清华永新统一安全网关 TN-SG5000 V5.0

详细设计v1.0

江苏清华永新安全科技有限公司

版本历史

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版本 | 作者 | 参与者 | 完成日期 | 校验 | 备注 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**版权声明：**未经许可，不得以任何形式传播或提供非授权人阅读。谢谢！

目 录

第1章 总体架构 6

第2章 功能模块 7

2.1 网络基本功能 7

2.2 包过滤和状态检测 7

2.3 网络地址转换(NAT)： 9

2.4 用户管理 10

2.5 流量控制 10

2.6 IP/MAC地址绑定 11

2.7 虚拟专用网(VPN)： 11

2.8 日志与审计 12

2.8.1 日志头部 12

2.8.2 日志等级 13

2.8.3 日志类型 13

2.9 流量统计 13

2.10 安全管理： 14

2.11 高可用性 15

2.11.1 拓扑发现 15

2.11.2 角色选举 15

2.11.3 拓扑收敛 16

2.11.4 路由收敛 17

第3章 数据流 19

3.1 网络基本功能 19

3.2 包过滤和状态检测 19

3.3 NAT 20

3.4 用户管理 20

3.5 流量控制 22

3.6 IP/MAC地址绑定 23

3.7 VPN 23

3.8 日志审计 24

3.9 流量统计 24

3.10 安全管理 25

3.11 高可用性 25

第4章 数据结构 26

4.1 网络基本功能 26

4.2 包过滤和状态检测 32

4.3 NAT 34

4.4 用户管理 34

4.5 IP/MAC绑定 35

4.6 流量控制 36

4.7 VPN 37

4.8 日志与审计 41

4.8.1 事件日志正文 41

4.8.2 反病毒日志正文 41

4.8.3 网页过滤日志正文 41

4.8.4 IPS日志正文 42

4.8.5 traffic日志正文 42

4.9 流量统计 43

4.10 安全管理 46

4.11 高可用性 47

第5章 接口 49

5.1 网络基本功能 49

5.2 包过滤和状态检测接口 50

5.3 NAT 56

5.4 用户管理 57

5.5 流量控制 59

5.6 IP/MAC绑定 61

5.7 VPN 62

5.7.1 Path Prune 62

5.7.2 Path Graft 63

5.7.3 Topology Converge 63

5.7.4 Topology Expand 63

5.7.5 Topology Shrink 64

5.7.6 Topology Stale 64

5.8 日志与审计 64

5.9 流量统计 64

5.10 安全管理 66

5.11 高可靠性 68

5.11.1 处理HA协议报文 68

5.11.2 处理ACK报文 69

5.11.3 处理Hello报文 69

5.11.4 处理Convergence报文 69

5.11.5 处理Route Convergence报文 70

5.11.6 处理Path Prune报文 70

5.11.7 处理Self-Path-Prune报文 70

5.11.8 处理Merge报文 70

5.11.9 处理Restart报文 70

5.11.10 处理Leave报文 70

5.11.11 处理Stale报文 70

5.11.12 处理DMN报文 70

5.11.13 处理Link Detect报文 71

5.11.14 处理PORT SYNC报文 71

5.11.15 处理DATA报文 71

5.11.16 处理B2A报文 71

第6章 对应性分析 72

# 总体架构

采取业界的一般做法，逻辑上将整个系统分为控制面、数据面、管理面三个平面，公共服务平台提供设备、接口、HA框架等公共服务库框架。



本文将详细描述防火墙关键功能的设计。

# 功能模块

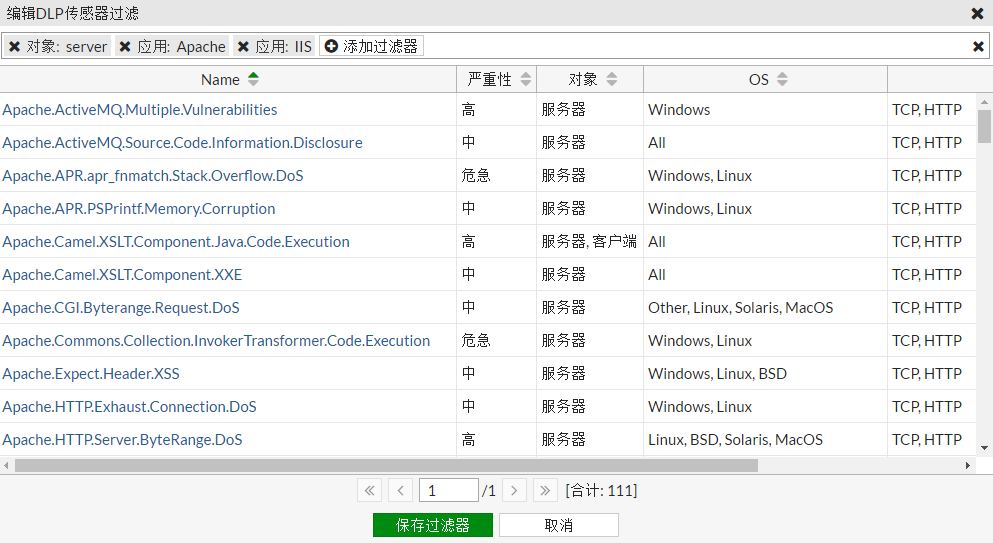
根据目前各级用户对防火墙安全产品功能的需求，**清华永新TN-SG5000 V5.0**防火墙系统设计功能模块如下：

## 网络基本功能

网络基本功能包括设备管理、网络接口管理、交换、路由（包括静态路由、动态路由、策略路由）。该部分功能基于linux操作系统，包装上层配置管理接口。

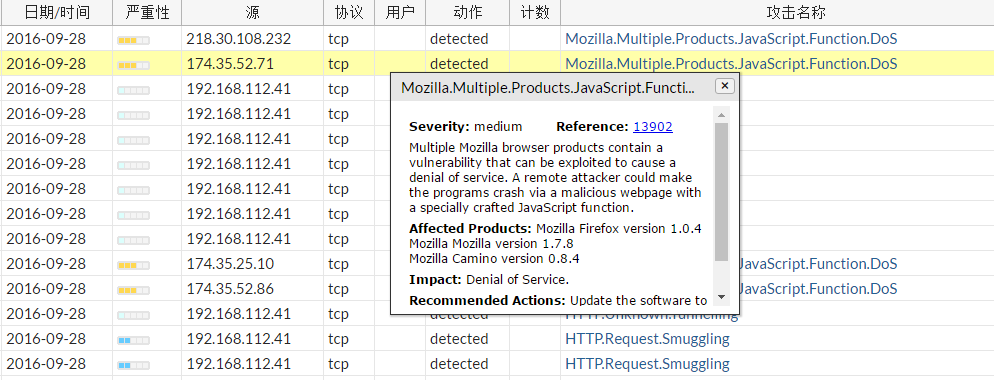
## 包过滤和状态检测

包过滤基于IPS，TNOS可以很方便的定义IPS特征过滤器，帮助用户迅速筛选对保护内部服务器有用的特征集合，并根据需要选择处理方式，与本用户网络及应用无关的特征被关闭，以免影响处理性能。例如，内网需要保护的服务器为IIS及Apache Web服务器，可以设置如下的IPS过滤器。

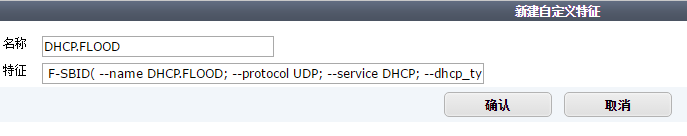


设置完成后，所有匹配的特征将会显示在界面中。

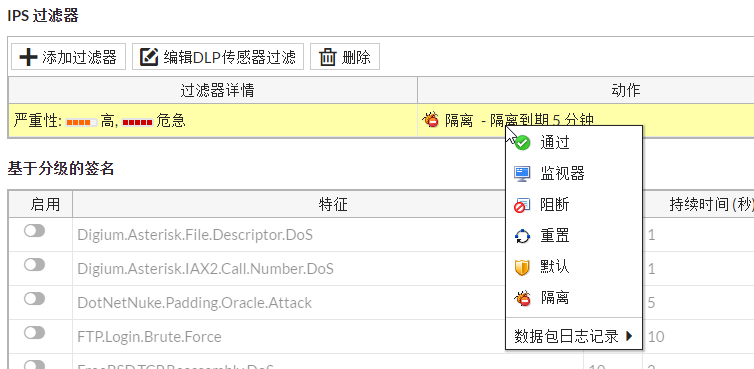
TNOS的IPS特征库内置10000多种攻击特征，并自动通过 Internet更新，确保用户在第一时间实现对最新攻击方式的防御。通过图形界面，可以查看内置特征库条目，点击每个特征的名称，可以弹出对此攻击简介的窗口。



除了系统自带的入侵特征，TNOS也支持用户自定义特征。



对于已发现的攻击，TNOS不但可以对攻击数据包进行阻挡，还可以隔离攻击者的来源IP地址。有效的防止攻击者对内部网络不断发送攻击。



TNOS内置的IPS还可以对SYN flood、ICMP flood等DoS/DDoS攻击进行防御，对于每一种DoS/DDoS攻击行为，都可以设置阈值，使策略符合实际网络环境和应用情况，降低误报和漏报率。



TNOS防DDoS功能可以针对不同的源或目的IP地址的TCP、UDP、ICMP会话数量进行限制，该功能不但可以防止蠕虫病毒爆发时导致的DoS/DDoS攻击，还可以对P2P等高并发会话应用进行有效抑制。

## 网络地址转换(NAT)：

网络地址转换（Network Address Translation）简称为 NAT，是将 IP 数据包包头中的 IP

地址转换为另一个 IP 地址的协议。当 IP 数据包通过路由器或者安全网关时，路由器或者安全网关会把 IP 数据包的源 IP 地址和/或者目的 IP 地址进行转换。在实际应用中，NAT 主要用于私有网络访问外部网络或外部网络访问私有网络的情况。NAT 有以下优点：

  通过使用少量的公有 IP 地址代表多数的私有 IP 地址，缓解了可用 IP 地址空间枯竭的速度。

  NAT 可以隐藏私有网络，达到保护私有网络的目的。

私有网络一般使用私有地址，RFC1918 规定的三类私有地址如下：

  A 类：10.0.0.0 - 10.255.255.255（10.0.0.0/8）

  B 类：172.16.0.0 - 172.31.255.255（172.16.0.0/12）

  C 类：192.168.0.0 - 192.168.255.255（192.168.0.0/16）

上述三个范围的 IP 地址不会在因特网上被分配，因而可以不必向 ISP（Internet Service

Provider）或注册中心申请，而在公司或企业内部自由使用。

NAT 的基本转换过程

安全网关执行 NAT 功能时，处于公有网络和私有网络的连接处。图 17-1 描述了 NAT 的基本转换过程。于同一个本地认证服务器的用户组可以划分到不同的用户组中，并且同一个用户组可以同时属于不同的用户组。图 4-1 以缺省本地 AAA 认证服务器“Local”的用户配置说明用户以及用户组关系。

## 用户管理

安全网关支持以下基于角色的用户身份认证功能：

  基于角色的策略规则：实现不同用户的访问控制。

  基于角色的 QoS：实现不同用户的带宽控制。

  基于角色的统计集：统计不同用户的带宽、会话数以及新建会话速率。

  基于角色的会话限制：实现对特定用户的会话数限制。

  SCVPN 基于角色的主机安全检测：实现不同用户对特定资源的访问控制。

  基于角色的策略路由：实现根据不同源用户选择路由。

## 流量控制

流量控制基于QOS实现。QoS（Quality of Service）即“服务质量”。它是指网络为特定流量提供更高优先服务的同

时控制抖动和延迟的能力，并且能够降低数据传输丢包率。当网络过载或拥塞时，QoS 能够确保重要业务流量的正常传输。

QoS 是网络中管理数据流的可用带宽、延迟、抖动以及分组丢失的技术集合。所有的 QoS 机制的目的就是影响这些特征中的至少一个，某些情况下甚至是全部。

通常来讲，实现 QoS 管理功能的工具包括：

  分类和标记工具

  管制和整形工具

  拥塞管理工具

  拥塞避免工具

## IP/MAC地址绑定

IP-MAC绑定是很多路由，网关的一个常见功能，主要用来绑定局域网终端的IP地址，防止终端随便修改IP，导致网内IP地址冲突，而导致网络歇火。先来说IP-MAC绑定的原理分两个部分：前部分，所谓绑定，就是让这台电脑的MAC始终获取同一个ip地址或者指定ip，这个做到很容易，很多路由都号称有这样功能称之IP-MAC绑定。但是做到这一步，没啥意义，因为上网终端一旦修改了IP，照样可以上网，照样IP冲突，真正夯实IP-MAC绑定的是后半部分，这个上网行为管理才可以做到（核心交换机也能做到，但是配置很麻烦费事）要有上网数据，通过行为管理来检测这个这个MAC是不是一直都是获取的这个IP，一旦不是获取的这个IP或者IP被擅自修改了，就会让MAC断网。或者一旦有新的没有登记再案的终端进来（电脑，路由或者手机）进来，即使插上，也无法上网

## 虚拟专用网(VPN)：

Internet应用中，不可避免的要通过公用网络来传输信息，其中就有可能包括机密信息。由于Internet是一个开放的、公用的网络，黑客很容易通过在网络设备上安装网络嗅包器(如Sniffer、NIDS等)中途窃取信息，造成泄密；黑客也可以伪装成内部用户登录到内网中进行破坏活动，因此我们需要在网络上配置一整套VPN体系，对通过Internet进行的远程访问进行严格的认证和加密，使Internet上的VPN成为经过加密和认证的安全链路，保证各节点之间远程访问的安全。

采用VPN技术的目的是为了在不安全的信道上实现安全信息传输，保证内部信息在Internet上传输时的机密性和完整性，同时对Internet上传输的数据进行认证。

单独的VPN网关的主要功能是数据包的加密/解密处理和身份认证，没有很强的访问控制功能（状态包过滤、网络内容过滤、防DoS攻击等）。在独立的防火墙和VPN部署方式下，防火墙无法对VPN的数据流量进行任何访问控制，由此带来安全性、性能、管理上的一系列问题。因此，在防火墙安全网关上集成VPN能提供一个灵活、高效、完整的安全方案,是当前安全产品的发展趋势。它可以保证加密的流量在解密后，同样需要经过严格的访问控制策略的检查，保护VPN网关免受DoS攻击和入侵威胁；提供更好的处理性能，简化网络管理的任务，快速适应动态、变化的网络环境。因此，VPN技术已经成为安全网关产品的组成部分。TNOS便是一个集防火墙、VPN和应用层过滤网关功能于一身的综合安全网关，可以提供各安全功能之间的完美联动、良好的兼容性和性能。

TNOS 的VPN功能支持PPTP、L2TP、IPSec和SSL四种VPN协议，提供了前所未有的方便和灵活的选择。对远程移动用户或企业的出差用户来说，既可以使用Windows等系统自带的PPTP/L2TP拨号软件， 也可以使用IPSec客户端软件FortiClient和企业建立VPN的连接，还可以直接使用IE浏览器，通过Web方式创建基于SSL的VPN隧道。应用PPTP、L2TP、SSL的好处是方便使用，不需要附加的软件。而用 IPSec 客户端软件的好处是高度的安全性保证，可以采用动态的密钥保证数据的安全。在企业本地网络和远程网络之间可以采用IPSec协议来实现VPN的连接和数据的高度安全控制。配置简单灵活。

## 日志与审计

tnos 日志消息可以分成两部分：

* 日志头部
* 日志正文

例如：

**2018-02-02 11:25:07 EVENT-INTERFACE-WARNING:**user=’’ ui=’’ msg="Interface ethernet0 change to physical up"

其中粗体部分是日志头部。

### 日志头部

日志头部包含以下信息：

**YYYY:MM:DD HH:MM:SS Type Subtype Level**

* **YYYY:MM:DD：**日期，显示事件发生的年、月、日。
* **HH:MM:SS：**时间，显示事件发生的时、分、秒。
* **Type：**类型，日志的类型可以分为事件、会话、NAT、反病毒、IDS、网页过滤，参见表2。
* **Subtype：**子类型，日志的子类型，由每个功能模块自信定义，参见表-1。
* **Level：**日志等级，表明事件的严重程度，分为 8 个级别，从紧急到调试。参见表1

根据日志消息发送到的地点的不同，日志报头的格式稍有差别。如果日志打印到本地，格式如下：

**2018-02-02 11:25:07 EVENT-INTERFACE-WARNING:**

如果日志打印到文件中，格式如下：

**2018-02-02 11:25:07 type=EVENT subtype=INTERFACE level=WARNING:**

### 日志等级

Tsinghuanet将日志记录的严重程度分为 8 个级别。在 CLI 中使用级别编号配置严重程度级别。

表格 1 日志等级

|  |  |  |  |
| --- | --- | --- | --- |
| 级别 | 级别号 | 描述 | 日志定义 |
| 紧急（Emergencies） | 1 | 系统不可用信息 | EMER |
| 警报（Alerts） | 2 | 需要立即处理的信息 | ALERTS |
| 严重（Critical） | 3 | 危急信息 | CRITICAL |
| 错误（Errors） | 4 | 错误信息 | ERROR |
| 警告（Warnings） | 5 | 报警信息 | WARNING |
| 通告（notice） | 6 | 非错误信息 | NOTIFICATION |
| 信息（Informational） | 7 | 通知信息 | INFO |
| 调试（Debugging） | 8 | 调试信息 | DEBUG |

### 日志类型

表格 2 日志类型

|  |  |  |
| --- | --- | --- |
| 类型 | 子类型 | 描述 |
| 事件（event） | system |  |
|  | config |  |
|  | Interface |  |
|  | ARP |  |
|  | QOS |  |
|  | 由业务模块自行定义 |  |
|  |  |  |
| 流量（traffic） |  |  |
| 反病毒（Antivirus） |  |  |
| 入侵防御系统（IPS） |  |  |
| 网页过滤（Web Filter） |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## 流量统计

方便管理员实时查看网络接口流量，根据用户名、IP地址、协议、时间等参数统计流量，及时了解网络中的流量情况。

## 安全管理：

安全网关具有强大的安全保护功能，例如，攻击防护、IP-MAC 绑定

以及会话控制等。本章具体介绍以下功能：

  主机防御

  主机黑名单

  IP-MAC 绑定

  DHCP 监控

  ARP 检查

  ARP 防御

除此外TNOS统一威胁管理系统提供了全面的内容过滤功能包括：Web内容过滤、防垃圾邮件、信息泄露及SSL加密数据过滤等安全功能，轻松解决内容安全顾虑，提高企业或者学校等用户的工作效率，帮助用户解决后顾之忧。



## 高可用性

### 拓扑发现

* 拓扑发现

HA启动后，HA拓扑状态进入**Building状态**，HA拓扑中只有本地节点。HA会从所有UP的VSU口向外发送Hello报文；同时也在这些VSU口上监听其他设备的Hello报文。在此，我们把未建立拓扑连接的VSU口称为**边缘接口（Edge Interface）**，把存在边缘接口的设备称为**边缘设备（Edge Node）**。

当HA收到Hello报文后，从中解析出拓扑节点信息，查找HA拓扑中是否有对应节点。如果拓扑中没有对应节点，则新建节点；如果有，则视情况更新节点的信息。

Hello报文的直连邻居（即收到Hello报文时发现Hop等于1）在收到Hello报文后，会检查报文中是否包含本节点的入口（即源节点报文出口的直连口）信息，如果没有包含，直连邻居会将直连信息补充到报文中，然后将报文的Hop加1后从非入口的VSU口转发出去，这样其他成员收到Hello报文后，可以建立关于源节点的直连信息。Hello报文在拓扑中以洪泛的形式传播，每传播一跳，报文的Hop值加1，直至报文截止于拓扑末端或者源节点。

通过Hello报文的洪泛，拓扑中的所有成员都可以学到其他成员的拓扑信息，并建立起相应的拓扑关系。

### 角色选举

* 角色选举

HA的设备角色分为：Active Master（AM），Backup Master（BM），Candidate，其中AM和BM统称Master。

HA启动后，设备角色默认为Candidate，HA会运行一个名为Master Election Timer（简称MET）的定时器。在MET老化前，HA处于拓扑发现的过程中，每次有新的节点加入拓扑，HA会刷新MET。

当MET老化或者HA发现拓扑的形状为环形时，HA结束拓扑发现阶段，拓扑中的所有成员根据Master竞选规则开始Master角色的选举。

* **AM竞选规则**

AM的竞选规则如下（从第一条开始判断，如果本条无法选出AM，继续判断下一条）：

1. SM（Static Master，静态主设备）优先。
2. 成员优先级高的优先。
3. MAC地址小的优先。

* **BM的竞选规则**

AM选举出来后，HA按照BM竞选规则从剩余的成员中选出BM（从第一条开始判断，如果本条无法选出BM，继续判断下一条）：

1. SM优先。
2. 距离AM最近的优先。
3. 成员优先级高的优先。
4. MAC地址小的优先。

根据Master选举规则，AM选出来的BM肯定是它的直连邻居。

### 拓扑收敛

角色选举完成后，HA开始首次拓扑收敛。拓扑收敛由AM发起，拓扑的全员参与完成整个拓扑的收敛。

* **首次收敛**
* **过程描述**

1. 被选为AM的节点执行下列动作：
   * 1. 检查当前拓扑成员的数量是否超过规格上限，如果超规格则需要进行超规格裁剪。超规格裁剪原则为：如果是环形拓扑，则裁剪AM对位的节点；如果是线形拓扑，则裁剪离AM最远的边缘设备。
     2. 检查拓扑的设备编号（MID，即Switch ID）是否冲突，如果MID冲突，那么AM会重新分配MID。MID分配规则为：AM的MID指派为1，然后按照加入AM的本地拓扑的顺序为所有节点递增指派MID（RSR77分布式设备不支持MID冲突检测，在配置时要求每台设备配置不同的MID）。
     3. 根据BM竞选规则，选出BM。
     4. 向拓扑发布Convergence报文，Convergence报文中会包含AM竞选结果、BM竞选结果（如果有的话）、MID分配结果等信息。并且，在一定周期内，AM会持续发布Convergence报文，直到整个拓扑收敛或者超过预期的时间（60秒）。
2. 非AM节点收到Convergence报文后，检查收敛报文是否正确，如果正确则根据报文内容设置AM，设置BM，更新MID等信息，然后向拓扑发布Convergence Ack报文。如果非AM节点发现本地节点不在MID分配列表中，则认为自己不应该在AM所在的拓扑中（通常这种情况是由拓扑成员数量超规格引起的），本地节点将会离开拓扑，从而进入**Stale**状态。
3. 当节点被选为AM时，其拓扑中本地节点状态为收敛状态；当节点收到Convergence报文，其拓扑中AM节点和本地节点状态为拓扑收敛状态；当节点收到Convergence Ack报文时，其拓扑中Convergence Ack报文源对应的节点状态为拓扑收敛状态。当本地拓扑中的所有节点状态都为拓扑收敛状态时，HA进行拓扑收敛，HA会将拓扑信息下发给各注册模块。
4. 拓扑收敛后，HA拓扑状态由**Building**状态进入**Convergent**状态，拓扑建立完成，相应的拓扑称为收敛拓扑。

* **异常情况考虑**

1. 强制收敛

如果在一定的时间内，拓扑不能正常收敛，拓扑会删除所有邻居节点，然后本机单独收敛。之后，收敛的各设备可以通过拓扑合并（见后面“分裂与合并”一节描述）重新组合成VSU。

我们把限定拓扑收敛的时间称为Convergence Timer（简称CT）。

1. 拓扑发现不对称

非AM节点收到AM的Convergence报文时，需要检查主机上学习到的拓扑信息与本机是否一致，具体需要检查的项目包括：

* + 1. 本地拓扑中是否可以找到对应的AM节点，若找不到则认为Convergence报文是非法的，丢弃处理。
    2. 如果Convergence报文中有要求设置BM，而本地拓扑中又找不到对应的BM节点，则认为报文是非法的，予以丢弃。
    3. 对于Convergence报文中每个指派的节点，在本地拓扑中是否都能找到，如果有任何节点找不到，则认为报文是非法的，予以丢弃。
    4. 在没有发生成员数量超规格的情况下，还应该检查Convergence报文中声称的拓扑大小、拓扑形状和本地拓扑的拓扑大小、拓扑形状是否一致，如果不一致，则认为报文是非法的，予以丢弃。
* 拓扑重新收敛

在首次拓扑收敛后，如果拓扑发生了变化，会重新进行拓扑收敛，拓扑收敛的过程同首次拓扑收敛的过程类似，也是由主机收敛，成员设备相应收敛。在后面的“拓扑维护”章节，还会进一步分析各种拓扑状态进入拓扑收敛的场景。

### 路由收敛

当HA完成拓扑收敛后，就开始进行路由收敛。

当AM上具备路由收敛条件时，AM就开始触发路由收敛过程。AM会向拓扑发送路由收敛报文，成员设备收到报文后，设置AM具备路由收敛条件，然后检查拓扑上所有节点是否都具备路由收敛条件，如果具备则回复路由收敛确认报文，然后进行路由收敛。

成员设备收到路由收敛确认报文时，会设置报文源对应的节点具备路由收敛条件，然后检查拓扑上所有节点是否都具备路由收敛条件，如果具备则回复路由收敛确认报文，然后进行路由收敛。

路由收敛时，单播路由及组播路由的计算详见后面的“HA路由表”描述。

在拓扑收敛或拓扑进入recovery状态时，HA都会发生路由收敛。

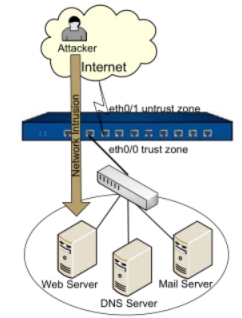
# 数据流

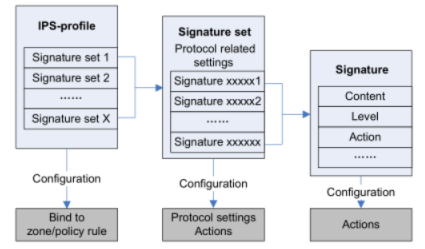
## 网络基本功能

网络层基本功能基于linux内核和用户Networking进程。



## 包过滤和状态检测





如果策略规则绑定了 IPS Profile，同时源安全域和目的安全域也绑定了 IPS Profile，系统 IPS

检测的优先级由高到低依次为：策略规则的 IPS Profile>目的安全域的 IPS Profile>源安全域的

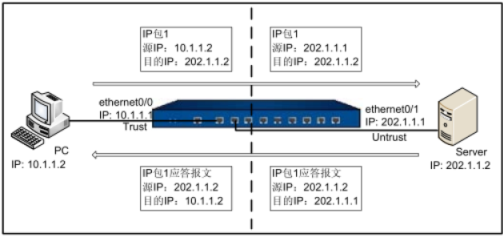
IPS Profile。

为系统配置 IPS 功能后，当系统发现入侵攻击，会生成相应的 IPS 日志信息。IPS 日志信息中

包含检测出的攻击特征 ID，根据特征 ID，用户可以访问相应的 IPS 在线帮助页面查看攻击特征的

具体信息。

## NAT



## 用户管理

【用户登录】



图 4 用户登录流图

如上图所示，经过traefik代理，登录请求首先到达dummy auth service，在此处实现认证安全策略（多次失败登录锁定、告警等），对有有效的请求再进一步访问keystone获取token返回。

【访问请求】



图 5 后端访问流图

登录成功后，后端请求都将携带授权用户的token，经traefik代理，请求直接到达对应业务服务后端：

* keystone middleware首先完成token的有效验证，将解析的roles list/project id等信息附加至request header；
* authorization plugin根据被访问URL的权限要求设定，向dummy auth service申请鉴权结果；
* 前列token验证、鉴权均通过后回调被访问URL的响应API。

【权限管理】



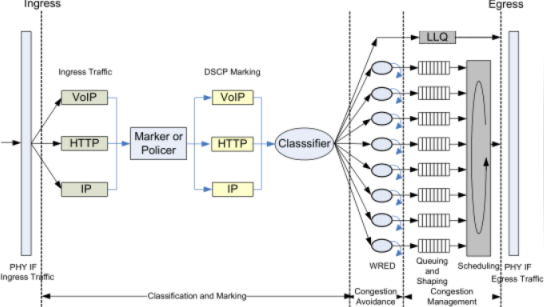
图 6 权限管理流图

权限管理请求分为两个响应后端：

* Keystone处理：user/role/project的创建，role的分配由keystone直接响应；

Dummy auth service：基于角色的策略规则管理、用户认证安全策略管理等由dummy auth service响应

## 流量控制



数据包通过入接口进入系统后，首先会被分类和标记。在这一过程中，系统

会通过管制机制丢弃一些数据包。然后，根据标记结果，数据包会被再次分类。系统会通过拥塞管

理（Congestion Management）机制和拥塞避免（Congestion Avoidence）机制对数据包进行管理，为数据包排列优先次序并且在发生拥塞时保证高优先级数据包的顺利通过。最后，系统会

将经过 QoS 管理的数据包通过出接口发送出去

## IP/MAC地址绑定

不涉及数据流

## VPN





## 日志审计

无数据流

## 流量统计

不涉及数据流

## 安全管理



## 高可用性

高可用性不涉及数据流

# 数据结构

## 网络基本功能

/\* 接口类型 \*/

typedef enum {

LLCB\_SWPORT, /\* switch port \*/

LLCB\_RU\_AP, /\* 本次新增，AP口 \*/

} LLCB\_TYPE;

**以下的数据结构均位于已有的<include/sys/ll\_intfevent.h>中：**

/\*\*

\* 接口事件类型索引

\*/

enum intf\_event\_type\_index\_e {

/\* 系统由于switchover而重启 \*/

K\_EVENT\_SWITCHOVER\_INDEX,

/\*端口加入AP的事件，异步接收时apifx可能为0或是错误的\*/

INTF\_EVENT\_ADD\_TO\_AP\_INDEX,

/\*端口退出AP的事件，apifx为0，无法提供是从哪个AP退出的\*/

INTF\_EVENT\_REMOVE\_FROM\_AP\_INDEX,

};

**以下的数据结构均位于已有的<include/sys/pktcb.h>中：**

**/\*** 进程面报文buffer **\*/**

struct sk\_buff {

/\* 用来限制隧道报文的本地封装层数，由隧道模 块进行维护。\*/

unsigned char tunnel\_encap\_level;

/\* 用来标记接收报文AP口的成员口 \*/

uint ap\_member\_index;

}；

typedef enum{

/\* 26: IP Egress起始标志 \*/

PKT\_SEQ\_EGRESS,

/\* 27: AG均衡处理 必须是出接口的第一个业务\*/

PKT\_SEQ\_AP\_BLC,

}；

**以下的数据结构均位于已有的<include/net/linklayer/ethernet/ ethernet \_def.h>中：**

struct struct\_ether\_cb{

/\* 配置检测，响应信息处理 \*/

card\_chk\_rsp\_func cfg\_chk\_rsp;

/\* 新增以太口对应的AP口端口号 \*/

ushort ape\_num;

/\* 成员口记录AP的ifx \*/

IFINDEX apifx;

/\* AP成员口是否是shutdwon, 加入AP之后是否执行了shutdown \*/

boolean AP\_shut;

/\* 用于标识与AP相关的转换状态 \*/

int AP\_status;

/\* 标识加入AP前是否被shutdown \*/

boolean AP\_shut\_switch;

/\* 用于标识端口设置的MAC地址监听个数\*/

ulong mac\_set\_num;

};

/\* 以太模块进程面通告快转面的事件 \*/

typedef enum ethernet\_info\_type\_ {

ETHERNET\_INFO\_VLAN\_ADD,

/\* 创建AP接口 \*/

ETHERNET\_INFO\_AP\_CREATE,

/\* 删除AP接口 \*/

ETHERNET\_INFO\_AP\_DELETE,

/\* 成员口加入AP接口 \*/

ETHERNET\_INFO\_AP\_ADD\_MEM,

/\* 成员口退出AP接口 \*/

ETHERNET\_INFO\_AP\_DEL\_MEM,

/\* 配置AP接口的负载均衡 \*/

ETHERNET\_INFO\_AP\_LOAD,

/\* BFD绑定AP接口的成员口 \*/

ETHERNET\_INFO\_BFD\_MEMBER\_ADD,

/\* BFD解绑定AP接口的成员口 \*/

ETHERNET\_INFO\_BFD\_MEMBER\_DEL,

} ethernet\_info\_type;

/\* 进程面向快转面传递的消息 \*/

typedef struct ethernet\_info\_s {

uint vlanid;

/\* AP口端口号 \*/

uchar ap\_num;

/\* 成员口的接口索引 \*/

IFINDEX member\_ifx;

/\* 配置BFD绑定的对端IP \*/

IPADDRTYPE peer\_ip;

} ethernet\_info\_t;

**以下的数据结构均位于已有的<include/net/networklayer/bfd.h>中：**

**/\*** 向BFD模块转送的消息结构 **\*/**

typedef struct bfd\_ip4\_session\_info\_s {

/\* 进程ID \*/

uint process\_id;

/\* AP成员口的接口索引 \*/

IFINDEX member\_idx;

};

**/\*** BFD模块向外通告的消息结构 **\*/**

typedef struct bfd\_ip4\_session\_chg\_info\_s {

/\* 增加vc id作为标识 \*/

u\_int32\_t vc\_id;

/\* AP成员口的接口索引 \*/

IFINDEX member\_idx;

}；

**/\*** BFD热备消息 **\*/**

typedef struct bfd\_rdnd\_msg\_s {

**/\*** BFD热备接口 **\*/**

int intf\_idx;

/\* AP成员口的接口索引 \*/

IFINDEX member\_idx;

} bfd\_rdnd\_msg\_t；

/\* 解析注册消息 \*/

typedef struct bfd\_msg\_info\_s {

**/\*** BFD注册的接口索引**\*/**

int ifindex;

/\* AP成员口的接口索引 \*/

IFINDEX member\_idx;

} bfd\_msg\_info\_t；

**以下的数据结构均位于新增的<include/net/linklayer/ethernet.h>中：**

/\* 定义位图集合数据结构及其操作 \*/

#define ETH\_BITMAP(bitmAP\_size) \

struct { ulong \_\_set\_\_[(bitmAP\_size + 31) / 32]; }

/\* 查询集合的成员\*/

#define ETH\_BITMAP\_IS\_MEMBER\_OF(bitmAP, a) \

(((bitmAP)->\_\_set\_\_[(a) / 32]) & (1UL << ((a) % 32)))

/\* 添加集合的成员 \*/

#define ETH\_BITMAP\_ADD\_MEMBER(bitmAP, a) \

((bitmAP)->\_\_set\_\_[(a) / 32]) |= (1UL<<((a) % 32))

/\* 删除集合的成员 \*/

#define ETH\_BITMAP\_REMOVE\_MEMBER(bitmAP, a) \

((bitmAP)->\_\_set\_\_[(a)/32]) &= (~(1UL<<((a) % 32)))

/\* 以太口最大接口数量 \*/

#define CFG\_ETH\_MAX\_PORT\_NUMBER (4096)

/\* 以太口最大接口数量 \*/

#define CFG\_ETH\_MAX\_PORT\_NUMBER (4096)

/\* AP成员端口集合 \*/

typedef ETH\_BITMAP(CFG\_ETH\_MAX\_PORT\_NUMBER) eth\_ap\_members\_set\_t;

/\* AP口的最大数量 \*/

#define CONFIG\_BRI\_MAX\_ETH\_AP\_NUMBER 16

/\* 每个AP口支持8个成员口\*/

#define ETH\_AP\_MAX\_MEMBER\_NUMBER 8

/\* AP成员口绑定bfd最大数量g \*/

#define CONFIG\_MEMBER\_BIND\_BFD\_MAX\_NUMBER 128

/\* AP流量平衡算法类型 \*/

enum eth\_ap\_loadbaltype\_e {

ETH\_AP\_SRC\_MAC = 0x1, /\* 基于源MAC \*/

ETH\_AP\_DEST\_MAC = 0x2, /\* 基于目的MAC \*/

ETH\_AP\_SRCDEST\_IP = 0x4, /\* 基于源IP和目的IP \*/

ETH\_AP\_SRCDEST\_MAC = 0x8, /\* 基于源和目的MAC \*/

ETH\_AP\_SRC\_IP = 0x10, /\* 基于源IP \*/

ETH\_AP\_DEST\_IP = 0x20, /\* 基于目的IP \*/

ETH\_AP\_SRC\_L4PORT = 0x40, /\* 基于L4 层源端口号 \*/

ETH\_AP\_DEST\_L4PORT = 0x80, /\* 基于L4层目的端口号 \*/

ETH\_AP\_SRCDEST\_L4PORT = 0x100, /\* 基于L4层源端口号+L4层目的端口号 \*/

ETH\_AP\_SRC\_IP\_SRC\_L4POR = 0x200, /\* 基于源IP+L4层源端口号 \*/

ETH\_AP\_SRC\_IP\_DEST\_L4PORT = 0x400, /\* 基于源IP+L4层目的端口号 \*/

ETH\_AP\_DEST\_IP\_SRC\_L4PORT = 0x800, /\* 基于目的IP+L4层源端口号 \*/

ETH\_AP\_DSET\_IP\_DEST\_L4PORT = 0x1000, /\* 基于目的IP+L4层目的端口号 \*/

/\* 基于源IP+目的IP+L4层源端口号 \*/

ETH\_AP\_SRCDEST\_IP\_SRC\_L4PORT = 0x2000,

/\* 基于源IP+目的IP+L4层目的端口号 \*/

ETH\_AP\_SRCDEST\_IP\_DEST\_L4PORT = 0x4000,

/\* 基于源IP +L4层源端口号+L4层目的端口号 \*/

ETH\_AP\_SRC\_IP\_SRCDEST\_L4PORT = 0x8000,

/\* 基于目的IP +L4层源端口号+L4层目的端口号 \*/

ETH\_AP\_DEST\_IP\_SRCDEST\_L4PORT = 0x10000,

/\* 基于源IP+目的IP +L4层源端口号+L4层目的端口号 \*/

ETH\_AP\_SRCDSET\_IP\_SRCDEST\_L4PORT = 0x20000,

/\* 本卡优先的均衡策略 \*/

ETH\_AP\_LOCAL\_FIRST = 0x40000,

};

/\* AP控制面向AP快转面和线卡同步消息的类型 \*/

typedef enum ef\_eth\_ap\_info\_type\_ {

/\* AP口创建事件 \*/

EF\_ETHERNET\_INFO\_AP\_CREATE,

/\* AP口删除事件 \*/

EF\_ETHERNET\_INFO\_AP\_DELETE,

/\* 成员口加入AP口事件 \*/

EF\_ETHERNET\_INFO\_AP\_ADD\_MEM,

/\* 成员口退出AP口事件 \*/

EF\_ETHERNET\_INFO\_AP\_DEL\_MEM,

/\* 配置负载均衡事件 \*/

EF\_ETHERNET\_INFO\_AP\_LOAD,

/\* BFD绑定AP口成员口事件 \*/

EF\_ETHERNET\_INFO\_MEM\_BIND\_BFD\_ADD,

/\* BFD解绑定AP口成员口事件 \*/

EF\_ETHERNET\_INFO\_MEM\_BIND\_BFD\_DEL,

EF\_ETHERNET\_INFO\_AP\_MAX

} ef\_eth\_ap\_info\_type;

typedef struct eth\_ap\_p2ef\_msg\_ eth\_ap\_p2ef\_msg;

/\* AP控制面向AP快转面和线卡同步的消息结构 \*/

struct eth\_ap\_p2ef\_msg\_

{

/\* 向AP快转面和线卡发送的消息类型 \*/

ushort msg\_type;

/\* AP端口号\*/

ushort ap\_num;

/\* AP端口的全局索引 \*/

ulong APifgid;

/\* AP端口的本地索引 \*/

ulong APifidx;

/\* AP成员口全局的索引 \*/

ulong mbifgid;

/\* AP端口的本地索引 \*/

ulong mbifidx;

/\* AP端口的负载均衡类型 \*/

ulong ap\_load;

/\* AP成员口绑定BFD的对端IP \*/

IPADDRTYPE peer\_ip;

};

/\* 控制面AP口实体控制块 \*/

typedef struct eth\_ap\_entry\_s {

/\* AP号 \*/

ulong ape\_number;

/\* 该AP实体关联的接口LLCB控制块 \*/

struct llcb\_s \*ape\_llcb;

/\* 该AP实体的名字 \*/

char ape\_name[128];

/\* 该AP实体的引用计数 \*/

ulong ape\_refcount;

/\* AP的成员个数 \*/

ulong ape\_member\_count;

/\* LINK UP的成员口计数 \*/

ulong ape\_linkup\_count;

/\* 该AP的成员端口集合 \*/

eth\_ap\_members\_set\_t ape\_port\_member;

/\* AP口信息状态和AP成员口信息显示函数 \*/

show\_interface\_sup\_func AP\_show\_interface;

/\* 该AP的流量平衡算法 \*/

eth\_ap\_loadbaltype\_t balance\_type;

/\* AP成员的类型 \*/

LLCB\_TYPE llcb\_type;

/\* 判断AP口是否被删除了\*/

boolean is\_delete;

/\* bfd绑定成员口个数统计 \*/

ulong ape\_bind\_bfd\_count;

/\* AP成员口绑定BFD结构体 \*/

eth\_bfd\_bind\_member\_t bind\_bfd\_mem[8];

} eth\_ap\_entry\_t;

/\*

\* AP 成员口bfd状态

\*/

typedef enum {

/\* BFD会话由UP变为DOWN \*/

MEM\_BFD\_DOWN = 0,

/\* BFD会话由DOWN变为UP \*/

MEM\_BFD\_UP = 1,

};

/\* AP热备的消息类型 \*/

typedef enum AP\_APi\_msg\_type\_{

/\* 负载均衡 \*/

AP\_LOAD\_BALANCE,

/\* AP成员口加入AP口\*/

AP\_MEMBER\_ADD,

/\* AP成员口退出AP口 \*/

AP\_MEMBER\_DEL,

/\* BFD绑定AP成员口 \*/

AP\_MEMBER\_BFD\_ADD,

/\* BFD解绑定AP成员口 \*/

AP\_MEMBER\_BFD\_DEL,

/\* 设置AP口的MTU \*/

AP\_SET\_MTU,

/\* AP成员口状态变化 \*/

AP\_MEMBER\_STATE\_CHANGE,

/\* AP成员口的BFD链路状态变化 \*/

MEMBER\_BFD\_STATE\_CHANGE,

AP\_MAX

}AP\_APi\_msg\_type;

/\* AP热备结构 \*/

typedef struct AP\_rdnd\_msg\_s {

/\* 消息类型 \*/

ushort msg\_type;

/\* AP号 \*/

ushort ap\_num;

/\* 该AP的流量平衡算法 \*/

eth\_ap\_loadbaltype\_t balance\_type;

/\* 是否有前缀 \*/

boolean flag;

/\* 成员口索引号 \*/

IFINDEX mem\_idx;

/\* AP口索引 \*/

IFINDEX AP\_ifindex;

/\* BFD绑定的对端的IP \*/

IPADDRTYPE peer\_ip;

/\* AP口的MTU \*/

ulong mtu;

/\* AP成员口变换的状态 \*/

LLCB\_STATES newstate;

/\* AP成员口绑定的BFD变换的状态 \*/

BFD\_STATES bfdstate;

} AP\_rdnd\_msg\_t;

## 包过滤和状态检测

{

"task\_id": "", # 任务id，后端分配，全局唯一

"app": "", # 学习任务所属的app

"name": "", # 任务名称

"author": "", # 作者

"type": "", # 任务类型，目前包括： predict\_numeric/predict\_category/cluster

"description": "", # 任务描述

"create\_time": "",

"update\_time": "",

"model\_id": "", # 模型id，后端分配 predict\_numeric/predict\_category使用

"model\_name": "", # 训练后模型的名字 predict\_numeric/predict\_category使用

"model\_description": "", # 模型描述 predict\_numeric/predict\_category使用

"train\_id": "", # 当前使用的训练id

"train\_count": num, # 一个学习任务下，训练的次数

"trains": [], # train id

}

{

"train\_id": "", # 训练id，后端分配，全局唯一

"task\_id": "", # 关联的学习任务id

"model\_id": "", # 模型id，后端分配

"start\_time": "", # 训练开始时间

"end\_time": "", # 训练结束时间

"state": "", # 任务运行状态：waiting、running、complete

"active": true or false, # 是否被使用

"data": {

"index": {

"source\_type": "", # log or report

"name": "", # 如果是source\_type是log，name有效，表示分类日志的名字. 如果source\_type是report，name表示report result的pivot\_table的名字

"type": "", # 如果是source\_type是log，type有效，表示分类日志

}

"start": "", # 数据时间范围utc时间

"end": "", # 数据时间范围UTC时间

"search": {}, # es 搜索条件

"fields": [{"type": "", "name": ""}], # 该模型需要的字段

},

"preprocessing": [ # 数据预处理步骤，多个步骤

{

"method": "", # 数据预处理方法

"fields": [{"type": "", "name": ""}], # 预处理字段

"params": {"param": "value"}, # 预处理方法的参数

},

],

"algorithm": {

"name": "", # 算法的名字：

# 分类预测： LogisticRegression/SVM/RandomForestClassifier/GaussianNB/BernoulliNB/DecisionTreeClassifier

# 值预测：LinearRegression/RandomForestClassifier/Lasso/Ridge/DecisionTreeClassifier

# 聚类：KMeans/DBSCAN/Agglomerative

"params": { # 算法参数

"param": "value", # 多个参数

}

},

"predict\_field": {"type": "", "name": ""}, # 待预测的字段, predict\_numeric/predict\_category使用

"fields": [{"type": "", "name": ""}], # 使用的字段

"training\_scale": num, # 用于训练的数据百分比，默认70%， predict\_numeric/predict\_category使用

"result": { # 训练结果

"metrics": {

"accuracy": num, # 精度

"precision": num, # 准确率

"recall": num, # 召回率

"F1": num, # f度量，准确率与召回率的调和平均

"R2": num, # 回归模型的评估指标R平方

}

"result\_table": "", # 计算结果index，如下：ml\_cluster\_result， ml\_predict\_numeric\_result， ml\_predict\_category\_result

}

}

## NAT

ml\_cluster\_result: {

"id": "", # 每条数据id， es的\_id

"cluster": "", # cluster使用，表示所示类别。

"data": [num1, num2], # 数据点，对应fields

}

ml\_predict\_numeric\_result: {

"id": "", # 每条数据id， es的\_id

"actual": num, # 真实的数字

"predict": num, # 预测的数字

"residual": num, # predict\_numeric使用，残差

"data": [num1, num2], # 数据点

}

ml\_predict\_category\_result: {

"id": "", # 每条数据id， es的\_id

"actual": "", # 真实的分类

"predict": "", # 预测的分类

"data": [num1, num2], # 数据点，对应fields

}

## 用户管理

用户身份认证数据模型涉及的一些对象如下所示[3]：

* User：一个账号凭据，隶属于一个project或domain；
* Group：一组user，隶属于一个project或domain；
* Project：是所有权的基本单元，所有资源都应隶属于一个project，project隶属于一个domain，project默认属于default domain，project在domain下唯一；
* Domain：project/user/group的顶层容器，domain全局唯一；
* Role：与多个user-project对关联的第一类元数据；
* Token：与user或user和project相关的身份凭据；
* Extras：与user-project对关联的KV元数据桶；
* Rule：实施一个动作应满足的一系列条件；

## IP/MAC绑定

写入dev\_log\_alert

"uuid" : "416145cf-6a35-4b47-85ee-008c41b18ffa",

"group\_id" : 3,

#"@timestamp" : "2020-05-11T10:19:49.784Z",

"template\_name" : "VMWare-Esxi",

"index" : "dev\_log\_alert\_2020.05",

"level" : "HIGH",

"@version" : "1",

"rule\_name" : "I/O延迟增加超10000告警",

"rule\_id" : 9,

"description" : "在2020-05-11 18:09:49到2020-05-11 18:19:49期间由数据源<VMWare-Esxi>产生的日志匹配告警规则<I/O延迟增加超10000告警>，说明条件1：((value 最大值：当前值116925 > 期望值100000))",

"rt" : "2020-05-11T10:19:49.776435Z",

"typename" : "t3e95b63a08704026",

"query" : """{"query": {"bool": {"must": [{"range": {"rt": {"gt": "2020-05-11T10:09:49.777873Z", "lte": "2020-05-11T10:19:49.776435Z"}}}], "must\_not": [], "should": []}}, "aggs": {"max\_value": {"max": {"field": "value"}}}}""",

"result" : {

"max\_value" : 116925.0,

"count" : 8437

},

"end" : "2020-05-11T10:19:49.776435Z",

"group\_name" : "ESXI告警",

"start" : "2020-05-11T10:09:49.777873Z",

"\_app" : "esxi",

"status" : "unprocessed",

"suppress\_enable": true

"suppress\_seconds": 10

## 流量控制

从rulesrv\_sched\_event\_ml读取

{

"app\_context": "\_la",

"check\_count": 28463,

"check\_last": "2020-06-12 11:01:35",

"configs": {

"conditions": [

{

"level": "MEDIUM",

"op": "AND",

"value": [

{

"field": "",

"func": "count", # predict/residual

"op": "LT",

"value": "100000"

}

]

}

],

"extend\_query": {

"query": {

"bool": {

"must": [],

"must\_not": [],

"should": []

}

}

},

"index\_freq": "day",

"query": {

"bool": {

"must": [],

"must\_not": [],

"should": []

}

},

"range": 0,

"template\_name": "\u6d41\u91cf",

"typename": "\_traffic",

"model\_id": "", # 机器学习模型id

},

"created": "2020-06-08 10:17:51",

"description": "",

"group": {

"alert\_config": {},

"app\_context": "\_la",

"created": "2020-06-03 16:51:17",

"description": "",

"id": 1,

"name": "test",

"rule\_count": 1,

"type": "alert"

},

"group\_id": 1,

"id": 1,

"name": "test",

"pause": false,

"rule\_type": "event\_count",

"suppress\_enable": true,

"suppress\_seconds": 60,

"suppress\_status": true,

"suppress\_to": "2020-06-12 10:35:35",

"trigger": "interval",

"trigger\_args": {

"end\_date": "",

"seconds": 10,

"start\_date": ""

}

}

## VPN

下面是这些核心数据结构的定义：

|  |  |
| --- | --- |
| **软件目录** | **Tn/tn\_tm.h** |
| **名称** | **定义** |
| **tn\_if\_t** | typedef struct tn\_if\_s {  uchar vsl\_id; /\* VSU AP 口标识 \*/  uchar if\_state; /\* 协议状态, tn\_if\_state\_t \*/  uchar link\_state; /\* 链接状态, tn\_link\_state\_t \*/  uchar peer\_vsl\_id; /\* 对端VSU AP 口标识 \*/  ulong bw; /\* 带宽 \*/  ulong lat; /\* LAT \*/  void \* tn\_node; /\* 所属的拓扑节点 \*/  struct tn\_if\_s \* other\_if; /\* 另外一个VSU AP \*/  struct tn\_if\_s \* peer\_if; /\* 直连的VSU AP 口指针 \*/  uchar mbr\_num; /\* VSL成员端口数 \*/  uchar mbr\_ok\_num; /\* VSL成员端口检测OK数 \*/  uchar peer\_mac[6]; /\* 对端MAC用以标识身份 \*/  struct list\_head mbr\_list; /\* VSL成员端口列表 \*/  struct timeval timestamp; /\* VSL-AP UP的时间戳 \*/  tn\_if\_stat\_t pkt\_stat; /\* VSL-AP的报文收发统计信息 \*/  } tn\_if\_t, \* tn\_if\_p; |
| **tn\_node\_t** | typedef struct tn\_node\_s {  ulong distinction; /\* distinction \*/  uchar mac[6]; /\* MAC地址\*/  uchar array\_id; /\* 内部索引\*/  uchar mid; /\* 成员ID\*/  uchar topo\_state; /\* 拓扑状态\*/  uchar role; /\* 角色\*/  uchar sm\_option; /\* 角色掩码\*/  uchar priority; /\* 等级\*/  uchar if\_num; /\* TN AP 口数量 \*/  uchar node\_flag; /\* 标记, tn\_node\_flag\_enum\_t \*/  uchar ignore\_bm; /\* 选举BM时忽略此节点\*/  uchar ignore\_am; /\* 选举AM时忽略此节点\*/  ulong sv; /\* 软件版本\*/  ulong pre\_cfg\_state; /\* 结点预配置状态 \*/  tn\_if\_t if\_list[TN\_MAX\_IF\_NUM]; /\* TN AP 口列表\*/  tn\_rib\_array\_p rib\_array; /\* 单播表 \*/  tn\_mrib\_array\_p mrib\_array; /\* 组播表 \*/  vsu\_dev\_info\_t dev\_info;  } tn\_node\_t, \* tn\_node\_p; |
| **tn\_state\_t** | typedef struct tn\_state\_s{  uchar topo\_state; /\* 协议状态, 见 tn\_topology\_state\_enum\_t \*/  uchar his\_topo\_state; /\* 上次协议状态, 见 tn\_topology\_state\_enum\_t \*/  uchar shape; /\* 拓扑形状, 见 tn\_shape\_enum\_t \*/  uchar logic\_shape; /\* 逻辑拓扑形状, 见 tn\_shape\_enum\_t \*/  uchar active; /\* 协议是否激活, 见 tn\_state\_enum\_t \*/  uchar stale\_mode; /\* stale模式, 见 tn\_stale\_mode\_enum\_t \*/  uchar reserved[2];  } tn\_state\_t, \* tn\_state\_p; |
| **tn\_pre\_cfg\_state\_t** | typedef enum tn\_pre\_cfg\_state\_s {  TN\_PRE\_CFG\_OK = 0, /\* 结点OK \*/  TN\_PRE\_CFG\_CONFLICT, /\* 结点预配置冲突 \*/  TN\_PRE\_CFG\_UNINSTALL, /\* 结点为UNINSTALL \*/  TN\_PRE\_CFG\_UNUSED, /\* 结点为未使用状态 \*/    TN\_PRE\_CFG\_STATE\_MAX /\* 未知状态 \*/  } tn\_pre\_cfg\_state\_t; |
| **tn\_topology\_t** | typedef struct tn\_topology\_s{  tn\_state\_t tn\_state; /\* TN协议状态\*/  uchar tn\_version; /\* TN协议版本\*/  uchar domain\_id; /\* Domain ID \*/  uchar max\_size; /\* max topology size \*/  uchar cur\_size; /\* Current topology size \*/  uchar cur\_free; /\* Current free index \*/  uchar reserved[3];  tn\_node\_p am; /\* am node \*/  tn\_node\_p bm; /\* bm node \*/  tn\_node\_p node\_array[TN\_ARRAY\_SIZE]; /\* 物理拓扑结点 size equels max\_size \*/  tn\_node\_p logic\_node\_array[TN\_ARRAY\_SIZE];/\* 逻辑拓扑结点 \*/  } tn\_topology\_t, \* tn\_topology\_p; |

|  |  |
| --- | --- |
| **软件目录** | **Tn/tn\_link.h** |
| **名称** | **定义** |
| **tn\_vsl\_port\_t** | typedef struct tn\_vsl\_port\_s {  struct list\_head lst; /\* vsl端口列表 \*/  struct timeval ok\_timestamp; /\* 端口检测OK时间戳 \*/  struct timeval up\_timestamp; /\* 端口物理UP时间戳 \*/  ulong bw; /\* VSL端口带宽 \*/  uchar apid; /\* 本端端口所属的VSL-AP \*/  uchar slotid; /\* 本端槽位号 \*/  uchar portid; /\* 本端端口号 \*/  uchar peer\_slotid; /\* 对端槽位号 \*/  uchar peer\_portid; /\* 对端端口号 \*/  uchar state; /\* 端口状态 \*/  uchar pre\_state; /\* 上次端口状态 \*/  uchar next\_state; /\* 下次端口状态 \*/  tn\_timer\_t dt; /\* 发送探测报文定时器 \*/  tn\_timer\_t kat; /\* KeepAlive定时器 \*/  tn\_timer\_t lat; /\* VSL成员链路老化定时器 \*/  tn\_timer\_t dlt; /\* 延迟处理定时器 \*/  tn\_vslp\_pkt\_stat\_t pkt\_stat; /\* 报文统计 \*/  tn\_vslp\_pkt\_stat\_t lat\_pkt\_stat; /\* 报文统计，lat一半时记录 \*/  } tn\_vsl\_port\_t, \*tn\_vsl\_port\_p; |

|  |  |
| --- | --- |
| **软件目录** | **Tn/tn\_rm.h** |
| **名称** | **定义** |
| **tn\_oif\_t** | typedef struct tn\_oif\_s {  uchar vsl\_id; /\* VSU AP口标志\*/  uchar hop; /\* 跳数\*/  uchar reserved[2]; /\* 保留\*/  ulong bw; /\* 带宽\*/  } tn\_oif\_t, \* tn\_oif\_p; |
| **tn\_rib\_t** | typedef struct tn\_rib\_s {  uchar dst\_mid; /\* 目的节点标识\*/  uchar reserved[3]; /\* 保留\*/  tn\_oif\_p active\_oif; /\* 活跃出口\*/  tn\_oif\_p backup\_oif; /\* 备用出口,当前只有一个备用的\*/  tn\_oif\_t oif\_list[2]; /\* 出接口列表\*/  } tn\_rib\_t, \* tn\_rib\_p; |
| **tn\_rib\_array\_t** | typedef struct tn\_rib\_array\_s {  uchar max\_rib\_size;  uchar cur\_rib\_size;  uchar reserved[2];  tn\_rib\_t ribs[0];  } tn\_rib\_array\_t, \* tn\_rib\_array\_p; |
| **tn\_mrib\_t** | typedef struct tn\_mrib\_s {  uchar src\_mid; /\* 源节点标识\*/  uchar iif\_id; /\* 入口标识\*/  uchar oif\_id[TN\_MAX\_IF\_NUM]; /\* 出口列表标识\*/  } tn\_mrib\_t, \* tn\_mrib\_p; |
| **tn\_mrib\_array\_t** | typedef struct tn\_mrib\_array\_s {  uchar max\_mrib\_size;  uchar cur\_mrib\_size;  uchar reserved[2];  tn\_mrib\_t mribs[0];  } tn\_mrib\_array\_t, \* tn\_mrib\_array\_p; |

## 日志与审计

### 事件日志正文

事件日志记录管理和行为事件。管理事件包括对系统配置的修改，管理员和用户的登录和登出。行为事件包括系统活动、例如 VPN 通道的建立和防火墙认证事件。

每个事件日志消息记录了事件发生的日期和事件，以及一个对事件的描述。对于到tnos的以管理为目的的连接和对配置的修改，事件日志消息还包含了管理者电脑的 IP 地址。

一个事件日志正文的例子包含了如下信息：

**user=<用户名> ui=<用户管理设备的方式> msg="<日志消息> "**

* **user**：如果是用户行为触发的消息，指明用户名。如果是系统自身产生的消息，该值为空
* **ui**：如果是用户行为触发的消息，指明用户登录设备的方式。如果是系统自身产生的消息，该值为空

例如：

2018-03-02 07:36:00 type=EVENT subtype=USER level=WARNING: user=admin ui=ssh(10.10.10.1) msg="User admin logout from ssh(10.10.10.1),time out"

2018-03-02 07:36:00 type=EVENT subtype=CONFIG level=INFO: user=admin ui=ssh(10.10.10.1) msg="Execute command "mange ping" success "

2018-03-02 07:36:00 type=EVENT subtype=CONFIG level= WARNING: user=’’ ui=’’ msg="Interface ethernet0 change to physical up"

**使用tnsyslog\_event()接口打印日志**

### 反病毒日志正文

每个病毒日志消息记录了检测到病毒的日期和时间，病毒的类型，以及被感染的通讯的源 IP 地址和目的 IP 地址。

一个病毒日志正文的例子包含如下信息：

**src=<IP地址> dst=<IP地址> src\_int=<源\_接口\_名称> dst\_int=<目的\_接口\_ 名称 > service= {http | smtp | pop3 | imap | ftp} status={blocked | passthrough} from=<发件人电子邮件地址> to=<收件人电子邮件地址 > msg="< 消息字符串 >"**

**使用tnsyslog\_antivirus()接口打印日志**

### 网页过滤日志正文

每个网页过滤日志消息记录了内容被阻塞、URL 被阻塞或者 URL 被从阻塞中排除的日期和时间，以及 HTTP 通讯的源 IP 地址和目的 IP 地址。一个网页过滤日志正文的例子包含如下信息：

**src=<IP地址> dst=<IP地址> src\_int=<源\_接口\_名称> dst\_int=<目的\_接口 \_ 名称 > service=http status={blocked | passthrough} dstname=<主机名称 > arg=<URL 的路径名 > msg="< 消息字符串 >"**

**使用tnsyslog\_webfilter()接口打印日志**

### IPS日志正文

IPS 日志记录了被 tnos 入侵防御系统检测到的攻击。每个 IPS 日志消息记录了攻击发生的日期和时间，攻击的类型以及这次攻击的源 IP 地址和目的 IP 地址。

一个 IPS日志正文的例子包括如下内容：

**attack\_id=<攻击ID> src=<攻击的源IP> dst=<攻击的目的IP> src\_port=<攻击的源port> dst\_port=<攻击的目的port> status=detected proto=<协议号> service=<服务类型> msg="<消息字段> "**

**使用tnsyslog\_ips()接口打印日志**

### traffic日志正文

traffic日志正文记录了到达和通过 tnos接口的全部流量。

一个traffic日志正文包含如下内容：

**duration=<持续时间\_数字> policyid=<策略ID\_数字> proto=<协议\_数字> service=< 服务 \_ 字符串 > status=<{ 拒绝 | 接受 }> src=< 源 \_IP 地址 > srcname=<源名称\_字符串> dst=<目的\_IP地址> dstname=<目的名称\_字符串> src\_int=<源接口名称\_字符串> dst\_int=<目的接口名称\_字符串> sent=<发送\_数字 > rcvd=< 接收 \_ 数字 > send\_pkt=< 发送数据包 \_ 数字 > rcd\_pkt=< 接收数据包 \_ 数字 > src\_port=< 源端口号 \_ 数字 > dst\_port=< 目的端口号 \_ 数字 > vpn=<VPN通道名称\_字符串> tran\_ip<转换IP地址> tran\_port=<转换端口号\_数字 > dir\_disp=< 源 | 重放 > tran\_disp=<noop|snat|dnat>**

* duration：持续时间以秒为单位的会话或者数据包持续的时间。
* Policyid：策略 ID 这个通讯类型使用的策略。
* Proto：协 议 在 IP 报头中的协议编号。
* Service：服务
  + 对于 TCP 和 UDP 通讯，服务是 "dst\_port\_number/tcp" 或者"dst\_port\_number/udp"。
  + 对于 ICMP，是 "icmp"。
  + 对于其他通讯，应该是 "unknown"。

如果日志设置或者通讯过滤的显示设置为服务名：

* + 对于 TCP 通讯，服务将被解析为 "echo", "daytime", "netstat", "qotd", "ftp-data", "ftp", "ssh", "telnet", "smtp", "time", "nameserver", "dns", "gopher","finger", "http", "hostnames", "pop3", "imap", "imap3", 和 "https"。
  + 对于 UDP 通讯，服务将被解析为 "echo", "daytime", "ssh", "time", "dns","tftp", "gopher", "http", "pop3", "imap", "imap3", "https".
  + 如果它不能被解析为上面所列出的服务名，则将显示为"dst\_port\_number/tcp" 或者 "dst\_port\_number/udp"。
  + 对于其他通讯，将显示为 "other"。
* Status：状态 拒绝或接受。
* Src：源 通讯发送者的源 IP 地址。
* Srcname：源名 发送者的 IP 地址或者名称。
* Dst：目的 接收者的 IP 地址。
* Dstname：目的名接收者的 IP 地址或名称。
* src\_int：源接口， 通讯进入的接口，对于从防火墙向外发出的通讯它将是 “未知”。
* dst\_int：目的接口， 通讯发出的接口，对于进入防火墙的向内的通讯它将是 “未知”。
* Sent：发送， 以字节为单位统计的通讯发送的数据总数。
* Rcvd：接收， 以字节为单位统计的通讯接收的数据总数。
* send\_pkt：发送，数据 包 当前会话中发送的数据包的总数。仅限于会话日志。
* rcd\_pkt：接收 收 - - 数据 包 当前会话中接收的数据包的总数。仅限于会话日志。
* src\_port：源端 口， TCP 或 UDP 通讯的源端口，对于其他类型的通讯，它将为 “0”。
* dst\_port：目的 端 口 TCP 或 UDP 通讯的目的端口，对于其他类型的通讯，它将为 “0”。
* vpn： 如果通讯使用了 VPN 通道，则为通讯所使用的通道名。如果通讯不是来自于VPN 通道，则在日志中没有此项信息。
* tran\_ip：转换 换 IP 在 NAT 模式中转换的 IP ；对于透明模式，它将是 “0.0.0.0”。仅限于数据包日志。
* tran\_port：转 换端 口 NAT 模式下转换的端口；对于透明模式，它将是 “0”。仅限于数据包日志。
* dir\_disp：原始数据包或者应答数据包。仅限于数据包日志。
* tran\_disp：NAT 转换的源数据包或者 NAT 转换的目的数据包。仅限于数据包日志。

**使用tnsyslog\_traffic()接口打印日志**

## 流量统计

|  |  |
| --- | --- |
| **名称** | **定义** |
| tn\_ext\_mid\_enum\_t | typedef enum tn\_reg\_mid\_enum\_s {  TN\_EXT\_MID\_SSP = 0,  TN\_EXT\_MID\_GDM,  TN\_EXT\_MID\_CLI,  TN\_EXT\_MID\_DAD,  TN\_EXT\_MID\_MAX  } tn\_mid\_enum\_t; |
| tn\_event\_type\_enum\_t | typedef enum tn\_event\_type\_enum\_s {  TN\_EV\_TOPO\_CVG = 0,  TN\_EV\_ROUTE\_CVG,  TN\_EV\_MULTI\_ROUTE\_CVG,  TN\_EV\_GO\_STALE,  TN\_EV\_SHAPE\_CVT,  TN\_EV\_MAX  } tn\_event\_type\_enum\_t; |
| tn\_shape\_enum\_t | typedef enum tn\_shape\_enum\_s {  TN\_SHAPE\_CHAIN = 0,  TN\_SHAPE\_RING,  TN\_SHAPE\_MAX  } tn\_shape\_enum\_t; |
| tn\_role\_enum\_t | typedef enum tn\_role\_enum\_s {  TN\_ROLE\_CANDIDATE = 0,  TN\_ROLE\_BM,  TN\_ROLE\_AM,  TN\_ROLE\_TYPE\_MAX  } tn\_role\_enum\_t; |
| tn\_pre\_cfg\_state\_t | typedef enum tn\_pre\_cfg\_state\_s {  TN\_PRE\_CFG\_OK = 0, /\* 结点OK \*/  TN\_PRE\_CFG\_CONFLICT, /\* 结点预配置冲突 \*/  TN\_PRE\_CFG\_UNINSTALL, /\* 结点为UNINSTALL \*/  TN\_PRE\_CFG\_UNUSED, /\* 结点为未使用状态 \*/    TN\_PRE\_CFG\_STATE\_MAX /\* 未知状态 \*/  } tn\_pre\_cfg\_state\_t; |
| tn\_topo\_desc\_t | typedef struct tn\_dev\_desc\_s {  uchar valid;  uchar role;  uchar mac[6];  uchar dev\_id;  uchar slot\_id;  uchar cpu\_id;  uchar dev\_type; /\* 比如BOX, CHASSIS等 \*/  ulong type\_id; /\* 比如8606, 8610, 8614等 \*/  ulong pre\_cfg\_state; /\* 结点状态，见GDM定义 \*/  } tn\_dev\_desc\_t, \* tn\_dev\_desc\_p;  typedef struct tn\_topo\_desc\_s {  uchar shape; /\* 拓扑形状\*/  uchar role; /\* 本机角色\*/  uchar local\_id; /\* Local DEV\_ID \*/  uchar dev\_num; /\* DEV NUM \*/  uchar am\_id; /\* AM DEV\_ID \*/  uchar bm\_id; /\* BM DEV\_ID \*/  uchar reserved[2];  } tn\_topo\_desc\_t, \* tn\_topo\_desc\_p; |
| tn\_set\_fib\_t | typedef struct tn\_set\_fib\_s {  uchar owner\_mid; /\* 表项所在的设备\*/  uchar dsc\_id; /\* 目的设备的switch id \*/  uchar oif\_id; /\* 到达目的设备的出口ID \*/  uchar reserved; /\* 保留\*/  } tn\_set\_fib\_t, \* tn\_set\_fib\_p; |
| tn\_fib\_msg\_t | typedef struct tn\_fib\_msg\_s {  uchar version; /\* 版本号 \*/  uchar fib\_num; /\* 需要操作的fib数目 \*/  ushort length; /\* 消息长度 \*/  uchar am\_id; /\* AM 的设备switch id \*/  uchar bm\_id; /\* BM 的设备switch id \*/  uchar reserved[2]; /\* 保留字段\*/  tn\_set\_fib\_t set\_fibs[0];/\* fib操作信息，见tn\_set\_fib\_t \*/  } tn\_fib\_msg\_t, \* tn\_fib\_msg\_p; |
| tn\_set\_mfib\_t | typedef struct tn\_set\_mfib\_s {  uchar owner\_mid; /\* 表项所在的设备\*/  uchar src\_id; /\* 组播源设备的switch id \*/  uchar iif\_id; /\* 入口ID \*/  uchar reserved\_1; /\* 保留\*/  uchar oif\_id[TN\_MAX\_MFIB\_OIF\_NUM];/\* 到达目的设备的出口ID \*/  uchar reserved\_2[2]; /\* 保留\*/  } tn\_set\_mfib\_t, \* tn\_set\_mfib\_p; |
| tn\_mfib\_msg\_t | typedef struct tn\_mfib\_msg\_s {  uchar version; /\* 版本号 \*/  uchar mfib\_num; /\* 需要操作的fib数目 \*/  ushort length; /\* 消息长度 \*/  uchar am\_id; /\* AM 的设备switch id \*/  uchar bm\_id; /\* BM 的设备switch id \*/  uchar reserved[2]; /\* 保留字段\*/  tn\_set\_mfib\_t set\_mfibs[0];/\* mfib操作信息，见tn\_set\_mfib\_t \*/  } tn\_mfib\_msg\_t, \* tn\_mfib\_msg\_p; |
| tn\_topo\_notif\_fun\_t | 注册拓扑事件通知用的回调函数  原型：typedef int (\*tn\_topo\_notif\_fun\_t)(ulong event\_type, void \* msg);  参数event\_type：见tn\_event\_type\_enum\_t；  参数msg：  event\_type等于TN\_EV\_TOPO\_CVG时，msg对应结构为tn\_topo\_desc\_t；  event\_type等于TN\_EV\_ROUTE\_CVG时，msg对应结构为tn\_fib\_msg\_t；  event\_type等于TN\_EV\_MULTI\_ROUTE\_CVG时，msg对应结构为tn\_mfib\_msg\_t；  event\_type等于TN\_EV\_GO\_STALE时，msg为空；  event\_type等于TN\_EV\_SHAPE\_CVT时，msg对应结构为uchar；  返回值：0成功；-1失败。 |
| vsu\_exit\_type\_t | /\* 节点退出检查确认结果 \*/  typedef enum vsu\_exit\_type\_t {  VSU\_EXIT\_OK = 0, /\* 可以直接退出，不需要用户确认的 \*/  VSU\_EXIT\_ERR, /\* 错误情况: 比如参数错误\*/  VSU\_EXIT\_REQ, /\* 对拓扑有影响，需要用户确认的退出 \*/    VSU\_EXIT\_TYPE\_MAX,  } vsu\_exit\_type\_t; |
| vsu\_show\_type\_t | typedef enum vsu\_show\_type\_s {  VSU\_SHOW\_VSL\_AP = 0,  VSU\_SHOW\_VSL\_PORT,    VSU\_SHOW\_TYPE\_MAX  } vsu\_show\_type\_t; |

## 安全管理

|  |  |
| --- | --- |
| **软件目录** | **Tn/tn\_link.h** |
| **名称** | **定义** |
| **tn\_vsl\_port\_t** | typedef struct tn\_vsl\_port\_s {  struct list\_head lst;  struct timeval ok\_timestamp;  struct timeval up\_timestamp;  ulong bw;  uchar apid; /  uchar slotid;  uchar portid;  uchar peer\_slotid;  uchar peer\_portid;  uchar state;  uchar pre\_state;  uchar next\_state;  tn\_timer\_t dt;  tn\_timer\_t kat;  tn\_timer\_t lat;  tn\_timer\_t dlt;  tn\_vslp\_pkt\_stat\_t pkt\_stat;  tn\_vslp\_pkt\_stat\_t lat\_pkt\_stat;  } tn\_vsl\_port\_t, \*tn\_vsl\_port\_p; |

## 高可用性

{

"max\_depth": num, #

"max\_features": num, # max\_features：

"min\_samples\_split": num,

"max\_leaf\_nodes": num,

"splitter": "", # best or random

}

{

"n\_estimators": num,

"max\_depth": num, # 树的深度

"max\_features": num, # max\_features：每个节点处，算法随机选择特征的一个子集，并对其中一个特征寻找最佳测试。max\_features用于控制这个子集的特征个数。

"min\_samples\_split": num, # 分裂一个内部节点所需的最小样本数, 默认值为2

"max\_leaf\_nodes": num, # 最大叶节点数量，当max\_depth设置时，该值无效

}

{

"C": float, # 值越小，说明模型非常受限，每个数据点影响范围有限。C值越大，点对模型的影响越大，使决策边界发送弯曲。

"gamma": float # gamma越小说明高斯核的半径越大，许多点被看作比较靠近。小的gamma值表示决策边界变化很慢，生成复杂度较低的模型，反之生成更为复杂的模型

}

{

"max\_depth": num, # 树的深度

"max\_features": num, # max\_features：每个节点处，算法随机选择特征的一个子集，并对其中一个特征寻找最佳测试。max\_features用于控制这个子集的特征个数。

"min\_samples\_split": num, # 分裂一个内部节点所需的最小样本数, 默认值为2

"max\_leaf\_nodes": num, # 最大叶节点数量，当max\_depth设置时，该值无效

"splitter": "", # best or random

"criterion": "", # gini or entropy

}

{

"n\_estimators": num, # 构造树的个数

"max\_depth": num, # 树的深度

"max\_features": num, # max\_features：每个节点处，算法随机选择特征的一个子集，并对其中一个特征寻找最佳测试。max\_features用于控制这个子集的特征个数。

"min\_samples\_split": num, # 分裂一个内部节点所需的最小样本数, 默认值为2

"max\_leaf\_nodes": num, # 最大叶节点数量，当max\_depth设置时，该值无效

}

{

"alpha": float， # alpha越大，平滑化越强，模型复杂度越低。

"binarize": float, # 样本特征的二值化(映射到布尔值)的阈值。如果没有，则假定输入已经由二进制向量组成。

}

{

"n\_clusters": integer, # 聚类数量--------必填

}

{

"eps": 邻域半径,eps（默认0.5）设置非常小意味着没有点是核心样本，所有点是噪声。eps非常大，可能会导致所有点形成单簇

}

{

"n\_clusters": integer, # 聚类数量--------必填

}

# 接口

## 网络基本功能

**【功能说明】**

接口创建初始化

**【声明】**

void interface\_ini\_devid(DTRID\_T devid)

**【参数】**

DTRID\_T devid 本卡设备id

**【算法及内部实现】**

|  |
| --- |
| devid赋值给本设备id的标识 |

**【返回值】**

无

**【使用说明】**

传入devid后interface模块初始化完成

**【注意】**

无

**【功能说明】**

二层网络初始化

**【声明】**

static L2\_init\_actgm\_start(fsm\_t \*fsm, uint para)

**【参数】**

fsm\_t \*fsm 状态机

uint para 参数

**【算法及内部实现】**

|  |
| --- |
| 发出自身是init的事件，开始推动状态机 |

**【返回值】**

根据执行结果，返回DONE/SLEEP/PEND

**【使用说明】**

无

**【注意】**

无。

**【功能说明】**

三层网络初始化

**【声明】**

static L3\_init\_start(fsm\_t \*fsm, uint para)

**【参数】**

fsm\_t \*fsm 状态机

uint para 参数

**【算法及内部实现】**

|  |
| --- |
| 三层初始化 |

**【返回值】**

根据执行结果，返回DONE/SLEEP/PEND

**【使用说明】**

无

**【注意】**

。

## 包过滤和状态检测接口

说明：创建规则

url:/ml/api/v1.0/ml/create

method: post

param:

{

"app": "", # 学习任务所属的app

"name": "", # 任务名称

"author": "", # 作者

"type": "", # 任务类型，目前包括： predict\_numeric/predict\_category/cluster

"description": "", # 任务描述

"data": {

"index": {

"source\_type": "", # log or report or file

"name": "", # 如果是source\_type是log，name有效，表示分类日志的名字. 如果source\_type是report，name表示report名字； 如果source\_type=file，name表示文件名

"indices": [], # 如果是source\_type是log, 表示数据源的index列表（非必填项）， 如果source\_type是report， 表示result的pivot\_table的index；如果source\_type=file，表示文件存入es的index

},

"start": "", # 数据时间范围utc时间

"end": "", # 数据时间范围UTC时间

"search": {}, # es 搜索条件

"fields": [{"type": "", "name": ""}], # 该模型需要的字段

},

}

result: {"id": xxxxx}, 200 # 随机ID，唯一标识训练任务

说明：修改操作

url:/ml/api/v1.0/ml/modify

method: post

param:{

"task\_id": "xxxx",

"name": "", # 任务名称

"author": "", # 作者

"description": "", # 任务描述

"model\_name": "", # 训练后模型的名字 predict\_numeric/predict\_category使用

"model\_description": "", # 模型描述 predict\_numeric/predict\_category使用

}

说明：删除操作

url:/ml/api/v1.0/ml/delete?id=xxx

method: delete

说明：用户获取一个学习任务信息

url:/ml/api/v1.0/ml/get/ml\_task?id=xxx

method: get

result：

{

"task\_id": "", # 任务id，后端分配，全局唯一

"app": "", # 学习任务所属的app

"name": "", # 任务名称

"author": "", # 作者

"type": "", # 任务类型，目前包括： predict\_numeric/predict\_category/cluster

"description": "", # 任务描述

"create\_time": "",

"update\_time": "",

"model\_id": "", # 模型id，后端分配 predict\_numeric/predict\_category使用

"model\_name": "", # 训练后模型的名字 predict\_numeric/predict\_category使用

"model\_description": "", # 模型描述 predict\_numeric/predict\_category使用

"current\_id": num, # 当前使用的训练id

"train\_count": num, # 一个学习任务下，训练的次数

"trains": [train\_task1, train\_task2], # train\_task

}

说明：获取任务列表

url:/ml/api/v1.0/ml/get/ml\_tasks?type=xxx

method: get

result：

{

"entries":[{}],

"page": num,

"page\_size": num,

"reason": [],

"success": ture,

"title": [],

"total\_num": num,

"total\_page": num

}

entries：

{

"task\_id": "", # 任务id，后端分配，全局唯一

"app": "", # 学习任务所属的app

"name": "", # 任务名称

"author": "", # 作者

"type": "", # 任务类型，目前包括： predict\_numeric/predict\_category/cluster

"description": "", # 任务描述

"create\_time": "",

"update\_time": "",

"model\_id": "", # 模型id，后端分配 predict\_numeric/predict\_category使用

"model\_name": "", # 训练后模型的名字 predict\_numeric/predict\_category使用

"model\_description": "", # 模型描述 predict\_numeric/predict\_category使用

"current\_id": num, # 当前使用的训练id

"train\_count": num, # 一个学习任务下，训练的次数

"trains": [train\_task\_id, train\_task\_id], # train\_task id

}

说明：执行

url:/ml/api/v1.0/ml/train

method: post

param:

{

"task\_id": "",

"type": "", # 任务类型，目前包括： predict\_numeric/predict\_categorical/cluster

"datas": {

"index": {

"source\_type": "", # log or report

"name": "", # 如果是source\_type是log，name有效，表示分类日志的名字. 如果source\_type是report，name表示report result的pivot\_table的名字

}

"start": "", # 数据时间范围utc时间

"end": "", # 数据时间范围UTC时间

"search": {}, # es 搜索条件

"fields": [{"type": "", "name": ""}], # 该模型需要的字段

},

"preprocessing": [ # 数据预处理步骤，多个步骤

{

"method": "", # 数据预处理方法

"fields": [{"type": "", "name": ""}], # 预处理字段

"params": {"param": "value"}, # 预处理方法的参数

},

],

"algorithm": {

"name": "", # 算法的名字：

# 分类预测： LogisticRegression/SVM/RandomForestClassifier/GaussianNB/BernoulliNB/DecisionTreeClassifier

# 值预测：LogisticRegression/RandomForestClassifier/Lasso/Ridge/DecisionTreeClassifier

# 聚类：K-means/DBSCAN/Agglomerative

"params": { # 算法参数

"param": "value", # 多个参数

}

},

"predict\_field": {"type": "", "name": ""}, # 待预测的字段, predict\_numeric/predict\_categorical使用

"fields": [{"type": "", "name": ""}], # 使用的字段

"training\_scale": num, # 用于训练的数据百分比，默认70%， predict\_numeric/predict\_categorical使用

}

result: {"msg": "Success"， "id": xxx}, 200 或者

result:{"msg", "error message", "id": None}, 400

说明：执行

url:/ml/api/v1.0/ml/cluster

method: post

param:

{

"task\_id": "",

"type": "", # 任务类型，目前包括： predict\_numeric/predict\_categorical/cluster

"datas": {

"index": {

"source\_type": "", # log or report

"name": "", # 如果是source\_type是log，name有效，表示分类日志的名字. 如果source\_type是report，name表示report result的pivot\_table的名字

}

"start": "", # 数据时间范围utc时间

"end": "", # 数据时间范围UTC时间

"search": {}, # es 搜索条件

"fields": [{"type": "", "name": ""}], # 该模型需要的字段

},

"preprocessing": [ # 数据预处理步骤，多个步骤

{

"method": "", # 数据预处理方法

"fields": [{"type": "", "name": ""}], # 预处理字段

"params": {"param": "value"}, # 预处理方法的参数

},

],

"algorithm": {

"name": "", # 算法的名字：

# 分类预测： LogisticRegression/SVM/RandomForestClassifier/GaussianNB/BernoulliNB/DecisionTreeClassifier

# 值预测：LogisticRegression/RandomForestClassifier/Lasso/Ridge/DecisionTreeClassifier

# 聚类：K-means/DBSCAN/Agglomerative

"params": { # 算法参数

"param": "value", # 多个参数

}

},

"fields": [{"type": "", "name": ""}], # 使用的字段

}

result: {"msg": "Success"， "id": xxx}, 200 或者

result:{"msg", "error message", "id": None}, 400

说明：获取结果

url:/ml/api/v1.0/ml/get/result?id=xxxx&page=x&page\_size=xx

method: get

result: 显示前10000条，数据格式和返回的pivot table一致

{

"datas":[{}],

"page": num,

"page\_size": num,

"reason": [],

"success": ture,

"title": [],

"total\_num": num,

"total\_page": num

}

说明：对比

url：/ml/api/v1.0/ml/get/contrast\_line?id=xxxx

method: get

result: 显示前10000条，数据格式和返回的report曲线数据一致

说明：残差

url：/ml/api/v1.0/ml/get/residual\_line?id=xxxx

method: get

result: 显示前10000条，数据格式和返回的report曲线数据一致

说明：点击保存

url:/ml/api/v1.0/ml/save

method: post

param:

{

"task\_id": "", # 学习任务id，后端分配，全局唯一

"train\_id"" "", # 训练任务id

"model\_name": "", # 训练后模型的名字 predict\_numeric/predict\_category使用

"model\_description": "", # 模型描述 predict\_numeric/predict\_category使用

}

说明：IPS模型获取

url:/ml/api/v1.0/ml/get/models

method: get

result: [

{

"type": "", # 模型类型，目前包括： predict\_numeric/predict\_category

"predict\_field": {"type": "", "name": ""},

"fields": [{"type": "", "name": ""}], # 该模型需要的字段

"model\_id": "", # 模型id，后端分配

"model\_name": "", # 训练后模型的名字 predict\_numeric/predict\_category使用

"model\_description": "", # 模型描述 predict\_numeric/predict\_category使用

}

]

## NAT

说明：获取原始数据

url:/ml/api/v1.0/ml/get/raw?page=x&page\_size=xx

method: post

"param": {

"index": {

"source\_type": "", # log or report

"name": "", # 如果是source\_type是log，name有效，表示分类日志的名字. 如果source\_type是report，name表示report result的pivot\_table的名字

}

"start": "", # 数据时间范围utc时间

"end": "", # 数据时间范围UTC时间

"search": {}, # es 搜索条件

"fields": [{"type": "", "name": ""}], # 该模型需要的字段

},

result: 显示前10000条

{

"datas":[{}],

"page": num,

"page\_size": num,

"reason": [],

"success": ture,

"title": [],

"total\_num": num,

"total\_page": num

}

说明：IDS获取

url:/ml/api/v1.0/ml/predict

param： {

"train\_id": "",

"data": [{"field1": "value1", "field2": "value2", "field3": "value3"} ]

}

result: {

"entries":[],

"reason": [],

"success": True,

"total\_num": 0,

}

说明：获取列表

url:/ml/api/v1.0/ml/get/ml\_trains?id=xxxx

id为学习任务

result: {

"entries":[],

"reason": [],

"success": True,

"total\_num": 0,

}

说明：获取历史列表

url:/ml/api/v1.0/ml/get/ml\_train?id=xxxx # 训练id

result: 训练任务数据

## 用户管理

说明：创建用

url：/ml/api/v1.0/ml/create/?id=xxxx

method: POST

result:

{

"name": num,

"description": num,

"id": num,

"role": num，

}

说明：创建角色

url:/ml/api/v1.0/ml/get/cluster/scatter?id=xxxx

method: post

param： ["field0", "field1"]

result:

{

"axis": [{"x": "field0", "y": "field1"}]

"data": [

{

"field0": value0,

"field1": value1,

"field2": value2,

"field3": value3,

"field4": value4,

}

],

"msg": "Success"

}

param： ["field0", "field1", "field2"]

result:

{

"axis": [{"x": "field0", "y": "field1", "z": "field2"}]

"data": [

{

"field0": value0,

"field1": value1,

"field2": value2,

"field3": value3,

"field4": value4,

}

],

"msg": "Success"

}

param： ["field0", "field1", "field2", "field3"]

result:

{

"axis": [

{"x": "field0", "y": "field1"},

{"x": "field0", "y": "field2"},

{"x": "field0", "y": "field3"},

{"x": "field1", "y": "field2"},

{"x": "field1", "y": "field3"},

{"x": "field2", "y": "field3"},

]

"data": [

{

"field0": value0,

"field1": value1,

"field2": value2,

"field3": value3,

"field4": value4,

}

],

"msg": "Success"

}

说明：删除

url:/ml/api/v1.0/ml/delete/user?id=xxxx

method: delete

## 流量控制

**【功能说明】**

基本通信功能就绪

**【声明】**

static fsm\_act\_stat\_t sripc\_vsu\_ipc\_comm(fsm\_t \*fsm, uint para)

**【参数】**

fsm\_t \*fsm 状态机

uint para 参数

**【算法及内部实现】**

|  |
| --- |
| 更新主控id  特殊处理的连接id开始和主控建立连接  在代理设备控制块中的链接，迁移到真实的设备控制块上  发出框架OK事件，推动状态机 |

**【返回值】**

根据执行结果，返回DONE/SLEEP/PEND

**【使用说明】**

无

**【注意】**

无

**【功能说明】**

通告应用模块，IPC依赖的全部基础模块已READY

**【声明】**

static fsm\_act\_stat\_t sripc\_vsu\_act\_ipc\_ok(fsm\_t \*fsm, uint para)

**【参数】**

fsm\_t \*fsm 状态机

uint para 参数

**【算法及内部实现】**

|  |
| --- |
| 发出框架OK事件，推动状态机 |

**【返回值】**

根据执行结果，返回DONE/SLEEP/PEND

**【使用说明】**

无

**【注意】**

无

**【功能说明】**

流量查询

**【声明】**

static int sripc\_vsu\_dev\_proc\_bcmsg(DTRID\_T devid, uint dev\_type)

**【参数】**

DTRID\_T devid 源报文设备id

uint dev\_type 参数

**【算法及内部实现】**

|  |
| --- |
| GM收到报文后，创建线卡设备id，并回复相应的线卡 |

**【返回值】**

正确返回OK，否则ERROR

**【使用说明】**

无

**【注意】**

无

**【功能说明】**

实时统计

**【声明】**

static int sripc\_vsu\_dev\_proc\_iam\_mcm(DTRID\_T devid)

**【参数】**

DTRID\_T devid GM设备id

**【算法及内部实现】**

|  |
| --- |
| 线卡端创建以主控devid命名的设备控制块  发出主控回复事件，推动状态机 |

**【返回值】**

正确返回OK，否则ERROR

**【使用说明】**

无

**【注意】**

无

## IP/MAC绑定

1秒定时器扫描

【原型】：static void tn\_scan\_timer\_one\_sec()

【算法】：

1. 调用tn\_check\_hft检查并处理HFT超时；
2. 调用tn\_check\_kat检查并处理KAT超时。

HFT超时检查处理

【原型】：static void tn\_check\_hft()  
【算法】：

1. 如果HFT超时，发送Hello报文，如果是AM且拓扑状态不是Convergent，还要发送Convergence报文；
2. 调用tn\_check\_met检查并处理MET超时；
3. 调用tn\_check\_ct检查并处理CT超时。

MET超时检查处理

【原型】：static void tn\_check\_met()

【算法】：

1. 如果MET没有超时，退出；
2. 调用tn\_elect\_am选举出AM。

CT超时检查处理

【原型】：static void tn\_check\_ct()

【算法】：

1. 如果CT没有超时，退出；
2. 裁剪未收敛的节点，然后进行拓扑收敛。

KAT超时检查处理

【原型】：static void tn\_check\_hft()  
【算法】：

1. 如果KAT超时，发送Keep Alive报文；
2. 调用tn\_check\_lat检查并处理LAT超时。

LAT超时处理

【原型】：void tn\_check\_lat()

【算法】：

1. 遍历本节点的所有VSU AP口；
2. 如果LAT超时，发送Path Prune报文，然后进行链路剪枝。

链路检测kat超时处理

【原型】：void tn\_check\_ld\_kat()

【算法】：

1. 检查VSU AP是否UP，如果不为UP，则处理结束，否则继续；
2. 遍历VSL端口列表，如果端口状态不为OK，继续遍历，否则继续；
3. 如果发送ka报文倒计时为0，则发送ka，否则减一；
4. 检查是否有收到ka，如果有则设置接收ka报文倒计时为初始值；否则接收ka报文倒计时减一；

检查接收ka报文倒计时是否为0，如果是则设置端口状态为检测状态，同时，更新带宽和VSU AP状态，通知GDM，调用tn\_link\_change，进行链路变化处理。

## VPN

### Path Prune

【原型】：int tn\_path\_prune(uchar \* mac, uchar prune\_if)

【算法】：

1. 调用tn\_get\_node查找对应节点node；
2. 根据prune\_if，将node的对应节点状态置为Down，并修改其直连邻居的链路状态；
3. 如果node的所有链路状态都为down，则从拓扑中删除node；
4. 如果原来的拓扑形态为线形拓扑，删除受path\_prune影响的所有node；
5. 调用tn\_route\_convergence进行路由收敛；
6. 如果AM被剪掉，调用tn\_bm\_to\_am进行BM->AM处理；
7. 如果BM被剪掉，掉用tn\_candidate\_to\_bm进行Candidate->BM处理；
8. 如果AM和BM都被剪掉，调用tn\_topology\_stale进行失效处理。

### Path Graft

【原型】：void tn\_path\_graft(tn\_node\_p loca\_node, uchar oif, uchar \* peer\_mac, uchar peer\_iif, ulong bw)

【算法】：

1. 根据peer\_mac，调用tn\_get\_node查找对应节点peer\_node。
2. 连接local\_node和peer\_node的接口。
3. 如果拓扑形状为线形，重新计算拓扑形状，如果拓扑形状还是线形则返回。
4. 根据拓扑状态进行相关处理：
   1. 如果是Building状态，且所有节点都是Building状态，则触发AM选举。
   2. 如果是Convergent状态，通知形状变化，路由重新收敛。

### Topology Converge

【原型】：void tn\_topology\_converge()

【算法】：

1. 拓扑状态进入Convergent状态。
2. 计算拓扑形状。
3. 路由收敛。
4. 通告拓扑收敛。

### Topology Expand

【原型】：void tn\_topology\_expand(tn\_node\_p new\_node)

【算法】：

1. 设置所有点拓扑状态为Expanding状态。
2. 设置拓扑状态为Expanding状态。
3. 激活并刷新CT定时器。
4. 如果是AM：
   1. 如果拓扑成员数量只有2个，设置new\_node为BM。
   2. 为new\_node分配MID。
   3. 发布Convergence报文.

### Topology Shrink

【原型】：void tn\_topology\_shrink(uchar pre\_size)

【算法】：

1. 如果AM和BM都为空，调用tn\_topology\_stale进入stale处理，返回。
2. 如果AM为空
   1. 如果拓扑大小不足pre\_size的一半，调用tn\_topology\_stale进入stale处理，返回。
   2. 如果上一次拓扑状态为Shrinking而当前状态为Expanding，调用tn\_topology\_stale进入stale处理，返回。
   3. 调用tn\_bm\_to\_am处理BM->AM，返回。
3. 如果BM为空，调用tn\_candidate\_to\_bm处理BM->AM，返回。
4. 否则：
5. 重新路由收敛。
6. 通告拓扑收敛。

### Topology Stale

【原型】：void tn\_topology\_stale(ulong stale\_mode, uchar \* reason)

【算法】：

1. 设置所有节点状态为Stale。
2. 设置拓扑状态为Stale。
3. 取消CT定时器。
4. 通告拓扑收敛事件。

如果stale\_mode为TN\_STALE\_MODE\_PRUNE\_NBR，清空所有邻居。

## 日志与审计

所有向外部发送的日志定义如下表：



## 流量统计

路由收敛

【原型】：int tn\_route\_converge ()

【算法】：

1. 重置路由收敛开关（未完成状态）
2. 重置所有节点的单播表和组播表
3. 检查是否满足路由收敛条件，不满足则退出
4. 检查是否满足设置静态路由条件
5. 如果满足设置静态路由条件，触发VSL合并，然后设置静态路由
6. 如果不满足设置静态路由条件，还原VSL合并，然后计算静态路由
7. 设置路由收敛开关（完成状态）

单播路由算法

【原型】：int tn\_calculate\_rm()

【算法】：

1. 扫描所有的节点
2. 计算节点间的跳数
3. 根据跳数及带宽及索引编号，调用tn\_set\_rib设置节点的单播路由表项

设置单播表的算法

【原型】：int tn\_set\_rib(tn\_node\_p owner, uchar dst\_mid, uchar oif\_id, uchar hop, ulong bw)

【算法】：

1. 根据dst\_mid查找是否存在对应的rib表；
2. 如果存在rib表，检查oif\_id对应的oif是否已经存在rib表中，存在则返回，否则创建oif并插入到oif\_list中，并和active\_oif比较hop及bw，如果oif的hop小或者hop相等但是bw大或者hop相等bw相等但是oif\_id较小，则设置oif为active\_oif，原来的active\_oif为backup\_oif；
3. 如果不存在rib表，创建rib表，并设置rib表的属性，创建oif，并设置oif的属性，将oif插入到oif\_list中，设置active\_oif为oif。

组播路由表算法

【原型】：int tn\_calculate\_mrm()

【算法】：

1. 遍历拓扑节点，扫描每个节点的单播表项
2. 对于每个单播表项：
   1. 单播表项的目的地址即为组播表项的源地址；
   2. 单播表项的最优出口即为组播表项的入口；
   3. 如果单播表存在非最优出口，则非最优出口即是组播表项的出口；否则，检查入口的other\_if是否存在peer\_if，如果存在，则other\_if为组播表项的出口src\_mid = rib->dst\_mid;
   4. 调用tn\_set\_mrib添加组播表项。

设置组播表的算法

【原型】：int tn\_set\_mrib(tn\_node\_p owner, uchar src\_mid, uchar iif\_id, uchar oif\_id)

【算法】：

1. 根据src\_mid查找是否存在对应的mrib表；
2. 如果存在mrib表，更新iif\_id和oif\_id；
3. 如果不存在mrib表，创建mrib表，设置src\_mid、iif\_id和oif\_if;
4. 如果iif\_id不为255，调用tn\_prune\_wrong\_iif处理可能的wrong\_iif情况。

WRONG\_IIF算法

【原型】：int tn\_prune\_wrong\_iif(tn\_node\_p owner, tn\_mrib\_p mrib)

【算法】：

1. 查询owner的直连邻居是否存在同源的组播表；

如果存在同组播表，且组播出接口指向Owner，则发生了Wrong IIF，需要将peer\_mrib和mrib的出口剪枝掉。

## 安全管理

**【功能说明】**

主机防御

**【声明】**

static int attack\_vsu\_dev\_proc\_cnreq(attack\_dev\_cnctl\_t \*cn\_msg, DTRID\_T devid)

**【参数】**

attack\_dev\_cnctl\_t \*cn\_msg 请求报文中携带的消息

DTRID\_T devid 线卡设备id

**【算法及内部实现】**

|  |
| --- |
| 滑动窗口序号协商  回复确认信息  设置请求设备的状态 |

**【返回值】**

正确返回OK，否则ERROR

**【使用说明】**

无

**【注意】**

无

**【功能说明】**

攻击防御

**【声明】**

static int attack\_vsu\_dev\_proc\_cnack(attack\_dev\_cnctl\_t \*cn\_msg, DTRID\_T devid)

**【参数】**

attack\_dev\_cnctl\_t \*cn\_msg 确认报文中携带的消息

DTRID\_T devid 主控设备id

**【算法及内部实现】**

|  |
| --- |
| 滑动窗口序号协商  设置请求设备的状态  发出链接间握手事件，推动状态机 |

**【返回值】**

正确返回OK，否则ERROR

**【使用说明】**

无

**【注意】**

无

**【功能说明】**

Arp检测

**【声明】**

static int attack\_vsu\_arp\_proc\_switch(DTRID\_T devid, attack\_dev\_xctl\_t \*dctl\_msg)

**【参数】**

attack\_dev\_xctl\_t \*dctl\_msg 新GM发送的切换消息

DTRID\_T devid 新GM设备id

**【算法及内部实现】**

|  |
| --- |
| 创建新GM设备控制块  所以的卡重新向新主控发起设备间握手请求  断开原主控设备控制块的链接，并且迁移到新主控的设备控制块 |

**【返回值】**

正确返回OK，否则ERROR

**【使用说明】**

无

**【注意】**

无

**【功能说明】**

防御黑白名单

**【声明】**

static attack\_cncb\_t \*

attack\_mbc\_search\_cncb(attack\_mbcb\_t \*mbcb, uchar cn\_attr, DTRID\_T rmt\_did, uint prio, DTRID\_T grpid)

**【参数】**

attack\_mbcb\_t \*mbcb 邮箱指针

uchar cn\_attr 链接属性

DTRID\_T rmt\_did 远端设备id

uint prio 链接优先级

DTRID\_T grpid 组播id

**【算法及内部实现】**

|  |
| --- |
| 从挂载到邮箱链表中的多个链接中，按照参数查找到需要的链接 |

**【返回值】**

查找到返回链接指针，否则返回NULL

**【使用说明】**

无

**【注意】**

无

**【功能说明】**

Web攻击识别

**【声明】**

DTRID\_T attack\_web \_mb\_p2p(attack\_mb\_conn\_para\_t \*reg\_para)

**【参数】**

attack\_mb\_conn\_para\_t \*reg\_para 注册时的参数

**【算法及内部实现】**

|  |
| --- |
| 是web攻击特征。 |

**【返回值】**

成功返回链接id，否则返回0

**【使用说明】**

无

**【注意】**

## 高可靠性

### 处理HA协议报文

【原型】：int HA\_recv\_packet(ulong slotid, ulong portid, void \* pkt, ulong length)

【算法】：

1. 如果pkt为空则直接返回。
2. 根据slotid和portid查找vsu\_ap，如果vsu\_ap\_id大于等于2则返回。
3. 如果length小于sizeof(HA\_packet\_head\_t)或大于等于1024则返回。
4. 如果HA为激活则返回。
5. 如果版本号不一致则返回。
6. 如果报文类型不支持则返回。
7. 如果ntohs(pkt\_head->length不等于length则返回。
8. 如果domain id不一致则返回。
9. 如果prod\_id不一致则返回。
10. 如果报文的source mac为自己的且报文类型不是DMN：如果distinction不同则回复DMN并返回，否则记录收到自己的报文并返回。
11. 如果报文的destination mac是单播且不是自己的则从另外一个VSU口转发并返回。
12. 如果报文的校验和不对则返回。
13. 统计报文接收。
14. 根据报文的类型，调用相应的接口进行处理；

### 处理ACK报文

【原型】：void HA\_recv\_ack(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：

1. 如果ack的类型为Convergence，调用void HA\_recv\_cvg\_ack(HA\_packet\_head\_p pkt\_head, uchar iif)处理；
2. 如果ack的类型为Route Convergence，调用void HA\_recv\_route\_cvg\_ack(HA\_packet\_head\_p pkt\_head, uchar iif)处理。
3. 如果ack的类型为Path Prune，调用void HA\_recv\_path\_prune\_ack(HA\_packet\_head\_p pkt\_head, uchar iif)处理；

【原型】：void HA\_recv\_cvg\_ack(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.1描述。

【原型】：void HA\_recv\_route\_cvg\_ack(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.1描述。

【原型】：void HA\_recv\_path\_prune\_ack(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.1描述。

### 处理Hello报文

【原型】：int HA\_recv\_hello(HA\_packet\_head\_p pkt\_head)

【算法】：参考参考2.8.1.2描述

### 处理Convergence报文

【原型】：void HA\_recv\_cvg(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.3描述

### 处理Route Convergence报文

【原型】：void HA\_recv\_route\_cvg(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.4描述

### 处理Path Prune报文

【原型】：void HA\_recv\_path\_prune(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.5描述

### 处理Self-Path-Prune报文

【原型】：void HA\_recv\_spp(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.6描述

### 处理Merge报文

【原型】：void HA\_recv\_merge(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.7描述

### 处理Restart报文

【原型】：void HA\_recv\_restart(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.8描述

### 处理Leave报文

【原型】：void HA\_recv\_leave(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.9描述

### 处理Stale报文

【原型】：void HA\_recv\_stale(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.10描述

### 处理DMN报文

【原型】：void HA\_recv\_dmn(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.11描述

### 处理Link Detect报文

【原型】：void HA\_recv\_ld(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.12描述

### 处理PORT SYNC报文

【原型】：void HA\_recv\_port\_sync(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.3描述

### 处理DATA报文

【原型】：void HA\_recv\_port\_sync(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.14描述

### 处理B2A报文

【原型】：void HA\_recv\_port\_sync(HA\_packet\_head\_p pkt\_head, uchar iif)

【算法】：参考2.8.1.15描述

# 对应性分析

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 检测项目（GB/T 20281-2015信息安全技术 防火墙安全技术要求和测试评分方法（基本级）） | | | | 备注 |
| 安全功 能检测 | 网络层 控制 | 包过滤 | | 2.2 |
| NAT | | 2.3 |
| 状态检测 | | 2.2 |
| 策略路由 | | 2.1 |
| 动态开放端口 | | 2.1 |
| IP MAC 地址绑定 | | 2.6 |
| 流量会话管理 | 流量统计 | 2.9 |
| 连接数控制 | 2.9 |
| 抗拒绝服务攻击 | | 2.10 |
| 网络扫描防护 | | 2.2 |
| 应用层协议控制 | | | 2.2 |
| 安全运维管理 | 运维管理 | | 2.4 |
| 安全审计 | | 2.8 |
| 安全管理 | 管理接口独立 | 2.10 |
| 安全支撑系统 | 2.10 |
| 异常处理机制 | 2.10 |