基于OLSR的入侵检测系统设计技术方案

# OLSR协议运行过程

在OLSR协议中，路由发现过程主要包括邻居发现、MPR节点集选取、拓扑传播、路由生成几部分。路由发现的主要流程共分为以下几步：

第一步：所有节点周期性广播Hello消息，用于探测一跳邻居节点情况。

第二步：节点接收到Hello消息后，更新一跳邻居节点和两跳邻居节点列表信息，并丢弃Hello消息，不再转发。

第三步：通过OLSR协议的MPR节点生成算法，从邻居列表中选取出MPR节点集，并在Hello消息中通知相应的MPR节点。

第四步：收到通知的节点，生成相应的MPR选取者集，并使用TC消息将自己的MPR选取者集广播给网络中的其它节点。

第五步：接收到TC消息的节点更新拓扑表，并以此计算出路由表。

通过以上对路由发现过程的分析可知，对特定节点而言，所涉及的路由消息操作只包括收发Hello消息、收发TC消息。具体过程可概括为：所有节点将周期性收发Hello消息，MPR节点还将周期性收发TC消息。当收到Hello消息后，节点将更新一跳邻居节点表和两跳邻居节点表以及MPR表和MPR选取者表。当收到TC消息后，节点将更新拓扑表和路由表，如果该节点为MPR节点，将继续转发这个TC消息。

在实际网络环境中，Hello消息和TC消息携带的路由信息，是对网络拓扑情况的抽象化描述。正常情况下，各个节点反映的网络拓扑应该与实际拓扑保持一致。当恶意攻击者对OLSR协议实施路由攻击时，必须对原有的Hello消息和TC消息进行修改，修改后的路由消息，必定与实际的网络拓扑相矛盾，因此可以通过比较收到的Hello消息和TC消息反映的拓扑信息与实际网络拓扑的一致性，判断是否存在恶意攻击。

# OLSR协议的安全性分析

|  |  |
| --- | --- |
| 消息类型 | 关键域 |
| Hello消息 | 一跳邻居节点集 |
| MPR节点集 |
| TC消息 | MPR选取者集 |
| 广播邻居序列号 |

**OLSR协议路由消息的关键域**

在OLSR协议中，路由消息主要为Hello消息和TC消息，因此攻击者一定会伪造或篡改这两个路由消息。所有针对OLSR协议的路由攻击，都会使网络中出现异常的Hello消息和异常的TC消息。

由于TC消息需要全网转发，对TC消息的攻击，可以分为初始伪造和转发篡改两项。因此，针对Hello消息和TC消息关键域的攻击，可能包括以下四种基本攻击方法。

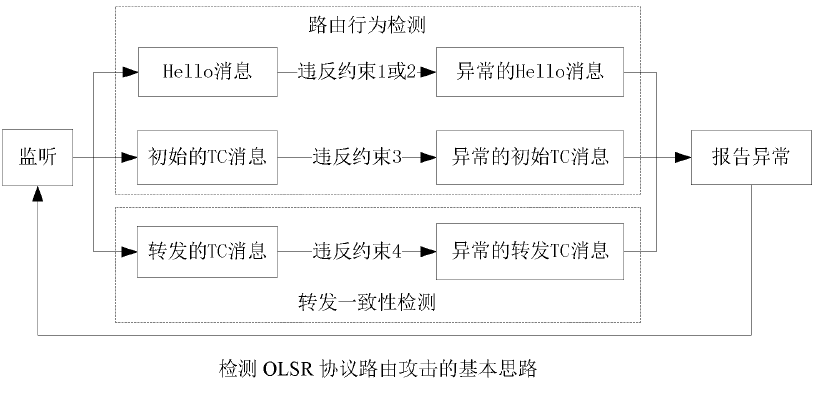
* A1：在Hello消息中，提供错误的一跳邻居节点列表。可能将不存在的节点添加到邻居列表中，或者将真实的邻居节点从邻居列表中删除。
* A2：在Hello消息中，提供错误的MPR节点集，即提供的MPR节点集与实际计算得到的MPR节点集不符。
* A3：在初始发送的TC消息中，提供错误的MPR选取者集。
* A4：在转发TC消息时，篡改MPR选取者集或广播邻居序列号。

总结以上四种基本攻击方法，可以分为两大类：A1、A2、A3为伪造本地路由消息攻击，使生成的Hello消息或TC消息与实际网络拓扑不符；A4为篡改转发路由消息攻击，使TC消息在转发过程中被恶意篡改。

OLSR路由消息与实际网络拓扑的对应关系，应符合以下四个约束条件：

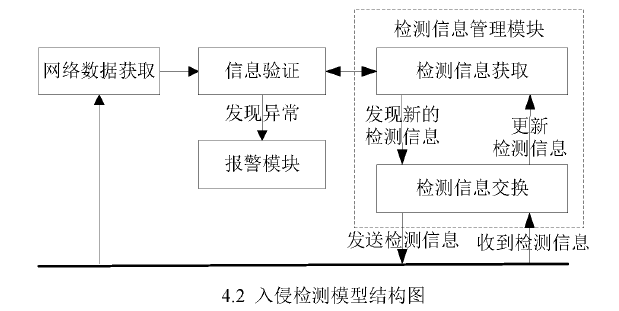
* C1：邻居节点的关系应该是相互的。如果节点2是节点1的邻居节点，那么节点1也应该是节点2的邻居节点。
* C2：一个节点的MPR节点集，必须能够一跳到达这个节点的所有两跳邻居节点。
* C3：一个节点的MPR选取者集，必须与相应的MPR节点对应。如果节点2是节点1的 MPR 选取者，那么节点1应该是节点2的MPR节点。
* C4：转发的TC消息必须保证一致性。当节点转发TC消息时，跳数增加1，生存时间减小1，其它内容保持不变。

根据这四个约束条件涉及的内容不同，可以划分为两大类。第一类：C1、C2、C3是路由消息与网络拓扑之间的约束关系，可用于检测生成的路由消息是否与网络拓扑一致。第二类：C4是转发路由消息的约束条件，可用于检测转发的路由消息是否正确。



# 入侵检测模型框架

入侵检测模型采用了分布式的检测方式，即位于各节点的入侵检测模型独立收集路由数据，检测网络中是否存在恶意路由攻击。入侵检测模型的基本结构如下图所示。



入侵检测模型共由网络数据获取模块、信息验证模块、报警模块、检测信息管理模块四部分组成。网络数据获取模块，负责捕获 OLSR 协议收发的路由消息，并将这些消息传递给信息验证模块进行入侵检测。信息验证模块，根据路由协议的约束条件和掌握的检测信息，对捕获的路由消息进行验证。如果发现异常，激活报警模块向用户发出入侵警报并记录入侵情况。如果没有发现异常，则传递给检测信息管理模块。检测信息管理模块由检测信息获取模块和检测信息交换模块组成。检测信息获取模块，负责从收到的路由消息中获取检测信息，当发现新的检测信息时，通过检测信息交换模块通知其它节点。当掌握的检测信息不足以完成入侵检测时，通过检测信息交换模块从其它节点获取检测信息。下面将对各模块做详细介绍。

# 数据获取模块

网络数据获取模块，用于获取入侵检测模型需要验证的路由消息。入侵检测模型需要完成对路由消息的检测，因此网络数据获取模块只负责捕获路由消息。本系统设计的网络数据获取模块，通过捕获路由协议收发的路由消息，获得检测数据。

# 信息验证模块

信息验证模块主要进行路由行为检测，负责对 Hello 消息和初始 TC 消息进行检测。Hello 消息携带一跳邻居节点信息和 MPR 节点信息，TC 消息携带 MPR 选取者信息。Hello 消息为一跳消息，初始 TC 消息第一次被接收时也是一跳消息，相对于检测节点而言，它们携带的信息都是对检测节点两跳范围内网络拓扑的描述。因此只要检测节点掌握两跳以内的拓扑信息，就能完成对 Hello 消息和初始 TC 消息的检测。在 OLSR 协议中，节点可以通过 Hello 消息掌握一跳邻居节点的拓扑信息，但是无法掌握两跳邻居节点的拓扑信息。因此本系统设计了检测信息交换消息，用于两跳邻居节点间交换拓扑信息。通过 Hello消息和检测信息交换消息，检测节点可以掌握两跳以内邻居节点的拓扑信息，实现对 Hello消息和初始 TC 消息正确性的检测。

## 入侵检测方法

### A1入侵方式检测方法

A1入侵方式检测过程如下：当收到Hello消息后，提取邻居节点信息，使用约束条件C1（邻居节点关系是相互的）进行检测。例如节点1收到节点2发送的Hello消息，声称节点3为节点2的邻居节点，那么节点1在检测信息库中查找，是否节点3也将节点2作为邻居。

A1算法主要涉及Hello 消息的检测。Hello 消息包括邻居节点信息和 MPR 节点信息两项内容。如果节点 F 需要检测节点 A 发送的 Hello 消息，那么节点 F 就需要知道节点 A 的邻居节点情况。对于节点 F 而言，节点 A 的邻居节点可以分为以下三类：节点 F 本身、节点 F的一跳邻居节点如节点 B 和 E、节点 F 的两跳邻居节点如节点 C 和 D。可以看出节点 A 的邻居节点，必定在节点 F 的两跳邻居节点范围内，因此节点 F 只要掌握两跳以内邻居节点的拓扑信息，就可以完成对节点A发送Hello消息的检测。通过节点F与邻居节点间的Hello消息和检测信息交换消息，节点 F 可以掌握两跳以内邻居节点的拓扑信息，因此节点 F 可以检测节点 A 发送的 Hello 消息的正确性。

下面介绍具体检测流程。首先定义如下的信息表和检测消息类型：

* 节点保存的信息表
* 本地链路信息表
* 邻居表
* 两跳邻居表
* MPR Selector表
* TC分组重复记录表
* 拓扑表
* 路由表

同时定义三种消息类型：检测证据发布消息(Evidence-send)、转发节点通告消息(Forwarder-announce)和检测证据请求消息(Evidence-request)。节点A被称为检测证据源节点，节点B被称为检测证据转发节点，节点C被称为检测证据请求节点。三种消息的具体定义如下：

* 检测证据发布消息：由检测证据转发节点发送，用于向邻居节点发布关于检测证据源节点的检测信息。
* 转发节点通告消息：由检测证据源节点发送，通知目的节点已经被选择为检测证据转发节点，以后将负责检测证据发布消息的发送。
* 检测证据请求消息：在入侵检测过程中，发现检测信息不足时，发送这个消息，请求检测证据转发节点发送关于检测证据源节点的检测信息

下面介绍检测方法的具体实现。节点B对节点C发送的Hello消息中邻居节点情况进行检测。在Hello消息中，节点C声称自己的邻居节点包括节点（A、B、D），因此节点B必须掌握节点（A、B、D）的邻居节点情况，才能完成检测。节点B的检测模块通过自己发送的Hello消息，可知节点C为自己的邻居节点。通过节点A发送的Hello消息可知，节点C也为节点A的邻居节点。节点D为节点B的两跳邻居节点，只能通过检测信息交换消息得知节点D的邻居节点情况。

当节点D发现新的检测信息时，向节点A发送转发节点通告消息，通知节点A发送关于节点D的检测信息。节点A收到节点D的转发节点通告消息后，广播关于节点D的检测证据发布消息。通过上述检测信息交换过程，节点B掌握了两跳节点D的邻居节点情况，从而可以判断节点C发送的Hello消息中声称的邻居节点是否正确。

下面介绍A1入侵方式的检测方法程序实现。首先定义基本数据结构如下：

struct OneHopNeighbour//一跳邻居节点集

{

int neigh;

OneHopNeighbour\* next;

};

struct TwoHopNeighbour//两跳邻居节点集

{

int neigh;

int middle;

struct TwoHopNeighbour\* next;

};

struct Defend//邻居检测链表

{

int ori;//当前节点

struct OneHopNeighbour\* one; //一跳邻居节点集

struct TwoHopNeighbour\* two; //两跳邻居节点集

int cnt;

Defend\* next;

};

static Defend\* DefendHead;//邻居检测链表

编写VerifyA1（Node，Neighbor）函数，负责利用（Node，Neighbor）节点对来更新邻居检测链表，并检查邻居检测链表是否存在异常。其中Node为当前节点，DefendHead为Hello消息邻居检测链表。具体伪代码如下：

VerifyA1(Node, Neighbor)//A1入侵方式检测

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*先将(Node, Neighbor)邻居节点对添加到邻居检测链表中\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if(DefendHead == OPC\_NIL)

//邻居检测链表为空

{

向DefendHead添加Node 节点，并添加(Node, Neighbor)邻居对;

向DefendHead添加Neighbor节点，并添加(Neighbor, Node)邻居对;

}

else

//检测当前Node-Neighbor以及Neighbor-Node节点对是否在邻居检测链表中

{

if(DefendHead中找到Node节点)

{

if(Node节点一跳邻居含有 Neighbor)

Node节点cnt = Node节点cnt+1;

else

{

向Node节点添加(Node, Neighbor)邻居对;

向Neighbor节点添加(Neighbor, Node)邻居对;

}

}

else//未在DefendHead找到Node节点

{

向DefendHead添加Node 节点，并添加(Node, Neighbor)邻居对;

向DefendHead添加Neighbor节点，并添加(Neighbor, Node)邻居对;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*检查邻居检测链表是否存在异常\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CheckTemp = DefendHead;

while(CheckTemp != OPC\_NIL)

{

CheckOrigin = CheckTemp当前节点;

CheckOne = CheckTemp当前节点的一跳邻居节点集;

while(CheckOne != OPC\_NIL)

{

ToBeCheck = CheckOne->neigh;//检查当前节点每一个邻居节点

//先检查 DefendHead 表中是否含ToBeCheck

SearchOrigin = DefendHead;

while((SearchOrigin非空且对应的节点!= ToBeCheck))

{

SearchOrigin = SearchOrigin->next;

}

if(SearchOrigin == OPC\_NIL)

{

printf("伪造的邻居地址");

}

else

//然后检查ToBeCheck的邻居是否含 CheckOrigin

{

SearchOne = SearchOrigin->one;

while((SearchOne非空且对应的的邻居!= CheckOrigin))

{

SearchOne = SearchOne->next;

}

if(SearchOne == OPC\_NIL)

{

printf("错误的邻居信息：(CheckOrigin, ToBeCheck)”);

}

}

CheckOne = CheckOne->next;

}

CheckTemp = CheckTemp->next;

}

}

### A2入侵方式检测方法

A2入侵方式检测过程如下：当收到 Hello 消息后，提取 MPR 节点信息，使用约束条件 C2(MPR 节点集必须可以一跳到达所有的两跳邻居节点)进行检测。例如节点 1 收到节点 2 发送的 Hello 消息，声称节点 3 为 MPR 节点，那么节点 1 在检测信息库中查找，是否节点 3 可以一跳到达节点 2 的所有两跳邻居节点。

A2算法主要涉及Hello 消息的检测。Hello 消息包括邻居节点信息和 MPR 节点信息两项内容。节点 B对节点 E 发送的 Hello 消息中 MPR节点情况进行检测。在 Hello 消息中，节点 E 声称节点 A 为 MPR 节点。为了完成对 Hello消息中 MPR 节点的检测，节点 B 需要判断节点 A 的一跳邻居节点集是否与节点 E 的两跳邻居节点集相同。节点 A 为节点 B 的邻居节点，因此节点 B 可以通过 Hello 消息掌握节点A 的邻居节点集(B、C、D、E)。节点 E 的两跳邻居节点集是它的所有邻居节点(A、B、D)的一跳邻居节点的并集。因为节点 B 是节点 E 的邻居节点，因此对于节点 B 而言，节点 E 的邻居节点分为以下三种情况：节点 B 本身、节点 B 的一跳邻居节点如节点 A、节点B 的两跳邻居节点如节点 D。根据A1算法的介绍可知，节点 B 可以掌握这些节点的邻居节点情况，因此可以判断节点 E 声称的 MPR 节点是否可以一跳到达节点 E 的所有两跳邻居节点。

具体检测流程与A1算法类似。首先定义与A1算法相同的数据结构，然后编写VerifyA2（Node，Neighbor）函数，负责利用（Node，Neighbor）节点对来更新两跳邻居检测链表，并检查Node和Neighbor的 MPR节点集是否存在异常。其中Node为当前节点，DefendHead为Hello消息的一跳及两跳邻居检测链表。具体伪代码如下：

VerifyA2(Node, Neighbor)//A2入侵方式检测

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*先将(Node, Neighbor)邻居节点对添加到两跳邻居检测链表中\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if(DefendHead == OPC\_NIL)

//邻居检测链表为空

{

向DefendHead添加Node 节点，并添加(Node, Neighbor)邻居对;

向DefendHead添加Neighbor节点，并添加(Neighbor, Node)邻居对;

}

else

//检测当前Node-Neighbor以及Neighbor-Node节点对是否在两跳邻居检测链表中

{

if(DefendHead中找到Node节点)

{

if(Node节点一跳邻居含有 Neighbor)

Node节点cnt = Node节点cnt+1;

else

{

向Node节点添加(Node, Neighbor)邻居对;

向Neighbor节点添加(Neighbor, Node)邻居对;

向Node节点的邻居节点添加(Node, Neighbor)两跳邻居对;

向Neighbor节点的邻居节点添加(Neighbor, Node) 两跳邻居对;

}

}

else//未在DefendHead找到Node节点

{

向DefendHead添加Node 节点，并添加(Node, Neighbor)邻居对;

向DefendHead添加Neighbor节点，并添加(Neighbor, Node)邻居对;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*检查MPR节点集是否存在异常\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

for(i=0; i<Node的MPR节点个数; i++)

{

ToBeCheck = Node的第i个MPR节点;

if(ToBeCheck的两跳邻居节点集 == Node的一跳邻居节点集)

{

printf("伪造的MPR节点：ToBeCheck");

}

}

for(i=0; i<Neighbor的MPR节点个数; i++)

{

ToBeCheck = Neighbor的第i个MPR节点;

if(ToBeCheck的两跳邻居节点集 == Neighbor的一跳邻居节点集)

{

printf("伪造的MPR节点：ToBeCheck");

}

}

}

# 检测信息管理模块

检测信息管理模块包括检测信息获取模块和检测信息交换模块两部分。检测信息获取模块主要负责提取检测信息，检测信息交换模块主要负责与其它两跳邻居节点进行检测信息交换。为了实现两跳邻居节点间检测信息的交换，设计了检测证据发布、转发节点通告、检测证据请求三个检测信息交换消息。检测信息获取模块从接收的 Hello 消息、自己发送的 Hello 消息和检测证据发布消息中获取所需的检测信息。当检测信息不足、发现新的检测证据、收到检测证据请求消息或转发节点通告消息时，检测信息交换模块与其它节点进行检测信息交换。