详细设计

|  |  |  |  |
| --- | --- | --- | --- |
| 模块名称 | 应用层协议分析模块 | 标识 |  |
| 父模块 | 无 | 回溯标识 |  |
| 模块功能 | 该模块用于对一个数据包的应用层协议做分析，来发现当前网络中使用的常用软件。首先，系统读取由用户指定的待检测协议，将用于协议匹配的正则表达式加载入内存。然后，当数据包到达时，系统将数据包中应用层数据匹配正则表达式，如果匹配成功，则输出源IP、源端口、匹配的时间、协议名到数据库 | | |
| 相关数据 | 请见子模块 | | |
| 限制条件 | 请见子模块 | | |
| 输入数据 | //以太网数据包  const char\* pkt | | |
| 输出数据 | 数据库表：  /\*====================================================\*/  /\* Table: approto \*/  /\*====================================================\*/  CREATE TABLE approto  (  id INT UNSIGNED NOT NULL AUTO\_INCREMENT,  time INT UNSIGNED NOT NULL ,  ip INT UNSIGNED NOT NULL,  port SMALLINT UNSIGNED NOT NULL,  proto VARCHAR(255) NOT NULL,  PRIMARY KEY(id)  );  id: 由mysql自动生成的记录标号，作为主键  time: 该次匹配成功的时间  ip: 局域网中用户所使用的IP地址  port: 局域网中用户使用该协议时所使用的端口号  proto: 局域网中用户使用的协议 | | |
| 图例 | 模块层次图 | | |
| 算法逻辑 | ///伪代码  ParsePatternFiles(); //规则管理模块  if ( ParsePacket(const char\* pkt) ) //连接管理、规则匹配模块  {  if ( FiltrateOutput() ) //输出过滤模块  {  OutPut(); //输出模块  }  } | | |
| 调用接口 | /// 连接管理模块  /// 提供连接管理和规则匹配功能  /// @param pkt 以太网数据包，从以太帧头部开始  /// @param srcip 分析后返回的源IP地址  /// @param srcport 分析后返回的源端口  /// @param proto 分析后返回的协议名  /// @return 协议号  int ParsePacket(const char\* pkt, int\* srcip, short\* srcport, char\*\* proto);  /// 输出过滤模块  /// 提供匹配结果的过滤功能  /// @param srcip 源ip地址  /// @param proto 协议名  /// @return 返回0，数据不需要输出；返回1, 数据需要输出  int FiltrateOutput (unsigned int srcip, char\* proto)；  /// 规则管理模块  /// 提供规则的管理  /// @param filename 配置文件的文件名  void ParseConfigurationFile(const char\* fileName); | | |
| 提供接口 | /// 应用层协议分析模块  /// 提供对数据包的应用层进行协议分析的功能  /// @param pkt 以太网数据包，从以太帧头部开始  void ApprotoDetection(cosnt char\* pkt); | | |
| 备注 | 无 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 模块名称 | 规则文件管理模块 | 标识 |  |
| 父模块 | 应用层协议分析模块 | 回溯标识 |  |
| 模块功能 | 1． 读取本地规则配置文件。  2． 根据配置文件查找指定协议的规则文件。  3． 将规则文件中的正则表达式编译并加载入内存  4． 读取WEB端规则配置文件  5． 根据配置文件查找指定协议的规则文件。  6． 将规则文件中的正则表达式编译并加载入内存 | | |
| 相关数据 | /// 该结构定义了规则的存储  struct Pattern  {  int mark; ///< 表示该规则对应的协议号  int eflags; ///< 处理正则表达式用到的标识  int cflags; ///< 处理正则表达式用到的标识  char\* name; ///< 协议名  char\* pattern\_string; ///< 编译后的正则表达式  regex\_t preg; ///< 处理正则表达式用到的句柄  }; | | |
| 限制条件 | 该模块用到了正则表达式库，对读入的正则表达式进行编译，为后面正则表达式的匹配打下基础。  本地规则配置文件：由管理员在本地指定的需要检测的应用层协议名  WEB端规则配置文件：由管理员在WEB端指定的需要检测的应用层协议名  本地规则文件夹：由管理员在本地指定的需要检测的规则文件集  WEB端规则文件夹：由管理员在WEB端指定的需要检测的规则文件集，其通过数据库来获取 | | |
| 输入数据 | //配置文件名  const char\* filename | | |
| 输出数据 | Pattern结构链表 | | |
| 图例 |  | | |
| 算法逻辑 | /// 伪代码  while ( 从配置文件中获取一行 )  {  分析当前行  if ( 当前行为空格或者注释 )  {  continue;  }  else  {  找到并分析规则文件;  获取正则表达式并挂到链表上  }  } | | |
| 调用接口 | 无 | | |
| 提供接口 | /// 规则管理模块  /// 提供规则的管理  /// @param filename 配置文件的文件名  void ParseConfigurationFile(const char\* fileName); | | |
| 备注 | 无 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 模块名称 | 连接管理模块 | 标识 |  |
| 父模块 | 应用层协议分析模块 | 回溯标识 |  |
| 模块功能 | 为了提高正则表达式匹配速度的效率，引入连接管理的概念。首先把一个数据包的源IP地址、源端口号、目的IP地址、目的端口看成一个整体，这个四元组唯一标识了一个连接，当一个连接连续匹配失败次数超过5次，则清除匹配失败的连接，使连接信息不会占用过多内存，为后面正则表达式匹配打下基础。  过程：  1． 查找当前连接（由源IP,源端口号,目的IP,目的端口号组成）是否是旧连接，若是，则读取旧连接，否则新建一个连接。  2． 查看该连接是否已经匹配成功,若匹配成功,则跳过后续操作  3． 若该连接上数据未匹配成功过,则遍历正则表达式链表，匹配当前连接上应用层的数据  4． 若匹配成功，则输出源IP、源端口号、协议名、匹配成功的时间  5． 若连续匹配失败超过5次，则删除该连接 | | |
| 相关数据 | /// 使用的底层数据结构  map<char[12], struct Connection\*>  /// 该结构定义了一个连接的表示和存储  struct Connection  {  unsigned int num\_packets; ///< 该连接上连续匹配包的数量  unsigned int mark; ///< 该连接上匹配的协议号  char key[12]; ///< 该连接的索引号  }; | | |
| 限制条件 | Snort：snort提供的以太网数据包作为输入 | | |
| 输入数据 | const char\* pkt; | | |
| 输出数据 | srcip: 匹配到的源IP地址  srcport: 匹配到的源端口号  dstip: 匹配到的目的IP地址  dstport: 匹配到的目的端口号  proto: 匹配到的协议名  mark: 匹配到的协议号  time: 匹配的时间 | | |
| 图例 |  | | |
| 算法逻辑 | ///伪代码  初始化map结构；  if ( 数据包是tcp或者udp )  {  map查找当前连接；  if ( 当前连接是旧连接 )  {  继续操作；  }  else  {  建立新连接；  }  if ( 当前连接已经匹配成功 )  {  return;  }  else if ( 当前连接匹配次数 <= 5 )  {  匹配正则表达式；  if ( 匹配到的协议号 > 2 )  {  返回源IP和源端口；  }  }  else  {  从map中删除当前连接;  }  } | | |
| 调用接口 | /// 规则匹配模块  /// @param data 待匹配的应用层数据  /// @param datalen data的长度  /// @param proto 匹配到的协议名  /// @return 匹配到的协议号  int Classify(char\* data, int datalen, char\*\* proto); | | |
| 提供接口 | /// 连接管理模块  /// 提供连接管理和规则匹配功能  /// @param pkt 以太网数据包，从以太帧头部开始  /// @param srcip 分析后返回的源IP地址  /// @param srcport 分析后返回的源端口  /// @param proto 分析后返回的协议名  /// @return 协议号  int ParsePacket(const char\* pkt, int\* srcip, short\* srcport, char\*\* proto); | | |
| 备注 | 无 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 模块名称 | 规则匹配模块 | 标识 |  |
| 父模块 | 连接管理模块 | 回溯标识 |  |
| 模块功能 | 1． 去除应用层数据中的0x00，防止匹配不完整  2． 遍历正则表达式链表，逐个匹配正则表达式  3． 若匹配成功，则返回协议号和协议名称  4． 若匹配不成功，返回协议号为0 | | |
| 相关数据 | 无 | | |
| 限制条件 | 无 | | |
| 输入数据 | char\* data； //待匹配的应用层数据  int datalen,； // data的长度  char\*\* proto； //匹配到的协议名 | | |
| 输出数据 | 匹配到的协议号和协议名 | | |
| 图例 |  | | |
| 算法逻辑 | /// 伪代码  去除data中的0x00；  while ( 规则链表中当前项合法 )  {  if ( data匹配正则表达式得到的协议号 > 2)  {  保存协议名；  break;  }  检查下一项；  }  return 协议号； | | |
| 调用接口 | 无 | | |
| 提供接口 | /// 规则匹配模块  /// @param data 待匹配的应用层数据  /// @param datalen data的长度  /// @param proto 匹配到的协议名  /// @return 匹配到的协议号  int Classify(char\* data, int datalen, char\*\* proto); | | |
| 备注 | 无 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 模块名称 | 输出过滤模块 | 标识 |  |
| 父模块 | 应用层协议分析模块 | 回溯标识 |  |
| 模块功能 | 为了提高管理员查看数据的效率，匹配结果需要被过滤后才插入数据库。过滤算法使用map作为底层数据结构，局域网中用户IP作为key，将该用户所使用的协议信息存为链表作为value。当某个用户在某个时间段内连续使用某一软件，则数据只被记录一次，这样很大减少数据库的负担，也使得数据库数据清晰明了，降低管理员的负担  过程：   1. 检查当前检测到的协议是否已经检查过 2. 若检查过，并且超过了检查间隙时间，则更新检查时间；反之，跳过后续操作 3. 若没检查过，则将记录下当前协议名和匹配时间等数据 | | |
| 相关数据 | /// 使用的底层数据结构  map<unsigned int ip, struct Info\*>;  /// 该结构定义了一个ip上所使用的应用层协议信息  struct Info  {  time\_t time; ///< 最近使用当前协议的时间  char\* proto; ///< 协议名  struct Info\* next; ///< 指向下一个info  }; | | |
| 限制条件 | 无 | | |
| 输入数据 | unsigned int srcip; //匹配到的源IP  char\* proto; //匹配到的协议名 | | |
| 输出数据 | 过滤的结果（源IP，源端口、协议名、匹配时间） | | |
| 图例 |  | | |
| 算法逻辑 | 查找源IP对应的协议链表头;  if ( 找到协议链表头 )  {  查找协议节点；  if ( 找到协议节点 )  {  if ( 本次匹配时间 – 上次匹配时间 > 1分钟 )  {  修改协议最后匹配时间；  return 1;  }  else  {  return 0;  }  }  else  {  插入当前节点；  return 1;  }  }  else  {  建立协议链表头；  插入当前节点；  return 1;  } | | |
| 调用接口 | 无 | | |
| 提供接口 | /// 输出过滤模块  /// 提供匹配结果的过滤功能  /// @param srcip 源ip地址  /// @param proto 协议名  /// @return 返回0，数据不需要输出；返回1, 数据需要输出  int FiltrateOutput (unsigned int srcip, char\* proto)； | | |
| 备注 | 无 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 模块名称 | 输出模块 | 标识 |  |
| 父模块 | 应用层协议分析模块 | 回溯标识 |  |
| 模块功能 | 为了提高数据库插入操作的效率，提高数据匹配和数据库IO操作的并行性，数据必须进行缓冲。双缓冲使用2个固定大小的数组作为2条循环队列，由2个应用级线程来管理，采用读线程优先算法，使得当读线程空闲时就进行队列切换，无需等待另外队列装满，比起单缓冲，双缓冲极大减少了读写的同步开销，提高了匹配和输出的并行性。  过程：   1. 写线程将数据拷贝到由写指针指定的缓冲区 2. 读线程从由读指针指定的缓冲区读取数据，并执行数据库IO操作。 | | |
| 相关数据 | /// 该结构表示缓冲区的元素  struct Insertion  {  unsigned int time; ///< 匹配时间  unsigned int ip; ///< 源IP  unsigned short port; ///< 源端口  char proto[255]; ///< 协议名  }; | | |
| 限制条件 | Mysql：分析出的结果通过mysql输出 | | |
| 输入数据 | //数据库输出需要的结构  struct Insertion \_insertion | | |
| 输出数据 | 数据库操作语句字符串 | | |
| 图例 | 写线程：  读线程： | | |
| 算法逻辑 | /// 伪代码(写线程)  if ( 读线程未打开 )  {  打开读线程；  }  if ( 写缓冲区满 )  {  sleep(1);  }  else  {  写入当前数据；  }  /// 伪代码( 读线程 )  if ( 读线程空闲 )  {  if ( 写线程空闲 )  {  sleep(1);  }  else  {  加锁；  切换队列；  解锁；  }  }  else  {  读取数据；  写数据库；  } | | |
| 调用接口 | 无 | | |
| 提供接口 | /// 提供存放缓冲区的功能  /// @param \_insertion 数据库输出需要的结构  Q\_PUT(struct Insertion \_insertion) | | |
| 备注 | 无 | | |