利用多线程技术改造 Snort 系统

林国庆, 王新梅
(西安电子科技大学 综合业务网理论及关键技术国家重点实验室,陕西 西安 710071)

摘要: Snort 是一个基于规则的轻量级网络入侵检测系统. 为提高 Snort 系统的性能,针对其工作流程是单线程的特征,用处理模块间设置缓冲队列、各个协议解码器和链表节点设置忙闲标识等方法实现了对其的多线程改造,并详细描述了改造后系统的工作流程,最后结合简化模型模拟实验结果,分析了改造前后的系统各性能的变化. 改造后的系统在检测速度和漏检率等性能方面有所提高,但也增加了 CPU 的工作量和内存的使用量.

关键词: 网络安全; Snort; 网络入侵检测系统; 多线程; Snort 工作流程
中图分类号: TP393 **文献标识码**: A **文章编号**: 1001-2400(2007)06-0887-08

Reform of the Snort system by the multithreading technique

LIN Guo-qing, WANG Xin-mei
(State Key Lab. of Integrated Service Networks, Xidian Univ., Xi'an 710071, China)

Abstract: The Snort system is a lightweight network intrusion detection system based on rules. In this paper, the principle, the basic structure and the workflow of this system are analyzed. Aiming at the Snort system working in a single thread, a reform scheme based on the multithreading technique for developing its performance is put forward, including a queue between two function modules and a busy sign flag in every decoder and chain node. The workflow of the reformed system is described then. Finally, the performance of the reformed system is analyzed theoretically associating with the result of a simulated experiment with a simplified model, which shows the detection efficiency is increased and the rate of miss-detection is decreased, but the workloads of CPU and the computer memory are increased.

Key Words: network safety; Snort; network intrusion detection system; multithreading; Snort workflow

Snort[1]是一个基于 libpcap[2]的轻量级网络入侵检测系统[2,3]. 它运行在一个"传感器"主机上,负责监听网络数据. Snort 能够把网络数据和系统中设定的检测规则进行模式匹配,从而检测出各种可能的入侵企图;还可以应用 Spade 插件,即采用统计学方法对网络上进行的异常攻击进行检测,并将入侵企图以文本、XML 等多种格式记录到 syslog(系统日志)或者数据库中. Snort 采用易于扩展的模块化体系结构,允许开发人员添加各种模块来扩展 Snort 的功能. 这些模块包括: HTTP 解码预处理器插件、TCP 数据流重组预处理器插件、端口扫描检测预处理器插件以及各种日志或者报警输入方式插件等. Snort 系统支持多种系统软硬件平台,如 Unix, Linux 和 Windows 等系统平台. 以下主要以 Snort-2.3.3 版本进行分析.

Snort 由 3 个重要的子系统构成:数据包捕获和解码子系统、检测引擎、日志与警报子系统.

1) 数据包捕获和解码子系统 数据包捕获和解码子系统利用 libpcap 库函数采集数据,并按 TCP/IP 协议的不同层次将数据包进行解析,经预处理后交给检测引擎进行规则匹配. 解码器运行在各种协议栈之上,从网络接口层到网络层,最后到传输层.

预处理器的引入使得 Snort 的功能可以很容易地扩展,用户和程序员能够将模块化的插件方便地融入 Snort 之中. 预处理程序代码在探测引擎被调用之前运行,但在数据包译码之后. 通过这个机制,数据包可以

收稿日期: 2007-08-05
基金项目: 国家自然科学基金资助(90604009);国家青年科学基金资助(60503010);国家"十一五"密码发展基金资助
作者简介: 林国庆(1978-),男,西安电子科技大学博士研究生.

通过额外的方法被修改或分析. 使用 Preprocessor 关键字加载和配置预处理程序. 预处理器模块主要有 3 种功能: ①数据包分片重组及数据流重组; ②处理一些特殊编码,以方便后续处理; ③协议异常检测,即检测型预处理器.

2) 检测引擎:检测引擎是 Snort 的核心,准确和快速是衡量其性能的重要指标. 前者主要取决于对入侵行为特征码提取的精确性和规则撰写的简洁实用性,后者主要取决于引擎的组织结构是否能够快速地进行规则匹配. 其中规则处理模块是检测引擎的关键模块. 规则处理模块完成与规则有关的各种功能,可以分为两个部分:Snort 规则解析和 Snort 规则检测. 规则解析在程序初始化的时候完成,作用是分析规则文件内容,把规则加载到内存中形成规则链表. 规则检测的作用是发现数据包是否符合某条规则.

3) 日志和报警子系统:日志与报警子系统用于对被检测到的数据包进行相应的输出处理,以输出插件的形式实现.

1 Snort 系统初始化过程

主控模块在 SnortMain()函数中. 当 Snort 初始化运行时,先从 main()函数开始执行,再调用 SnortMain()函数,然后 SnortMain()函数调用 ParseCmdLine()函数,分析运行指令,设置运行参数,没有设置的参数就调用默认的参数形式,进行 PV 结构的填充和初始化. 如果所有的初始化参数都正确,开始调用 OpenPcap()函数初始化数据源,初始化要加载的各种插件,根据网络数据包来源设置解码函数,然后对预处理插件进行初始化,根据规则文件生成相应的规则链表,然后设置好输出函数,循环地抓取数据包,检测数据包,直到结束程序运行. 详细介绍请参考文献[2],该过程如图 1 所示.

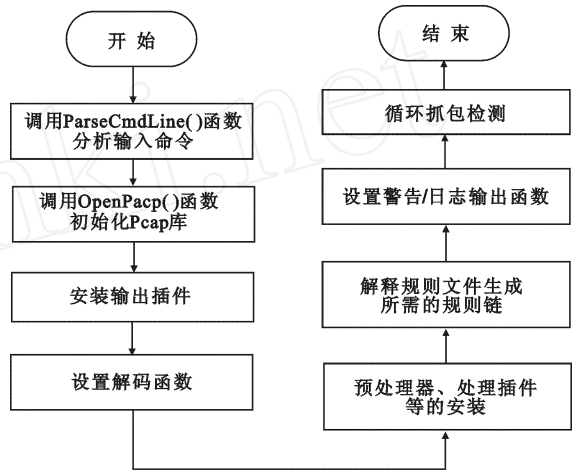

图 1 Snort 系统工作流程

2 初始化后内存中存在的链表结构和数据结构

2.1 数据包解码器

数据包解码器由各网络层次上针对不同协议的多个解码器组成,用于对原始数据包的详细解析. 数据包解码器结构如图 2 所示.

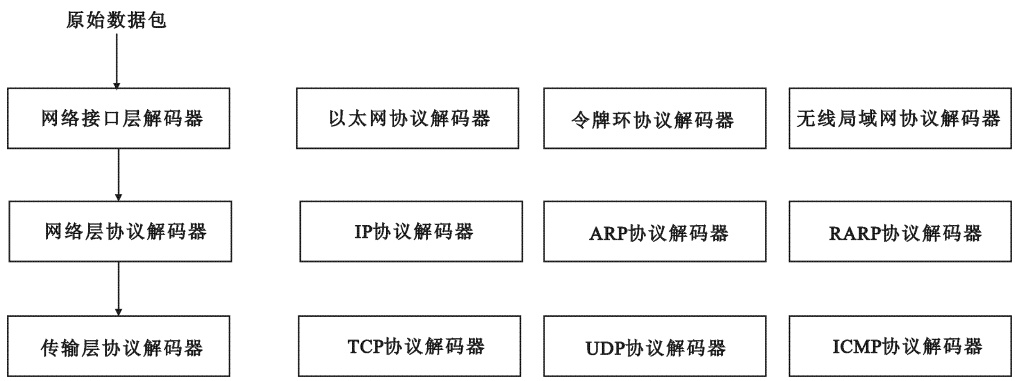

图 2 数据包解码器结构示意图

2.2 预处理器链表

预处理器链表也叫预处理函数链表,是预处理器插件被解析的产物,在规则解析完毕后生成,用于对解码后的数据包进行检查或处理. 每个链表节点都包含两部分:预处理器处理函数指针和指向下一个链表节点

的指针. 预处理器链表结构如图 3 所示.

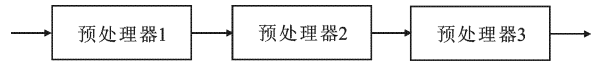

图 3　预处理器链表结构示意图

2.3　三维检测链表与数据结构 PORT_RULE_MAP

Snort 将所有已知的攻击以规则的形式存放在规则库中,为了能够快速准确地进行检测,Snort 将检测规则利用链表的形式进行组织,分为两部分:规则头和规则选项. Snort 规则中在圆括号前的部分是规则头部. 规则头部包含了定义数据包"从哪里来,到什么地方去,干什么",以及发现满足这个规则所有条件的数据包时应该干什么的消息. 规则头部包括规则操作、协议、IP 地址和端口以及方向操作符. 规则选项包括报警信息、需要检测的模式信息、索引信息. 一个规则的规则选项中可能有多个选项,不同选项之间使用";"分隔开了,它们之间为"与"的关系. 选项由关键字和参数组成,每个关键字和它的参数间使用冒号":"分隔.

规则头对应于规则树节点 RTN (Rule Tree Node),包含源 (目的) 地址、源 (目的) 端口号等. 规则选项对应于规则选项节点 OTN (Optional Tree Node),包含报警信息 (msg)、匹配内容 (content)、TCP 标志位 (flags) 等选项.

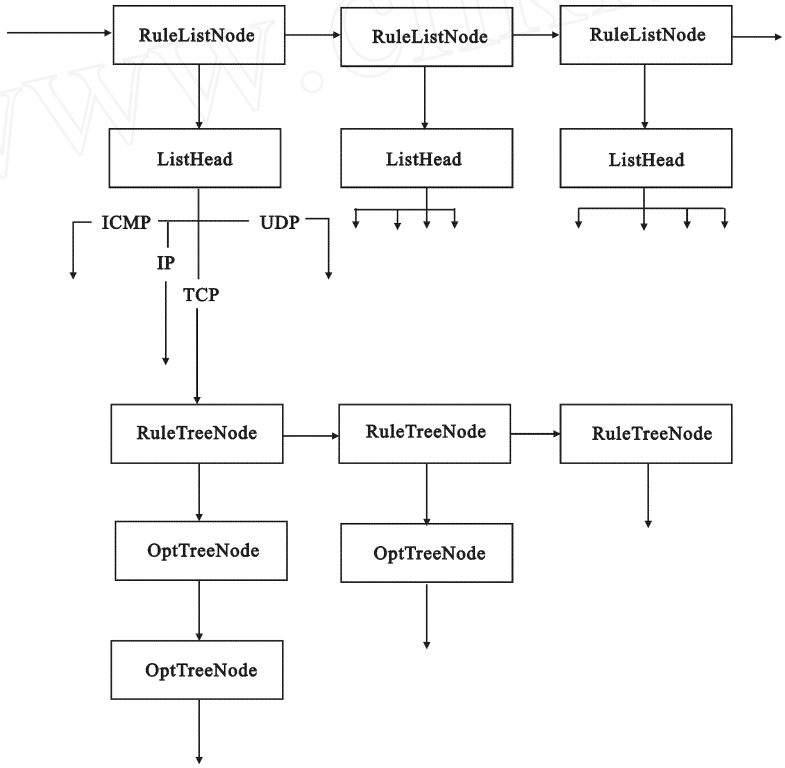

图 4　规则链表结构示意图

Snort 初始化并解析规则时,首先根据 Snort 规则头部中的 5 种操作规则 (alert,log,pass,activate,dynamic) 生成了 5 条规则链表,每条链表节点又通过 ListHead 结构类型的 RuleList 指针指向下面的各子规则链表,子规则链表为根据 Snort 规则头部中的协议划分的 TCP,UDP,ICMP 和 IP 子规则链表. 每一条子规则链表中包含独立的三维链表:规则头(RTN)、规则选项(OTN)和指向匹配函数的指针. 规则链表的层次结构如图 4 所示.

三维链表构造完成后,系统继续构造快速规则匹配引擎,依次读入规则链表中对应每条规则的节点的信息,建立用于快速规则匹配的数据结构 PORT_RULE_MAP.

构建快速规则匹配引擎的关键在于有效地划分规则集合. 为此,Snort 按照规则中的目的端口和源端口进行分类:

(1) 如果源端口值为特定值,目的端口为任意值(ANY),则该规则加入到源端口值对应的子集合中. 如果目的端口的值为特定值,则该规则同时加入到目的端口对应的子集合中.

(2) 如果目的端口值为特定值,源端口为任意值(ANY),则该规则加入到目的端口值对应的子集合中.如果源端口的值也为特定值,则该规则同时加入到源端口对应的子集合中.

(3) 如果目的端口和源端口都为任意值,则该规则加入到通用子集合中.

(4) 对于规则中端口值求反操作或者指定值范围的情况,等于端口值为 ANY 情况.

构建快速规则匹配引擎的执行过程如下:

(1) 根据规则协议类型,分别处理;

(2) 如果协议是 TCP 或 UDP 的话,根据规则中的目的端口的源端口值类型作相应处理,转(5);

(3) 如果是 ICMP 或 IP 协议,则根据规则中的目的端口的源端口值类型作相应处理并加入到 PORT_RULE_MAP 结构中;

(4) 判断相关内容的匹配选项的类型进行相应处理:①包含 content 内容匹配选项,将规则加入到相应的子集合 Content 中;②包含 uricontent 内容匹配选项,将规则加入到相应的子集合 Uri-Content 中;③无内容匹配选项,将规则加入到对应的子集合 NO-Content 中;

(5) 如果还没有遍历完毕,则继续遍历,转(1);

(6) 遍历已完毕,处理各个 PORT_RULE_MAP 的通用规则子集合;

(7) 处理各 PORT_RULE_MAP 的内容匹配模式信息.

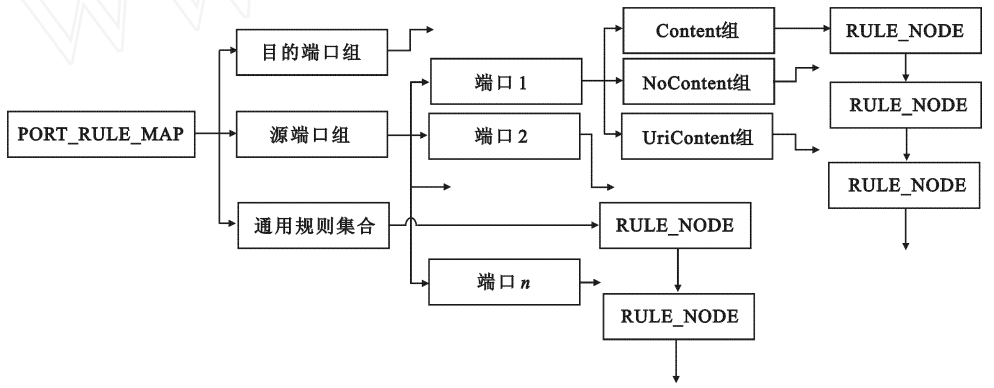

图 5 PORT_RULE_MAP 结构示意图

数据结构 PORT_RULE_MAP 如图 5 所示,其 RULE_NODE 链表中每个节点包含一个 OTNX 的指针,指向一个具体的原始链表的规则节点.

```
Typedef struct _otnx_{
OptTreeNode  *otn
RuleTreeNode  *rtn
} OTNX.
```

2.4 输出函数链表

日志和报警分别对应着两个不同的链表 Log 和 Alert,统称为输出链表或者输出函数链表,它们用于具体输出模式的实现.链表的每个节点都包含两部分:输出处理函数指针和指向下一个链表节点的指针.其结构如图 6 所示.

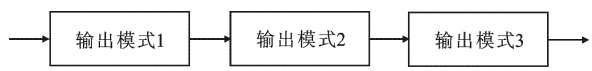

图 6 输出链表结构示意图

3 数据包的处理过程

Snort 系统开始工作后,依次进行以下处理:

(1) 捕获数据包 系统利用 libpcap 库函数采集到一个数据包后,随即送往数据包解码器.

(2) 数据包解码 系统初始化时,程序先获得网络接口层协议类型,存入指针 grinder 中.收到数据包后,系统根据指针 grinder 调用相应网络接口层协议解码器解析相应报文头.解析完毕,网络接口层协议解

析函数再调用相应网络层协议解码器解析相应网络层协议报文头,若存在选项字段(只限 IP 协议类型),调用相应选项字段解码器解析之.随后调用传输层协议解码器做类似处理.所有解析结果存入 Packet 数据结构的相应字段中.

(3) 预处理　解码后的数据包按要求匹配预处理器链表中的相应节点,即对解码后的数据包进行检测前的预处理,这些预处理器功能包括数据分片重组、流重组、代码转换等,也有部分预处理器是检测型处理器,用于实现检测链表所不具备的检测功能,如异常检测.

(4) 规则匹配　预处理完成后,对数据包进行规则链表匹配.先根据协议类型找到对应的规则子集合,然后根据端口值找到相应的 RULE_NODE 链表集合,按各个 RULE_NODE 链表中各节点的指针找到对应的原始规则链表节点,依次进行匹配.若发现与某节点相匹配,则按照该节点所设定的输出方式(若无具体输出设置则按默认方式)查找输出函数链表,进行具体的输出处理.

(5) 输出　当一个数据包与某检测型预处理器或者检测链表某节点相匹配时,则按照该节点所设定的输出方式(若无具体输出设置则按默认方式)查找输出函数链表,找到对应的输出模式节点,调用节点中处理函数,进行具体的输出操作.

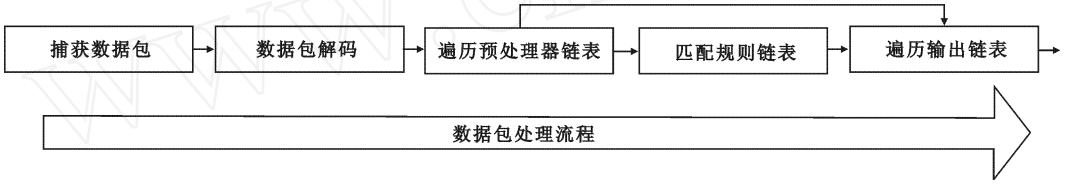

图 7　数据包的处理过程

详细介绍请参考文献[4,5].整个数据包的处理流程如图 7 所示.

4　多线程改造的原理与实现

4.1　多线程改造原理

从对数据包的处理过程的分析可知,当标准的 Snort 系统对一个数据包进行处理时,整个 Snort 系统采用单线程方式运行,从数据包的截取到进行输出处理过程中,系统均采用了串行单线程的方式加以处理.这就意味着当数据包在某个细节上进行处理时,内存中整个数据包处理系统的其他部分均处于闲置状态,这无疑是一个巨大的浪费,但同时也是提高 Snort 性能的一个可利用的资源.同时,在大流量的环境下,Snort 系统因为处理能力跟不上网络流量,造成严重的丢包漏检现象,这无疑造成了 Snort 系统性能上的不足.如果可以在同一时间充分利用系统中的各个功能模块乃至每个功能模块中的每个子模块,对多个数据包同时进行处理,必然可以提高单位时间内系统对数据包的处理量,从而减少丢包漏检的可能性,提高系统的整体性能.笔者采用的具体作法是对 Snort 系统各模块中的各个细节模块设置忙闲检测点,各功能模块间设置缓冲队列,同时对在处理的数据包都提供一个独立的线程.

利用一个简化的模拟实验介绍多线程改造的原理.

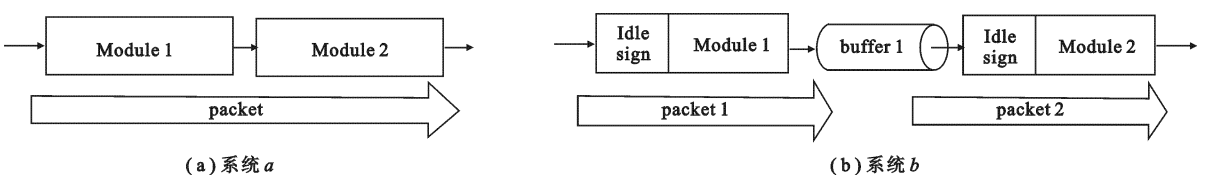

图 8　简化实验模型

如图 8 所示,系统 *a* 是 Snort 系统对数据包的处理过程的一个简化模型,由两个处理模块 Module 1 和 Module 2 组成的,同一时间一个处理模块只能对一个数据包进行处理.数据包依次经过两模块的处理.一个数据包经整个系统 *a* 处理完毕后,下一个数据包随即开始被处理.系统 *b* 是对系统 *a* 的改良,在两个处理模块

Module 1 和 Module 2 中分别设置忙闲标识,用以标识该模块是否处于工作中,同时在两模块间设置一个缓冲队列. 当系统开始处理一个数据包时,首先检查 Module 1 是否处于工作中,若为空闲,则该数据包占用该模块,即将忙闲标识设置为忙碌状态,随后开始被 Module 1 进行处理,处理完毕释放该模块,进入缓冲队列 buffer 1,排队等待进入模块 Module 2. Module 1 空闲后,下一个数据包随即占用该模块进行处理. Module 2 工作模式与 Module 1 相同. 数据包经 Module 2 处理完毕,结束整个处理过程.

实验目的:测定利用多线程技术改良后的系统 *b* 相对于原系统 *a* 在系统性能上的变化.

实验细节:应用 VC 语言实现整个模拟实验,整个实验系统包括 3 个部分:连续数据包生成器、系统 *a* 和系统 *b*. 利用连续数据包生成器生成数据包,分别发送到系统 *a* 和 *b* 中,从中得到所需实验数据.

连续数据包产生器包括两个部分:数据包生成器和缓冲队列. 数据包生成器的原理是每循环一定次数(1 000)后产生一个大小为 1 byte的虚拟数据包,送到缓冲队列中. 当前方系统(对于系统 *a* 前方系统为整个系统 *a*,对于系统 *b* 前方系统则是模块 Mudule1)为空闲时,缓冲队列中的第一个数据包被送到前方系统中进行处理. 当数据包生成速度远大于前方系统的处理速度时,连续数据包生成器就可以正常工作,即前方系统在处理完一个数据包后马上可以得到下一个数据包进行处理. 实验中缓冲队列设置得足够大,为 5 000 000 byte.

系统 *a*,*b* 中的处理模块 Module 1 和 Module 2 对虚拟数据包做空操作,它们的实际功能就是起到时延的作用,这个时延以循环一定次数(10 000)的形式实现. 系统 b 中的缓冲队列 buffer 1 大小设置得足够大,为 5 000 000byte.

表 1 是对两系统进行模拟实验的数据.

表 1 测试数据

处理包的数量	处理所有包耗时/ ms		CPU 平均使用率/ %		平均内存使用量/ kbyte	
	系统 *a*	系统 *b*	系统 *a*	系统 *b*	系统 *a*	系统 *b*
1 000	3 172	2 927	74	82	1 040	5 976
10 000	32 531	29 631	74	83	1 040	5 976
100 000	324 797	289 069	76	88	1 040	5 976

注:内存使用量不含连续数据包生成器所占内存. 两组模拟实验过程中 CPU 平均使用率统计包含相同的其他操作系统进程.

通过模拟实验的数据可以得出以下结论:(1)采用多线程技术的系统 *b* 在处理相同规格数量数据包时所用时间相对于系统 *a* 明显减少,这是因为利用忙闲标识和缓冲队列,提高了各模块的利用率,增加了系统处理数据包的能力;(2) 系统 *b* 所占用的系统资源,即 CPU 和内存的使用量,相对于系统 *a* 显著增加,前者是因为多线程增加了 CPU 的开销,后者主要是因为缓存队列占用了较多的内存空间. 显然,多线程改造后的系统 *b* 达到了使系统单位时间内处理更多数据包的目的.

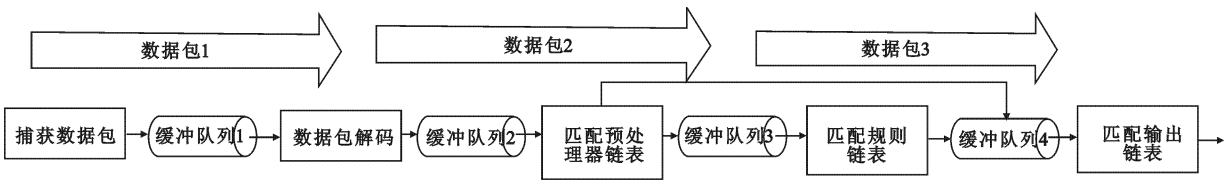

图 9 系统改造后数据包的处理过程

4.2 多线程处理的具体实现

利用上面的多线程改造原理,对 Snort 系统进行改造. 改造后的 Snort 系统流程如图 9 所示. 改造方案的主要细节如下:

1) 主要处理模块间设置缓冲队列. 为防止过多占用内存,缓冲队列大小上限应恰当设置. 方法是在实践过程中对每个缓冲队列中的排队数量做统计,最终得到合理的数值. 缓冲队列采取先到先服务的方式.

2) 在数据包解码器中,针对不同协议的解码器,都设置一个忙闲标识,用来标识该解码器是否处于工

作中.

3) 预处理器链表、输出链表中的每个节点以及三维链表中的每个规则选项节点,都设置一个忙闲标识,用来标识该节点是否处于工作中.

4) 对于每一个被解码后处于处理过程中的数据包,都用足够长的字母加数字位数(如 12)随机设定一个名字,用以标识该数据包. 该名字在开始解码时被赋予相应的 Packet 结构. 原系统中只有一个 Packet 结构类型变量,改造后的系统对于所有在处理的数据包都给予一静态 Packet 类型变量. 同时,对于每个在处理的数据包,都对应着独立的处理线程.

5) 在同一个处理模块中. 当一个数据包处理线程完成上一个处理单元(如匹配完一个节点),准备进入下一处理单元(如准备匹配下一个节点)时,发现该处理单元(下一个节点)处于忙碌状态,则该线程在占据当前处理单元(当前节点仍然标识为忙碌状态)的同时,不断查看下一个处理单元的忙闲状态,直到其变为空闲状态后,数据包占用此处理单元,并释放上一处理单元. 当多个数据包处理线程同时请求使用一个处理单元(如一个节点或者解码器)时,则该单元变为空闲状态后被某个数据包处理线程首先查询到,则该线程首先使用,其他的继续等待.

6) 改造后的系统对数据包的处理分为两个进程:数据包解码进程和入侵检测进程.

①数据包解码进程主要负责接收由 libpcap 接口从网络中捕获的每一个数据包,放入缓冲队列 1 中排队,而后将数据包放入解码器中解析,其中信息放入对应 Packet 结构中. 该进程的处理对象为原始数据包,并包含多个线程,每个数据包都对应着一个独立的处理线程.

②入侵检测进程主要负责接收由数据包解码器产生的每一个 Packet 结构变量,即解码后的数据包,并负责完成对数据包的一系列检测任务,直至数据包处理完毕. 该进程的处理对象为 Packet 结构变量,并包含多个线程,每个变量都对应着一个独立的处理线程.

系统改造后数据包的处理过程如下:

(1) 捕获数据包　系统利用 libpcap 库函数采集到一个数据包后,放入缓冲队列 1 中,随即继续采集下一个数据包.

(2) 数据包解码　系统不断将缓冲队列 1 最前的一个数据包取出,先创建一个 Packet 结构的静态变量,并给其一足够长的随机名字,然后按网络的层次顺序自下而上调用各种协议解码器来对数据包进行分析,并把分析结果存到该变量中. 网络接口层解析完毕后,该数据包会继续调用下一层的子解码器,系统随即从缓冲队列 1 中取出下一个数据包进行处理. 解码过程中,一个数据包处理线程在使用完一个协议子解码器后,先检查下一个需要调用的协议子解码器是否为空闲状态,如果是忙碌状态,则该数据包处理线程继续等待,直到其变为空闲状态后进行调用,随即释放上一解码器给下一数据包处理线程使用. 一个数据包被解码完毕后,将其对应的 Packet 结构的静态变量放入缓冲队列 2 中.

(3) 预处理　系统不断将缓冲队列 2 最前的一个数据包(Packet 结构的静态变量的形式)取出,依次按各自要求匹配预处理器链表中的相应节点. 同样地,当前数据包与节点 *a* 匹配完毕,占用下一个所需节点 *b* 后,释放 *a* 留给下一个数据包使用;当数据包所需下一节点为忙碌状态时,数据包占用当前节点并等待下一节点变为空闲. 一个数据包在预处理后,若被检测型预处理器发现有问题,则被直接送到缓冲队列 4,等待进行相应输出操作;其他情况则被送到缓冲队列 3 中,等待进行规则链表的匹配.

(4) 规则匹配　系统不断将缓冲队列 3 最前的一个数据包(Packet 结构的静态变量的形式)取出,依次匹配相应的规则链表节点. 对于每一个数据包,先根据协议类型找到对应的规则子集合,然后根据端口值找到相应的 RULE_NODE 链表集合,按各个 RULE_NODE 链表中各节点的指针找到对应的原始规则链表节点,依次进行匹配. 若发现与某节点相匹配,则按照该节点所设定的输出方式(若无具体输出设置则按默认方式)查找输出函数链表,进行具体的输出处理. 同样地,当前节点 *a* 匹配完毕,并占用下一个所需节点 *b* 后,释放 *a* 留给下一数据包使用;当数据包所需下一节点为忙碌状态时,数据包占用当前节点并等待下一节点变为空闲. 一个数据包在匹配完毕后,若发现有问题,则被送到缓冲队列 4,等待进行相应输出操作;其他情况就结束对该数据包的处理.

(5) 输出　系统不断将缓冲队列 4 最前的一个数据包(Packet 结构的静态变量的形式)取出,依次按照

所设定的输出方式(若无具体输出设置则按默认方式)查找输出函数链表,找到对应的输出模式节点,调用节点中相应的处理函数,进行具体的输出操作.若当前某节点被占用,则其他要调用该节点的数据包必须等待,直到其为空闲状态时方可使用.进行完输出操作后,对该数据包的整个处理过程结束.

5 结 论

改造后的 Snort 系统对数据包的处理由单线程串行变为多线程并行,在系统性能上有了很大提高.结合简化模型模拟实验的结果,可知改造后的系统在性能上将有以下变化:

1) 检测速度显著提高 改造后的系统充分利用了内存中的各处理模块,由在同一时刻只处理一个数据包变为同时对多个数据包进行处理,其单位时间处理数据包的数量增加,检测速度自然得到提高.

2) 相同流量下漏检率降低 由于改造后的系统检测速度提高,单位时间内能处理更多的数据包,降低了因处理能力不足造成漏检的可能.

3) 应对突发数据流能力增强 真实的网络环境中,系统对数据包的接收速度并不是匀速的.当突发的大数据流到来时,原系统会因为来不及处理而丢弃大量应该被检测的数据包;在此情况下,改造后的系统把这大量的数据包先放到缓存队列 1 中,随后慢慢进行处理,不至于丢弃数据包.

4) CPU 和内存的使用率显著增加 由于改造后的 Snort 系统线程数量大幅度增加, CPU 的工作量加大.内存中在处理的数据包数量也大幅增加,必然占用更多的存储空间.因此,改造后的系统要求更高的 CPU 处理速度和更大的内存空间,用硬件系统更大的工作量换取入侵检测系统处理速度的提高.

5) 单个数据包处理时间增加 在同一时刻多个数据包同时被处理,虽然内存中的各个模块资源被充分利用,但是当多个数据包同时要使用同一个资源(比如一个链表节点)时,必然产生等待现象,再加上在缓冲队列中的排队时间,处理单个数据包的时间必然增加.单个数据包处理时间增加,也降低了系统对数据包处理的实时性.但是,与整体检测速度显著地提高相比,这一代价是可以接受的.

通过引进多线程技术,Snort 系统检测速度提高,漏检率降低,应对突发数据流能力增强,但是 CPU 和内存的使用率显著增加,这是一个"空间换时间"的问题.虽然单个数据包处理时间也有所增加,但是多线程技术使得多个数据包可以同时被处理,因此整体检测速度仍然是增加的.下一步的工作将以多线程 Snort 系统的具体实现为重点,同时在理论上对该系统存在的一些问题,如防止缓冲队列可能溢出、控制 CPU 和内存的使用率防止系统崩溃等进行进一步研究.总而言之,多线程改造后的 Snort 系统提高了系统资源的利用率,改善了当前 Snort 系统难以适应大流量网络环境的问题.

参考文献:

[1] The Snort Project. Snort Users Manual[R/OL]. [2005-07-15]. www.snort.org.

[2] 唐正军. 入侵检测技术导论[M]. 北京: 机械工业出版社, 2004.

[3] 潘振业, 李学干. 基于模式匹配和协议分析的入侵检测系统设计[D]. 西安: 西安电子科技大学, 2005.

[4] 李春梅, 李学干. 基于网络的入侵检测系统的研究及实现[D]. 西安: 西安电子科技大学, 2005.

[5] 康振勇,田玉敏. 网络入侵检测系统 Snort 的研究与改进[D]. 西安:西安电子科技大学, 2006.

(编辑: 郭 华)