的问题。这是一个需要细微优化的领域还是一个严重的危机，或者介于二者之间？为弄清这一点，我们需要用真实的数据描述问题的严重程度。第一项任务就是确定要采集哪些数据以及如果进行评测。

实验设置

需要收集的最重要的数据项是服务器的 CPU 使用率。Oprofile 可通过标准的 CLK\_UNHALTED 计数器完成这个棘手的工作。另一感兴趣的数据是在处理 SACK 时扫描的包的数目，以及重传窗口（window）的平均大小。我测试了服务器的源代码以获得扫描包的计数器。在没有注释的情况下我又重新进行了测试以确保我得到的结果与标准服务器看到的结果相同。

传输中的窗口大小也是一个感兴趣的数据。如果在客户机可以跟踪发出的 SACK 选项的数目，则重传队列的大小可从扫描包计数器中计算得到。对于非 SACK 测试用例，我使用客户机报告的延迟应答队列的典型长度作为发送方窗口大小的近似值。

全部评测在标准 Linux 服务器上完成。我编写了定制客户机，用于驱动实验并触发服务器上感兴趣的代码路径。客户机实现自己的 TCP 栈并在原始 Socket 内核 API 上运行。客户机当然算不上完整的 TCP 协议栈，但这对测量感兴趣的各种变量而言已经足够了。

客户机启动实验，首先与服务器连接并发送一个简单的 HTTP 请求，请求目的是获取一个 700MB 的 ISO 文件。然后它消耗由服务器通过普通的 100Mbit/s 的网络发送来的全部数据。服务器发来的每个包在经过 1750ms 的延迟后得到应答。服务器在看到应答之前逐步增加同一时刻发送的包数目以尽力填满 1750ms 的时限。我对超过 14000 个传输中的包的窗口进行了观测。

客户机有一个可配置选项，在每个应答传输前，把与最近到达的数据相关的 SACK 信息增加到每个应答上。如果启用此选项，则产生的应答包括最多四个 SACK 选项。每个选项都指向当前应答延迟了的四个最先到达包中的一个 1 字节的随机范围。如果正等待的包不足四个，则产生相应数目的选项。

为了进行有意义的比较，我从三个不同的位置同时收集此数据：

1. **基准：**有首次测试中，我先确定基准评测的方法。没有使用定制的客户机和 TCP 栈，而是使用了标准的 Linux TCP 协议栈和 wget 命令行 HTTP 客户机。
2. **定制，不使用 SACK：**第二个数据点需要使用定制的引入大窗口的客户机，但根本没有使用 SACK 选项。这个数据点可以把由大窗口造成的基本影响与处理恶意 SACK 选项造成的影响区分开来。
3. **定制，启用 SACK：**最后的数据集使用了引入大窗口的客户机，同时也在每次发送 ACK 时增加四个 SACK 选项。

我使用的服务器比较陈旧，是 1.2GHz Athlon/XP 的服务器。

评测

表1.服务器使用率评测

| **方法** | **已处理的应答** | **SACK 处理过程中扫描的包** | **经历的时间** | **CPU 使用率** | **每个应答取样的滴答数** | **每传输 1KB 取样的滴答数** | **平均重传队列长度** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **基准** | 252,955 | 0 | 1:02 | 22% | 1.72 | 0.56 | 5 |
| **定制，不启用 SACK** | 498,275 | 0 | 2:59 | 9% | 1.47 | 1.03 | 7,000 - 10,000 |
| **定制，启用 SACK** | 534,202 | 755,368,500 | 12:47 | 33% | 10.87 | 8.13 | 1,414 |

**误导数据**

可以很容易地解释一些令人惊讶的数据点。

请注意定制的客户机的应答数很高，大约是基准 wget 的二倍。原因是定制的客户机的设计目的是进一步恶化服务器行为。因此，它不会对应答进行合并。如果两个相邻的应答不包括任何数据，则大多数 TCP 协议栈都会把两个应答合并成一个。定制的客户机可以提供这样的选项，但对于这个数据，没有启用这个选项。在启用此选项的情况下又进行了一次测量，结果证明这并不能从本质上影响结果。要想了解原因，请阅读下面的 [使用更大的服务器](https://www.ibm.com/developerworks/cn/linux/l-tcp-sack/#bigserver)小节。

读者也会注意到这样一个现象：触发 SACK 路径的节点每传输 1KB 的数据的滴答数是基准数的 16 倍之高，但 CPU 使用率的差异只为 1.5，比较适中。另一个关键的数据集是测试经历的时间。基准占用了 22% 的 CPU，占用时间为一分钟多一点，而启用 SACK 选项的客户机占用了 33% 的 CPU，占用时间为 13 分钟左右。最终，启用 SACK 选项的客户机每传输同样数目的数据会占用更多的 CPU 周期。

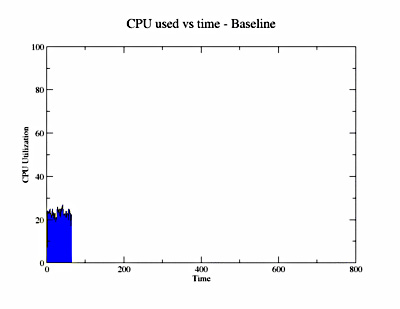
乍看起来，这些测量值看起来并不是非常糟糕。虽然在攻击过程中 CPU 使用率达到 33%，但它并没有完全耗尽。如果 CPU 被完全占用，其它工作就无法完成，这会造成拒绝服务。

遗憾的是，深入分析这些数据则会出现问题。总的传输时间达到：从基准的 1 分钟达到完全攻击场景中的 13 分钟。另外，与基准的 1 分钟不同，在全部的 13 分钟内，增加的 CPU 使用率一直保持较高的值。整体上而言，达到同样的目标需要花费更多的 CPU 周期。您可通过比较三种数据点中每传输 1KB 数据的取样滴答数清晰地看到这一点。

继续深入分析会发现 33% 的 CPU 使用率会误导我们。13 分钟内包括一个反复的 CPU 激增循环，在这期间，整个服务器的占用率为 100%。这些 CPU 激增之后，CPU 使用趋于平稳，然后进行下一次循环。总体结果是平均使用率为 33%，但很长一段时间内 CPU 被远程主机触发的 TCP 处理完全占用。

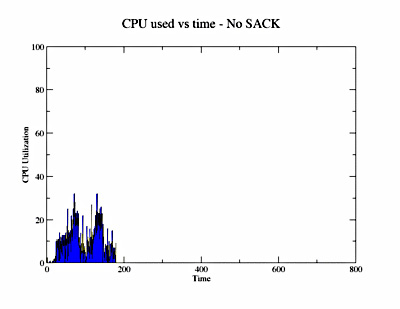
考虑这三种场景下 CPU 占用率和时间图：

图 1. 不启用 SACK 时使用 wget 的基准评测



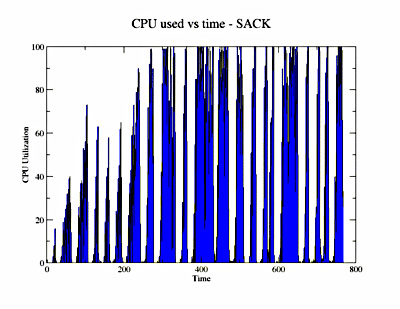
基准评测结果良好并且效率很高。传输很快完成并充分利用了可用带宽，而任何时刻 CPU 的占用率都未超过 25%。这证明了这类中等服务器在适合的环境下可以很好地利用网络。

图 2. 未启用 SACK 选项的大窗口定制客户机



定制客户机在未启用 SACK 的情况下也会生成一个合理的使用率图。管理大窗口的复杂性以及不可避免的损失将导致更多的 CPU 被占用，但在任何时刻，服务器都能保持正常运行。

图 3. 启用 SACK 选项的大窗口定制客户机



您肯定会被启用 SACK 的下载具有如此高的 CPU 使用率所震惊。虽然整体使用率均值为 33%，图中清晰地显示出反复出现的 CPU 峰值，在这期间服务器被全部占用。

图中反复描绘曲线 y=x^2 直到 y=100%。这表明每个 SACK 选项需要扫描整个重传队列以便找到它指向的数据。当发送方的拥塞窗口增大时，它会在发送出一个包和收到应答之间的时间内成倍地增加传输中的包的数目。对每个接收到的 SACK 选项都需要对这种翻倍的队列进行检查。令人惊讶的是，收到 50 万个应答的情况下，SACK 处理需要执行 7.55 亿次包比较。这种算法造成图呈现指数形式。

**使用更大的服务器**

实验最初的问题是仅仅用了普通的 1.2GHz Athlon 处理器作为服务器。这似乎低估了服务器端的处理能力。

可是，测试更快的服务器或实现优化（如把两个应答合并为一个）无法使这里看到的二次曲线有所改善。这种改变将使拥塞窗口超出这些评测中看到的 3200 个包的限制，但是服务器端的 CPU 使用率的曲辊式增长在将窗口增至 10000 或更高（这对充塞快速以太网是必要的）之前仍然会达到最大。

最后的疑惑是为什么这对服务器而言不是灾难性的。可以想象到使用率增加到 100% 并在随后的传输过程中保持这一水平。可是，我们却看到一系列呈周期性的使用率。这似乎不象是重传队列正在缩短。或者正是这样？

实际上，重传队列确实是周期性地一直减少到 0 个包。此时这个过程重新开始，拥塞窗口又一次增大，而 SACK 开销也是如此。整个过程在几秒种后又会达到可以影响服务器 CPU 使用率的大小。

队列开始减少，原因是高峰时大量的处理量触发了 TCP 基于超时的恢复机制。栈对超时的响应是把拥塞窗口重设回空状态。这个新的小窗口对传输速度有所阻碍，但当窗口再次增长到危险边缘时能保持服务器有效并可处理其它工作。

内核开发

正在开发哪些项目？

Linux 网络团队已经着手这些代码的工作。在 2007 年 11 月 15 日，Ilpo Jarvinen 发布了对 SACK 处理路径的重大修订。在 2008 年 1 月 28 日的合并窗口工作中，这些代码放入 Linus 的 pre-2.6.25 树中。全部系列是 10 个补丁（参见 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-tcp-sack/#artrelatedtopics) 获取这些内容的链接）。这里我将重点放在感兴趣的三个关键补丁上。

编写这些补丁的目的是在一般情况下改善 SACK 的性能。它们也会对攻击场景有些帮助，但不能把它们看成是本文所评测场景的解决方案。

Ilpo Jarvinen [2]：“我认为我们无法阻止恶意用户找到一些方法绕开典型、合法的用例中的调整和优化。”

第一个变化（参见 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-tcp-sack/#artrelatedtopics) 中的 “Abstract tp->highest\_sack accessing & point to next skb”）是在 SACK 选项只包含比已经选择性应答的数据的序列号高的数据信息时优化缓冲策略。通常来讲，这意味着在未完成的窗口中还存在一个已经报告的漏洞，而新数据在窗口的末端已经被选择性应答。对于常规操作这很常见。这个补丁通过把已缓存的引用从一个序列号转换为指向队列中最高的包（过去已经选择性应答）的指针从而对此情况加以优化。通过这一信息，接下来的补丁（参见 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-tcp-sack/#artrelatedtopics) “Rewrite SACK block processing & sack\_recv\_cache use”）通过缓存的指针作为列表遍历的起点就可以在处理 SACK 时只处理比缓存的值更高的数据，这为遍历列表节省了大量的工作。

遗憾的是，此策略不能对恶意的测试客户机进行优化。来自此客户机的典型应答包括一个 SACK 选项，它指向比前面出现的数据更高的数据，但它也包含指向前一个相邻包的序列。2.6.25 的实现需要从起点遍历重传队列以便找到数据。

后一个补丁包含对将来补丁中不同的 “跳跃式” 队列遍历算法的识别和支持。而这不会对此处提到的测试结果立刻有所帮助 — 因为跳跃遍历仍然通过线性遍历实现 — 它所支持的未来改变将对攻击场景产生重大影响。

这些补丁中包含的一些注释表明将来两个主要的改变正在进行中。第一个可能的未来改变是对未应答的包列表进行修改，因此包列表的组织形式不再是当前的线性列表而是通过索引组织成红－黑树。这允许对 SACK 选项指向的包进行 log(N) 查找。这个改变引入一些索引，允许对大的重传队列中的元素进行随机访问，这个改变对于解决 TCP 协议栈的 find-first 攻击问题很重要。

另一个结构上的改变解决了另一个本文未明确提出的问题。索引结构可以为个别包查找提供良好的性能，但 SACK 选项可能会覆盖包括多个包在内的任意多的字节数。如果恶意客户机发送的选项覆盖窗口中几乎全部的数据，则无法对此加以阻止。这与我关注的 find-first 攻击有所不同。实际上，第一个包可能会是列表中的第一个包，找到它可能非常容易。可是，如果需要对整个队列线性遍历来对 SACK 选项加以处理的话，快速找到感兴趣的包可能意义不大。这里的代码改变是把当前列表重新组织成 2 个列表，一个是已经选择性应答的数据，另一个是未选择性应答的数据。这很有帮助，因为它把搜索区域压缩到以前未选择性应答的数据范围中。称为 DSACK（复制 SACK）的相关规范也会引入一些复杂性，但分割正是我们思考的方向。

最后一个补丁（参见 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-tcp-sack/#artrelatedtopics) 中的 “non-FACK SACK follows conservative SACK loss recovery”）对拥塞控制语义进行修改以利用 RFC3517 中的 SACK 规则。这些改变允许内核在更多的情况下避免基于超时的恢复场景。这些基于超时的恢复机制要求把发送窗口一直缩减然后随时间逐渐增大到当前带宽延迟产物支持的水平。恢复时间负责中断测试期间发生的激增行为。

评测 2.6.25-pre

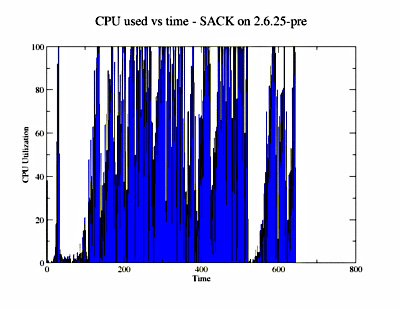
准备好这些新代码后，现在开始重新测量使用定制的延迟引入客户机（启用 SACK 选项）的数据点。此时，测试在 2.6.25 开发代码上完成。表中包含了早期的三个数据点便于参考。

表2　服务器使用率评测

| **方法** | **已处理的应答** | **SACK 处理中已经扫描的包** | **经历的时间** | **CPU 使用率** | **每个应答的取样滴答数** | **每传输 1KB 的取样滴答数** | **平均重传队列长度** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **基准** | 252,955 | 0 | 1:02 | 22% | 1.72 | 0.56 | 5 |
| **定制，不启用 SACK** | 498,275 | 0 | 2:59 | 9% | 1.47 | 1.03 | 7,000 - 10,000 |
| **定制，启用 SACK** | 534,202 | 755,368,500 | 12:47 | 33% | 10.87 | 8.13 | 1,414 |
| **定制，启用 pre-2.6.25 上的 SACK** | 530,879 | 2,768,229,472 | 10:42 | 49% | 13.6 | 10.07 | 5,214 |

此图是 pre-2.6.25 开发内核上启用恶意 SACK 选项时使用定制大窗口客户机的情况下，CPU 随时间变化的使用率：

图 4. pre-2.6.25 内核上启用 SACK 的大窗口定制客户机



此前，CPU 图中包含表示启动和停止的周期，现在则是一直占满。虽然内核代码的效率变得更高，但完成测试中同样多的文件传输占用更多的周期。这是一个不合理的结果。

新代码确实非常快，但是在每次测量中它都会长时间独占 CPU 并使用更多的全局 CPU 时间。这些情况的原因中缺少的一环是在新内核中减少了 TCP 基于超时的恢复机制，原因是与 RFC3517 有关的改变。在测试客户的每次运行中，2.6.22 代码平均有 17 次超时。而 2.5.25 代码平均仅为 2 次。超时事件间的空闲上升时间越来越少，把这个结果画成图很引人关注，结果造成停机时间更少。

超时更少意味着发送方要保持一个均值更大的窗口。大窗口对于高延迟链接中的正式吞吐量很重要。这个 TCP 栈的开发版在传输速度方面很有好处，完成时间比已经部署的栈快 2 分钟，原因是它可以保持更大的打开窗口。

可是，这些较大的平均窗口也意味着 SACK 接收代码需要对收到的每个包做很多工作，因为需要扫描的队列包含更多的包。文件传输中已扫描的 27 亿个包（是以前内核版本的 4 倍）以及每 KB 传输需要的 10.07 个取样滴答很精确地说明了需要做多少工作。

更快一些的处理器也无法显著改善这种情况。更快的处理器在同样的时间内会扫描更长的包链，但这样也会造成窗口略微增加，这使得对每个需要处理的新选项的工作量增大。为处理同样数目的 SACK 选项需要占用更多的处理周期数；更快的处理器带来自己的更大开销，而无法完成更多的工作。

结束语

恶意手动创建的 SACK 选项对性能的影响非常大，但还不至于上升到通常可实际运用的 DoS 攻击的层次，但其他客户机可以调节自己的节奏，占用服务器但并不将其压制到超时点，这不难想象。

不需要发送大块数据的计算机无需关心这个问题，因为这些计算机从不会填满整个大窗口，而这正是这一问题的前提。尽管选择性应答在高带宽延迟的网络连接上对性能的影响很大，但也可将其作为一个可禁用的选项，这不会牺牲互操作性。把 sysctl 的变量 net.ipv4.tcp\_sack设置为 0 即可禁用 TCP 协议栈中的 SACK 功能。

在当前的 Linux 内核开发树中，有一些关于普通 SACK 处理的工作进行得很好。这为将来的开发工作如包列表索引与分割打下了基础，而这些工作会减少一些攻击向量。