

# 数据库防火墙 Suricata 开发文档

tanb

Apr 22, 2014

Table 1: 文档更新历史

日期	作者	详情
May 4, 2014	tanb	加入 <a href="#">2.1.4</a> 节；修正之前的一些描述错误
May 6, 2014	tanb	加入 <a href="#">3.7</a> 节；
Jun 3, 2014	tanb	更正一些描述错误
Jun 6, 2014	tanb	加入 demo( <a href="#">4</a> ) 章节
Jun 17, 2014	tanb	修正 Oracle 协议中超长字符串的描述 ( <a href="#">2.1.2</a> )

# Contents

1	数据库环境的准备	5
1.1	Oracle 11g R2 环境的安装和使用	5
1.2	Mysql 环境的安装和使用	7
1.3	DB2 环境以及 SQL Server 的安装和使用	8
1.4	如何用 Wireshark 抓取数据包	8
2	数据库通信协议分析	15
2.1	Oracle 数据包协议分析	16
2.1.1	Oracle 登陆数据包的组成	17
2.1.2	Oracle 查询数据包的组成	19
2.1.3	Oracle 中其它数据包分析	24
2.1.4	Oracle 12C TNS 协议的异同	28
2.1.5	Oracle 退出登陆数据包的组成	28
2.2	SQL Server 通信协议分析	28
2.3	IBM DB2 数据包协议分析	30
2.4	MySQL 数据包协议分析	31
2.5	超长 SQL 语句的处理	31
3	Suricata 开发手册	33
3.1	Suricata 的整体架构	33
3.2	在 Suricata 中加入 MySQL 协议检测	34
3.3	修改 Suricata 配置文件	36

3.4	在 Suricata 中增加关键字识别 .....	36
3.5	增加关键字识别的实现 .....	37
3.6	增加 Json 输出模块 .....	39
3.7	Suricata 开发过程中碰到的疑难问题 .....	39
3.7.1	Suricata 只能将小部分的 TNS 协议数据包送到应用协议层 .....	39
4	Suricata 预研 Demo .....	41
4.1	主要内容 .....	41
4.2	网络结构 .....	41
4.3	Suricata IDS/IPS 对现有数据库协议支持的展示 .....	41
4.3.1	IDS 功能演示 .....	42
4.3.2	IPS 功能演示 .....	43

# Chapter 1

## 数据库环境的准备

本章主要记录如何准备各种目标数据库，其中着重记录了 **Oracle 11g R2** 的安装以及配置。因为它总是如此让人捉摸不透。

### 1.1 Oracle 11g R2 环境的安装和使用

近期可获取的 **Oracle** 的最老版本为 **11g R2**，安装之前可以先去 **Oracle** 官网下载该版本的数据库。

准备条件：<sup>1</sup>

- 一个带桌面环境的 **Linux** 发行版，将其作为宿主机，你需要它的 **X** 环境来点击 **Oracle** 安装过程中的各种按钮。
- **Oracle** 专用的 **Linux** 发行版，推荐 OracleLinux-R6-U5-Server-x86\_64-dvd.iso，如果需要其它版本，到[这里](#)下载。
- **Oracle 11g R2**，到 **Oracle** 官网下载两个文件 linux.x64\_11gR2\_database\_1of2.zip 和 linux.x64\_11gR2\_database\_2of2.zip。
- **Oracle** 的 instant client，到 **Oracle** 官网下载 instantclient-basic-linux-12.1.0.1.0.zip 和 instantclient-sqlplus-linux-12.1.0.1.0.zip。<sup>2</sup>

如果没有实体机器，就用 **Virtualbox** 虚拟几个，一个用于安装 **Oracle Linux**，一个用于安装客户端，客户端推荐使用 **Linux**，可以用 **Ubuntu**，为节约性能，直接使用不带 **UI** 的 **Ubuntu** 即可，如 ubuntu-12.04.4-server-amd64.iso；但对服务器端

---

<sup>1</sup>本节中的内容全部在 **Linux** 环境，不考虑 **Windows** 的兼容性。

<sup>2</sup>我下载的大版本号是 **12**，其实它是兼容服务器端的 **11** 版本号。如果碰到问题，请直接下载其版本为 **11** 的客户端。

而言，最好选择 Oracle 自己的 Linux 发行版，其它版本的 Linux 虽然也可以安装 Oracle，但说多了都是泪，直接用 Oracle 自己的 Linux 发行版是最好的选择。

在 Virtualbox 中安装这两个 Linux 系统，如果可以，将服务器的内存尽量放大一点（2G+）。安装完后，将两个虚拟机的 IP 设置为桥接模式，重启网络后<sup>3</sup>，这样它们与宿主机就处于同一网段，假定 Oracle Linux 的 IP 为 192.168.37.194，Ubuntu 客户机的 IP 为 192.168.37.193。

有时候安装的 Oracle Linux 只能使用环回地址（lo），用 `ifconfig -a` 查看后，确实有 `eth0`，此时需要启动一下该网卡，并设置其通过 DHCP 获取 IP 地址。

```
ifconfig eth0 up
# 在 /etc/sysconfig/network-scripts/ifcfg-eth0 中
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
# 然后重新启动 eth0
ifup eth0
```

接下来将下载的 Oracle 数据库安装文件拷贝到 194，将 `instantclient` 拷贝到 193。分别以 SSH 登录到这两台机器。

```
xhost +192.168.37.194 # 设定 X 服务，使得 194 可以访问 host 上的 X 服务
ssh -X 192.168.37.194
ssh -X 192.168.37.193
```

由于两个虚拟机都不带桌面环境，所以此处用 `-x` 选项，使得它们可以访问宿主主机上的桌面环境。

在 Oracle Linux 上安装 Oracle 11g R2，参见[这里](#)的详细步骤。

然后安装客户端，将两个 zip 文件解压后，会得到目录 `instantclient_12_1`，然后设置两个环境变量 `PATH` 和 `LD_LIBRARY_PATH`，将它们指向该目录。

接下来在 `instantclient_12_1` 中，编辑两个文件。第一个为 `sqlnet.ora`，在其中加入如下内容

```
SQLNET.AUTHENTICATION_SERVICES= (NTS)
NAMES.DIRECTORY_PATH= (TNSNAMES, EZCONNECT)
```

第二个文件为 `tnsnames.ora`，在其中加入如下内容

```
databasename =
(DESCRIPTION =
  (ADDRESS_LIST =
    (ADDRESS = (PROTOCOL = TCP) (HOST = 192.168.37.194) (PORT = 1521))
  )
  (CONNECT_DATA =
    (SERVICE_NAME = orcl11g)
  )
)
```

---

<sup>3</sup>/etc/init.d/network restart

为便于连接 194 上的数据库，编写脚本 login-oracle.sh：

```
sqlplus system/test@'(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP) \
  (HOST=192.168.37.194) (PORT=1521)) (CONNECT_DATA=(SID=orcl11g)))'
```

客户端配置完毕，此时你应该可以在命令行执行该登录脚本。但是登录不会成功。

接下来配置服务器端，由于 Oracle Linux 默认开启了多个 iptables 的防火墙设置，所以需要将它们一一关闭，登录 194：

```
service iptables save
service iptables stop
chkconfig iptables off

service ip6tables save
service ip6tables stop
chkconfig ip6tables off
```

这样，在 193 上应该就能访问 194 上的服务了。

当下一次服务器重启的时候，可能你什么都找不到了，但是没关系，上面的链接中给了很多设定，最重要的是环境变量，没记错的话，应该都放到用户目录的 .bash\_profile 中去了。如果下一次启动的时候，这些环境变量都没了（典型的表现是输入 sqlplus 说找不到这个命令），那么在命令行执行

```
source .bash_profile
```

即可，这样所有环境变量又会来了。接下来启动 Oracle 数据库服务器：

```
dbstart &
```

执行上面的登陆脚本，此时应该就能正常登录了。

登陆进入 Oracle 数据库之后，可以用如下 SQL 语句，以测试数据包的抓取：

```
SELECT owner, table_name FROM dba_tables;
SELECT table_name FROM all_tables;
SELECT * FROM information_schema.columns WHERE table_name = 'abc';
SELECT DISTINCT OWNER FROM ALL_OBJECTS;

SELECT * from all_users; -- 列出所有用户
SELECT table_name FROM all_tables; -- 列出用户所能访问的表
SELECT * FROM all_tables; -- 列出用户所能访问的表
SELECT UTL_INADDR.get_host_address from dual; -- 列出当前登录的 server IP
```

## 1.2 Mysql 环境的安装和使用

这个比较简单，是个 Linux 都能跑 Mysql，以 Ubuntu 为例，直接在源中安装 mysql 即可。客户端也可以从源中安装。对于出现的错误，网上都有例子，以错误代码 Google 即可。不表。

### 1.3 DB2 环境以及 SQL Server 的安装和使用

DB2 和 SQL Server 的环境都比较好安装。DB2 分别提供了 Linux 和 Windows 的安装镜像。SQL Server 目前的版本为 SQL Server 2012，也可以去微软的官网下载。其安装和使用也比较简单，此处不表。

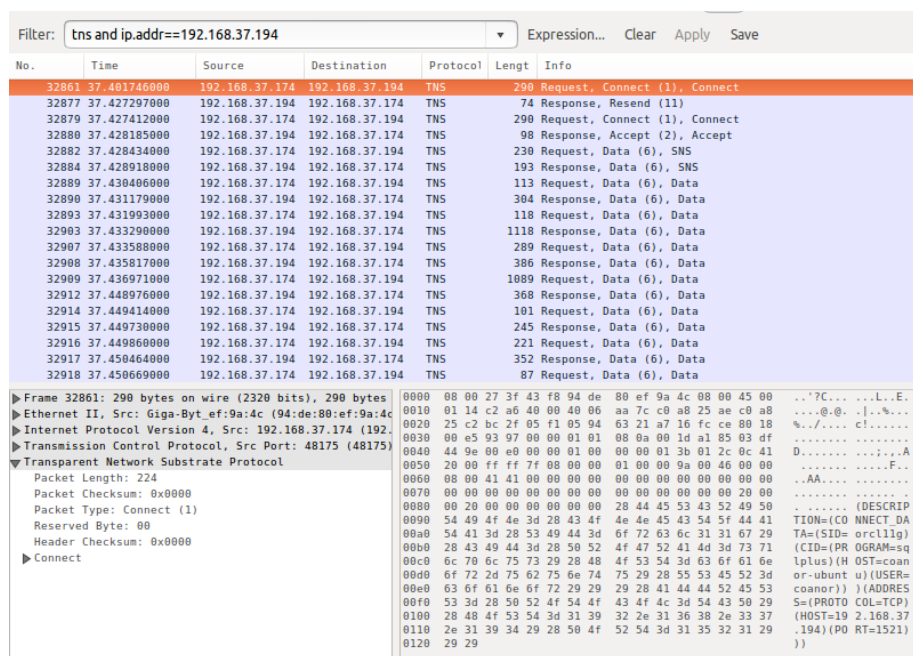
### 1.4 如何用 Wireshark 抓取数据包

本节主要讲述如何用 Wireshark 抓取数据包，由于要展现全貌，为便于排版，图片比较小，可放大查看。

以 Oracle 的 TNS 协议为例，下面演示了如何在 Wireshark 中抓取通信数据包的过程。

在 Linux 下，以 root 启动 Wireshark，设置一下侦听的网卡后即可抓取数据包。但是如果需要抓取特定协议的数据包，可以在 *Filter* 中输入过滤条件，以 TNS 协议为例，其设置如图 1.1 所示：

Figure 1.1: Wireshark 抓取的 Oracle 数据包示例



No.	Time	Source	Destination	Protocol	Length	Info
32861	37.481746000	192.168.37.174	192.168.37.194	TNS	290	Request, Connect (1), Connect
32877	37.427297000	192.168.37.194	192.168.37.174	TNS	74	Response, Resend (11)
32879	37.427412000	192.168.37.174	192.168.37.194	TNS	290	Request, Connect (1), Connect
32880	37.428185000	192.168.37.194	192.168.37.174	TNS	98	Response, Accept (2), Accept
32882	37.428434000	192.168.37.174	192.168.37.194	TNS	230	Request, Data (6), SNS
32884	37.428918000	192.168.37.194	192.168.37.174	TNS	193	Response, Data (6), SNS
32889	37.430406000	192.168.37.174	192.168.37.194	TNS	113	Request, Data (6), Data
32890	37.431179000	192.168.37.194	192.168.37.174	TNS	304	Response, Data (6), Data
32893	37.431993000	192.168.37.174	192.168.37.194	TNS	118	Request, Data (6), Data
32903	37.433290000	192.168.37.194	192.168.37.174	TNS	1118	Response, Data (6), Data
32907	37.433580000	192.168.37.174	192.168.37.194	TNS	289	Request, Data (6), Data
32908	37.435817000	192.168.37.194	192.168.37.174	TNS	386	Response, Data (6), Data
32909	37.436971000	192.168.37.174	192.168.37.194	TNS	1089	Request, Data (6), Data
32912	37.448976000	192.168.37.194	192.168.37.174	TNS	368	Response, Data (6), Data
32914	37.449414000	192.168.37.174	192.168.37.194	TNS	101	Request, Data (6), Data
32915	37.449730000	192.168.37.194	192.168.37.174	TNS	245	Response, Data (6), Data
32916	37.449860000	192.168.37.174	192.168.37.194	TNS	221	Request, Data (6), Data
32917	37.450464000	192.168.37.194	192.168.37.174	TNS	352	Response, Data (6), Data
32918	37.450669000	192.168.37.174	192.168.37.194	TNS	87	Request, Data (6), Data

Filter: tns and ip.addr==192.168.37.194

Expression... Clear Apply Save

No. Time Source Destination Protocol Length Info

32861 37.481746000 192.168.37.174 192.168.37.194 TNS 290 Request, Connect (1), Connect

32877 37.427297000 192.168.37.194 192.168.37.174 TNS 74 Response, Resend (11)

32879 37.427412000 192.168.37.174 192.168.37.194 TNS 290 Request, Connect (1), Connect

32880 37.428185000 192.168.37.194 192.168.37.174 TNS 98 Response, Accept (2), Accept

32882 37.428434000 192.168.37.174 192.168.37.194 TNS 230 Request, Data (6), SNS

32884 37.428918000 192.168.37.194 192.168.37.174 TNS 193 Response, Data (6), SNS

32889 37.430406000 192.168.37.174 192.168.37.194 TNS 113 Request, Data (6), Data

32890 37.431179000 192.168.37.194 192.168.37.174 TNS 304 Response, Data (6), Data

32893 37.431993000 192.168.37.174 192.168.37.194 TNS 118 Request, Data (6), Data

32903 37.433290000 192.168.37.194 192.168.37.174 TNS 1118 Response, Data (6), Data

32907 37.433580000 192.168.37.174 192.168.37.194 TNS 289 Request, Data (6), Data

32908 37.435817000 192.168.37.194 192.168.37.174 TNS 386 Response, Data (6), Data

32909 37.436971000 192.168.37.174 192.168.37.194 TNS 1089 Request, Data (6), Data

32912 37.448976000 192.168.37.194 192.168.37.174 TNS 368 Response, Data (6), Data

32914 37.449414000 192.168.37.174 192.168.37.194 TNS 101 Request, Data (6), Data

32915 37.449730000 192.168.37.194 192.168.37.174 TNS 245 Response, Data (6), Data

32916 37.449860000 192.168.37.174 192.168.37.194 TNS 221 Request, Data (6), Data

32917 37.450464000 192.168.37.194 192.168.37.174 TNS 352 Response, Data (6), Data

32918 37.450669000 192.168.37.174 192.168.37.194 TNS 87 Request, Data (6), Data

Frame 32861: 290 bytes on wire (2320 bits), 290 bytes captured (2320 bits) on interface eth0, 0 bytes captured on interface eth0, 0 bytes on interface eth0

Ethernet II, Src: Giga-Byt\_ef:9a:4c (94:de:88:ef:9a:4c), Dst: 192.168.37.194 (08:00:27:3f:43:f8:94:de)

Internet Protocol Version 4, Src: 192.168.37.174 (192.168.37.174), Dst: 192.168.37.194 (192.168.37.194)

Transmission Control Protocol, Src Port: 48175 (48175), Dst Port: 1521 (1521)

Transparent Network Substrate Protocol

Packet Length: 224

Packet Checksum: 0x0000

Packet Type: Connect (1)

Reserved Byte: 00

Header Checksum: 0x0000

Connect

0000 08 00 27 3f 43 f8 94 de 80 ef 9a 4c 08 00 45 00 ..?C... ..L..E..

0010 01 14 c2 a0 40 00 40 00 aa 7c c0 a8 25 ae c0 a8 ....@.@. [...%...

0020 25 c2 bc 2f 05 f1 05 94 63 21 a7 16 fc ce 80 18 %./.....C!.....

0030 00 e5 93 97 00 00 01 01 00 0a 00 1d a1 85 03 df D.....?..A

0040 44 9e 00 e0 00 00 01 00 00 00 01 3b 01 2c 0c 41 D.....?..A

0050 20 00 ff ff 7f 08 00 00 01 00 00 9a 00 46 00 00 ..AA.....F..

0060 08 00 41 41 00 00 00 00 00 00 00 00 00 00 00 ..AA.....

0070 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 ..... (DESCRIP

0080 00 20 00 00 00 00 00 00 28 44 45 53 43 52 49 50 ..... (DESCRIP

0090 54 49 4f 4e 3d 28 43 4f 4e 4e 45 43 54 5f 44 41 TION=(CO NNECT DA

00a0 54 41 3d 28 53 49 44 3d 6f 72 63 6c 31 31 67 29 TA=(SID= orcl11g)

00b0 28 43 49 44 3d 28 50 52 4f 47 52 41 4d 3d 73 71 (CID=(PR OGRAM=sq

00c0 6c 70 6c 75 73 29 28 48 4f 53 54 3d 63 6f 61 6e lplus)(H OST=coan

00d0 6f 72 2d 75 62 75 6e 74 75 29 28 55 53 45 52 3d or-ubunt u)(USER=

00e0 63 6f 61 6e 6f 72 29 29 28 41 44 44 52 45 53 coanor)) )(ADDRES

00f0 53 3d 28 50 52 4f 54 4f 43 4f 4c 3d 54 43 50 29 S=(PROTO COL=TCP)

0100 28 48 4f 53 54 3d 31 39 32 2e 31 36 38 2e 33 37 (HOST=19 2.168.37

0110 2e 31 39 34 29 28 50 4f 52 54 3d 31 35 32 31 29 .194)(PO RT=1521)

0120 29 29 ))

此处的过滤规则为 tns and ip.addr==192.168.37.194，其实，只要第一条即可，它限定了只显示与 TNS 协议相关的数据包。

图上列举的数据包中，来自一个 Oracle 的登陆操作。其中 174 作为客户端，



194 作为服务器端，在第一条数据包中，**Length** 列列出了数据包长度，**Info** 列列出了是该数据包的详细信息，其中 Request 指客户端请求服务器端，(1) 是数据包类型，此处指一个 Connect 操作。

在这条数据包中，从右下角的报文中可以看出，其中包含着一个 TNS 连接字符串，在该字符串中列举了连接的实例名 SID、客户端工具为 sqlplus，客户端的主机名，使用的通信协议（TCP），客户端的 IP 以及连接的服务器端的端口。

左下角即是具体的数据包分析，Wireshark 目前只能识别部分 TNS 协议，比如连接数据包，对其他数据包，则只能识别头部数据，数据包后面的数据则不能识别。

对客户端发送出来的连接数据包而言，其中列举了包长度、校验和、包类型等等，图 1.2 中列出了 Wireshark 所分析出来的登陆数据包：

Figure 1.2: Oracle 客户端发出的连接数据包

```

▶ Frame 1099: 290 bytes on wire (2320 bits), 290 bytes captured (2320 bits) on interface 0
▶ Ethernet II, Src: Giga-Byt_ef:9a:4c (94:de:80:ef:9a:4c), Dst: CadmusCo_3f:43:f8 (08:00:27:3f:43:f8)
▶ Internet Protocol Version 4, Src: 192.168.37.174 (192.168.37.174), Dst: 192.168.37.194 (192.168.37.194)
▶ Transmission Control Protocol, Src Port: 56810 (56810), Dst Port: ncube-lm (1521), Seq: 1, Ack: 1, Len: 224
▼ Transparent Network Substrate Protocol
  Packet Length: 224
  Packet Checksum: 0x0000
  Packet Type: Connect (1)
  Reserved Byte: 00
  Header Checksum: 0x0000
  ▼ Connect
    Version: 315
    Version (Compatible): 300
    ▶ Service Options: 0x0c41
      Session Data Unit Size: 8192
      Maximum Transmission Data Unit Size: 65535
    ▶ NT Protocol Characteristics: 0x7f08
      Line Turnaround Value: 0
      Value of 1 in Hardware: 0100
      Length of Connect Data: 154
      Offset to Connect Data: 70
      Maximum Receivable Connect Data: 2048
    ▼ Connect Flags 0: 0x41
      ...0 .... = NA services required: False
      .... 0... = NA services linked in: False
      .... .0.. = NA services enabled: False
      .... ..0. = Interchange is involved: False
      .... ...1 = NA services wanted: True
    ▼ Connect Flags 1: 0x41
      ...0 .... = NA services required: False
      .... 0... = NA services linked in: False
      .... .0.. = NA services enabled: False
      .... ..0. = Interchange is involved: False
      .... ...1 = NA services wanted: True
    Trace Cross Facility Item 1: 0x00000000
    Trace Cross Facility Item 2: 0x00000000
    Trace Unique Connection ID: 0x0000000000000000
    Connect Data: (DESCRIPTION=(CONNECT_DATA=(SID=orcl11g)(CID=(PROGRAM=sqlplus)(HOST=coanor-ubuntu)(USER=coan

```

如果验证通过，服务器端会返回其 Accept 数据包，Wireshark 的分析结果如图 1.3 所示：

连接建立以后，就可以发送 SQL 请求给服务器端，以下面的 SQL 语句为例：

```
SELECT table_name FROM all_tables;
```

Figure 1.3: Oracle 服务器端发出的 Accept 数据包

```

▶ Frame 409: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on inte
▶ Ethernet II, Src: CadmusCo_3f:43:f8 (08:00:27:3f:43:f8), Dst: Giga-Byt_ef:9a
▶ Internet Protocol Version 4, Src: 192.168.37.194 (192.168.37.194), Dst: 192.
▶ Transmission Control Protocol, Src Port: ncube-lm (1521), Dst Port: 57283 (5
▼ Transparent Network Substrate Protocol
  Packet Length: 32
  Packet Checksum: 0x0000
  Packet Type: Accept (2)
  Reserved Byte: 00
  Header Checksum: 0x0000
  ▼ Accept
    Version: 314
    ▼ Service Options: 0x0c41
      ..0. .... = Broken Connect Notify: False
      ...0 .... = Packet Checksum: False
      ....1... = Header Checksum: True
      .... .1.. = Full Duplex: True
      .... ..0. = Half Duplex: False
      .... ...0 = Don't Care: False
      .... ..0. = Don't Care: False
      .... ....0 = Direct IO to Transport: False
      .... ....0 = Attention Processing: False
      .... ....0.. = Can Receive Attention: False
      .... ....0.. = Can Send Attention: False
      Session Data Unit Size: 8192
      Maximum Transmission Data Unit Size: 32767
      Value of 1 in Hardware: 0100
      Accept Data Length: 0
      Offset to Accept Data: 32
      ▼ Connect Flags 0: 0x41
        ...0 .... = NA services required: False
        .... 0... = NA services linked in: False
        .... .0.. = NA services enabled: False
        .... ..0. = Interchange is involved: False
        .... ...1 = NA services wanted: True
      ▼ Connect Flags 1: 0x41
        ...0 .... = NA services required: False
        .... 0... = NA services linked in: False
        .... .0.. = NA services enabled: False
        .... ..0. = Interchange is involved: False
        .... ...1 = NA services wanted: True
0000  94 de 80 ef 9a 4c 08 00 27 3f 43 f8 08 00 45 00  ....L.. '?C...E.
0010  00 54 ed 8a 40 00 40 06 80 58 c0 a8 25 c2 c0 a8  .T..@.. .X..%...
0020  25 ae 05 f1 df c3 9d 00 6c 33 e7 69 c2 4f 80 18  %..... l3.i.0...
0030  00 82 1a e0 00 00 01 01 08 0a 00 2f 81 ca 00 2f  ...../.../
0040  81 ab 00 20 00 00 02 00 00 00 01 3a 0c 41 20 00  ... ..:..A .
0050  7f ff 01 00 00 00 00 20 41 41 00 00 00 00 00 00  ..... AA.....
0060  00 00

```

其 Wireshark 的分析结果如图 1.4 所示：

从右下角的报文可以看出，其中就包含有查询语句，但是从左下角 Wireshark 的报文分析来看，它并没有识别出该报文。而且经测试，同一条 SQL 语句，其报文结构可能不同，比如，再次输入上面的 SQL 语句，其报文如图 1.5 所示，此处再次列出上面的图片以做对比。<sup>4</sup>

<sup>4</sup>这种不同是有规律的，登陆后第一条查询语句的 payload 首字节为 0x035e，后续的查询都是以 0x1169 开头，后面相隔固定字符之后再追加 0x035e。

Figure 1.4: Oracle 的 SQL 查询语句数据包

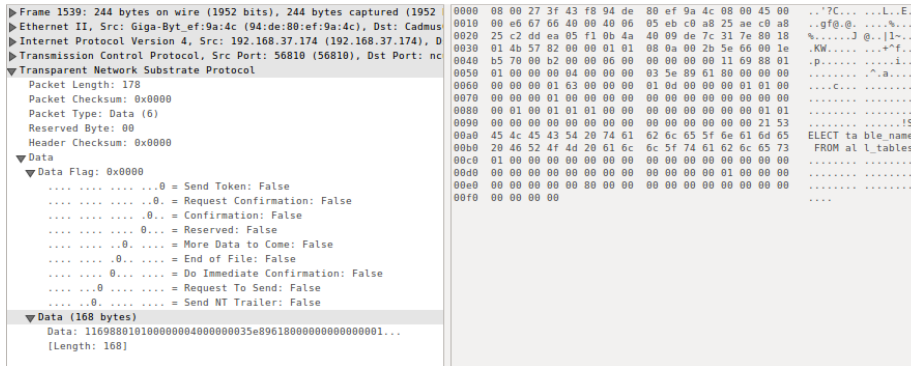
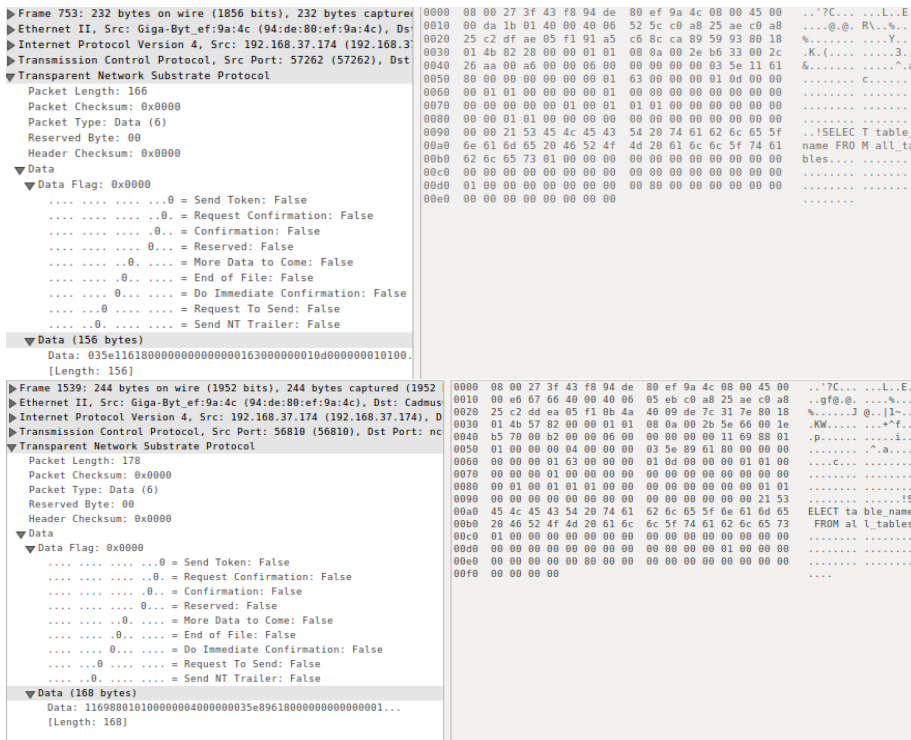


Figure 1.5: Oracle 中同一条 SQL 查询语句出来的数据包可能不同



以上是 Oracle 数据库通信协议的 Wireshark 截图。下面再用 Wireshark 分析一下 MySQL 的通信数据包。

在 MySQL 的客户端与服务器端的通信过程中，先是服务器发送一个握手数据包，然后客户端再发送连接数据包。其通信过程如图 1.6 所示：

Figure 1.6: MySQL 的通信模型

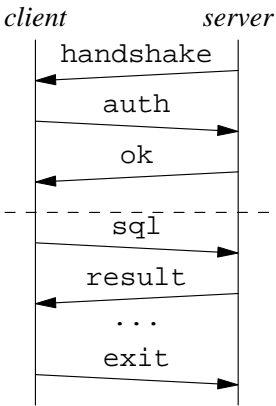
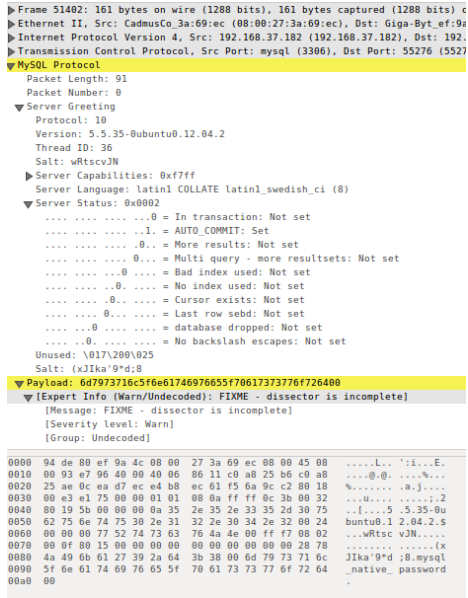


Figure 1.7: MySQL 服务器的握手数据包



Wireshark 能够完整地支持 Mysql 数据包解析，图 1.7 中列举的是一个 MySQL 服务器发送的握手数据包。

然后是客户端的连接数据包，如图 1.8 所示：

在 MySQL 的登陆过程中，会有一个 SQL 语句请求，在 Wireshark 中，其输出如图 1.9 所示：

另外的几种协议，比如 DB2 的 DADR 协议，SQL Server 的 TDS 协议，Wire-

Figure 1.8: MySQL 客户端的连接数据包

```

▶Frame 51404: 150 bytes on wire (1200 bits), 150 bytes captured (1200 bits) on
▶Ethernet II, Src: Giga-Byt_ef:9a:4c (94:de:80:ef:9a:4c), Dst: CadmusCo_3a:6
▶Internet Protocol Version 4, Src: 192.168.37.174 (192.168.37.174), Dst: 192
▶Transmission Control Protocol, Src Port: 55276 (55276), Dst Port: mysql (33
▼MySQL Protocol
  Packet Length: 80
  Packet Number: 1
  ▼Login Request
    Client Capabilities: 0xa605
      ...1 = Long Password: Set
      ...0 = Found Rows: Not set
      ...1 = Long Column Flags: Set
      ...0 = Connect With Database: Not set
      ...0 = Don't Allow database.table.column: Not set
      ...0 = Can use compression protocol: Not set
      ...0 = ODBC Client: Not set
      ...0 = Can Use LOAD DATA LOCAL: Not set
      ...0 = Ignore Spaces before '(': Not set
      ...1 = Speaks 4.1 protocol (new flag): Set
      ...1 = Interactive Client: Set
      ...0 = Switch to SSL after handshake: Not set
      ...0 = Ignore sspipes: Not set
      ...1 = Knows about transactions: Set
      ...0 = Speaks 4.1 protocol (old flag): Not set
      ...1 = Can do 4.1 authentication: Set
    Extended Client Capabilities: 0x000f
      ...1 = Supports multiple statements: Set
      ...1 = Supports multiple results: Set
    MAX Packet: 16777216
    Charset: utf8 COLLATE utf8_general_ci (33)
    Username: root
    Password: 1fdb6eccc21b0a8b2033ba4e7d2dfdba5b74b9d
  ▼Payload: 6d7973716c5f6e61746976655f70617373766f726400
  ▼[Expert Info (Warn/Undecoded): FIXME - dissector is incomplete]
    [Message: FIXME - dissector is incomplete]
    [Severity level: Warn]
    [Group: Undecoded]
0000 00 00 27 3a 69 ec 94 de 80 ef 9a 4c 08 00 45 08  ..'i...L..E.
0010 00 08 1c db 40 00 40 06 50 d8 c0 a8 25 ae c0 a8  ....@.0. P...%...
0020 25 b6 d7 ec 0c ea f3 6a 9c c2 e4 b8 ec c0 00 18  %.....j.....
0030 00 e5 b3 53 00 00 01 01 08 0a 00 32 00 61 ff ff  ...S.....2.a...
0040 0c 3b 50 00 00 01 05 a6 0f 00 00 00 00 01 21 00  .iP.....!.
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....FO 0t.....
0060 00 00 00 00 00 00 72 6f 6f 74 00 14 1f db c0 ec  .....f.....K.
0070 cc 21 b0 a8 b2 03 3b a4 e7 d2 df db a5 b7 4b 9d  .....2.....K.
0080 6d 79 73 71 6c 5f 6e 61 74 69 76 65 5f 70 61 73  mysql_na tive_pas
0090 73 77 6f 72 64 00                                sword.

```

Figure 1.9: MySQL 查询语句数据包

```

▶Frame 51407: 103 bytes on wire (824 bits), 103 bytes captured (824 bits) on Interface 0
▶Ethernet II, Src: Giga-Byt_ef:9a:4c (94:de:80:ef:9a:4c), Dst: CadmusCo_3a:69:ec (80:00:27:3a:69:ec)
▶Internet Protocol Version 4, Src: 192.168.37.174 (192.168.37.174), Dst: 192.168.37.182 (192.168.37.182)
▶Transmission Control Protocol, Src Port: 55276 (55276), Dst Port: mysql (3306), Seq: 85, Ack: 187, Len: 37
▼MySQL Protocol
  Packet Length: 33
  Packet Number: 0
  ▼Request Command Query
    Command: Query (3)
    Statement: select @version,comment limit 1
0000 00 00 27 3a 69 ec 94 de 80 ef 9a 4c 08 00 45 08  ..'i...L..E.
0010 00 50 1c dc 40 00 40 06 51 06 c0 a8 25 ae c0 a8  ....Y.@.0.~...%...
0020 25 b6 d7 ec 0c ea f3 6a 9c c2 e4 b8 ec c0 00 18  %.....j.....
0030 00 e5 b3 53 00 00 01 01 08 0a 00 32 00 61 ff ff  ...S.....2.a...
0040 0c 3b 50 00 00 01 05 a6 0f 00 00 00 00 01 21 00  .iP.....!.
0050 00 00 00 00 00 00 72 6f 6f 74 00 14 1f db c0 ec  .....f.....K.
0060 6d 79 73 71 6c 5f 6e 61 74 69 76 65 5f 70 61 73  mysql_na tive_pas
0070 73 77 6f 72 64 00                                sword.

```

shark 都有完整的支持，只需要按照 Wireshark 中的数据包分段即可将各种数据库的通信数据包抓取过来并进行分析。

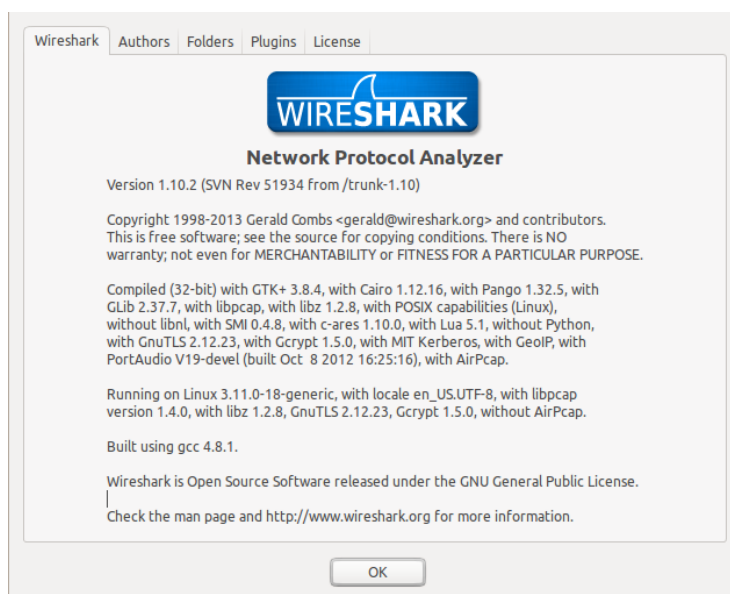


## Chapter 2

# 数据库通信协议分析

本章主要通过使用 **Wireshark** 对各种数据库的协议进行分析，以便在 **Suricata** 中加入协议分析代码，实现最终的 **IDS/IPS** 功能。其中涉及的协议有 **MySQL**、**DB2**、**SQL Server** 以及 **Oracle**。由于 **Wireshark** 已经完整支持了前三者的通信协议，本章着重分析 **Oracle** 的 **TNS** 协议。

为便于后续维护，以避免不同 **Wireshark** 版本带来的差异，现列出本版本的 **Wireshark** 帮助信息。



2.1 Oracle 数据包协议分析

由于 Wireshark 对 Oracle 数据包的支持不够，所以需要做一些反向工作，将 Oracle 中的 TNS 协议以及 Net8 协议分析出来。<sup>1</sup>。

本文 Oracle 客户端使用的 sqlplus 版本为 *SQL\*Plus: Release 12.1.0.1.0 Production*，服务器端的版本为 *Oracle 11g R2*。其它版本的分析结果可能会有所差异，需后续跟进。

Oracle 数据库操作也分为三个步骤，首先需要登陆，这期间客户端和服务端交换一些通信数据，比如版本、用户认证、Session 注册等等。登陆成功后，才能对数据库中的数据进行操作。操作完成后，再退出登陆。完成一次数据库交互。

Oracle 使用的 TNS 协议是私有协议，官方并未将协议的 Spec 放出来。但是各路大侠们捕风捉影，也能拿到一些靠谱的信息。对一个 TNS 数据包而言，它由两部分组成，一部分是头部数据，长度为 8 个字节（如果加上载荷上前面无用的 2 个字节，就是 10 个字节），其中包括长度、类型等信息。另一部分为载荷数据，其中包含了具体的数据包内容。本节主要分析其载荷数据。因为头部数据在 Wireshark 中已经可以完整查看。

关于头部的数据包类型字段，现收集如表 2.1 所示，以供参考。

Table 2.1: Oracle TNS 协议中数据类型列表

字节	意义	字节	意义	字节	意义
0x01	Connect	0x06	Data	0x11	Resend
0x02	Accept	0x07	NULL	0x12	Marker
0x03	ACK	0x08	—	0x13	Attention
0x04	Refuse	0x09	Abort	0x14	Control
0x05	Redirect	0x10	—		

对 TNS 协议中的载荷数据而言，它也有不同的类型，该类型字段位于载荷数据包的前面两个字节（即应用层数据包偏移 10 个字节），其类型如表 2.2 所示：

其中对 0x11 类型的数据而言，其后面会紧跟以下几种类型的数据：

- 0x6b: switch to detach session
- 0x78: close
- 0x87: OSCID
- 0x9a: OKEYVAL

对 0x03 类型的载荷而言，其后面所紧跟的种类在表 2.3 中列出。

<sup>1</sup>关于 Oracle 数据库的通信模型，[这里](#)有一个它的 SQL\*Net 文档。



Table 2.2: Oracle TNS 协议中载荷数据包类型

字节	意义
0x01	Protocol Negotiation. Following this flag are acceptable protocol versions 0x060504030201 and client platform string like IBMPC/WIN_NT-8.1.0
0x02	Exchange of Data type representations.
0x03	TTI (Two-Task Interface) Function call. The exact function id comes immediately after data packet id.
0x08	“OK” server to client response
0x11	Extended TTI (Two-Task Interface) Function call.
0x20	Used by external procedures and service registrations
0x44	Used by external procedures and service registrations
0xdeadbeef	Additional Network Options. Client may negotiate additional connection attributes such as authentication, encryption, data integrity, and supervisor.

### 2.1.1 Oracle 登陆数据包的组成

对登陆而言，其来往的数据包多达几十条，除了登陆数据包之外（类型为 0x01），客户端发往服务器端的数据包（类型为 0x06）还有如下几种，其中的字节列举的是 **payload** 中的首部字节。

1. 客户端发往服务器端的标有“网络连接属性”的数据包 (0xdeadbeef)
2. 协议数据包，客户端需要更服务器端对接数据通信协议 (0x01)
3. 交换“数据表示形式”的数据包 (0x02)
4. 带用户名和密码的两个登陆数据包，此处的用户名和密码放在两个数据包中传送 (0x0376, 0x0373)
5. 确定服务器数据库信息的数据包 (0x116b)
6. 一个带有 SQL 语句的数据包，其中的 SQL 语句为 `SELECT USER FROM DUAL`，该数据包会返回用户当前的登陆用户名 (0x035e)
7. 一个 Fetch 数据包 (0x0305)
8. (0x1169) 发送一个 SQL 语句 `BEGIN DBMS_OUTPUT.DISABLE; END;`
9. (0x1169) 发送一个 SQL 语句 `BEGIN DBMS_APPLICATION_INFO.SET_MODULE (:1, NULL); END;`
10. (0x1169) 发送一个 SQL 语句 `SELECT DECODE ('A', 'A', '1', '2') FROM DUAL;`
11. (0x030e) 发送一个 COMMIT 请求，完成登陆过程。

对服务器端而言，其返回的 **payload** 数据包中如果以 `0x08` 开头，则证明请求通过。对首部为 `0x03` 的登陆数据包而言（即 **TTI**，*two-task interface*），其后面跟着的字节分别表示各种意义，如表 2.3 所示：

Table 2.3: 登陆数据包中以 `0x03` 为首字节的 **TTI** 字节列表

字节	意义	字节	意义	字节	意义
0x02	Open	0x0f	Rollback	0x5c	OKOD
0x03	—	0x14	Cancel	0x5e	Query
0x04	Execute	0x2b	Describe	0x60	LOB Operations
0x05	Fetch	0x30	Startup	0x62	ODNY
0x08	Close	0x31	Shutdown	0x67	Transaction-end
0x09	Disconnect/logoff	0x3b	Version	0x68	Transaction-begin
0x0c	AutoCommit On	0x43	K2 Transactions	0x69	OCCA
0x0d	AutoCommit Off	0x47	Query	0x6d	Startup
0x0e	Commit	0x4a	OSQL7	0x51	Logon
0x52	Logon	0x73	Logon	0x76	Logon
0x77	Describe	0x7f	OOTCM	0x8B	OKPFC

表 2.3 中出现了多个重复意义的字段，它们各自的实际意义如下：

- `0x5e`：指一般的 **SQL** 语句查询
- `0x47`：在现有的分析中没有碰到这种类型的查询字段
- `0x30` 和 `0x60` 虽然都为 **Startup**，但是尚未碰到这种数据包
- `0x51`：带密码的登陆行为
- `0x52`：带用户名的登陆行为
- `0x73`：带密码的登陆行为，其中带有 `AUTH_PASSWORD` 信息
- `0x76`：带用户名的登陆行为，其中带有 `AUTH_SESSKEY` 信息

由于登录数据包中类型为 `0x01`、`0x011` 以及 `0x02` 数据包在 **Wireshark** 中能正确识别，所以就不做详细分析，只列出其基本的组成。对 `0x01` 类型的数据包而言，它是客户端发给服务器端的第一个连接数据包，其字节组成如下

```
TNS connect pkt
|
+- [2B] pkt len
+- [2B] pkt checksum 一般为 0x0000
+- [1B] pkt type
+- [1B] reserved byte
+- [2B] header checksum
+- [2B] version
+- [2B] compitible version
+- [2B] service option，其中有各种 bit 位，详见 Wireshark 结果
+- [2B] session data unit size
+- [2B] maximum transimission data unit size
+- [2B] NT protocol characteristics，各种 bit 位参见 Wireshark 结果
+- [1B] line turnaround value
+- [2B] value of 1 in hardware
```

```

+- [2B] length of connect data
+- [2B] offset to connect data
+- [4B] maximum receivable connect data
+- [1B] connection flags, 各种 bit 位参见 Wireshark 结果
+- [...]
+- [nB] connect data

```

对于用 `sqlplus` 登陆的客户端而言，当客户端第一次发送连接数据包之后，服务器端往往会要求客户端重传（0x11）这次连接数据包。<sup>2</sup> 重传的数据包中，应用层的数据包与前一次的字节全部相同。服务器端验证通过后，会发送 `Accept` 数据包（0x02），其数据包结构为

```

TNS accept pkt
|
+- [2B] pkt len
+- [2B] pkt checksum 一般为 0x0000
+- [1B] pkt type
+- [1B] reserved byte
+- [2B] header checksum
+- [2B] version, 此 version 值可能会低于客户端传送过来的 version 值
+- [2B] service option, 其中有各种 bit 位, 详见 Wireshark 结果
+- [2B] session data unit size
+- [2B] maximum transmission data unit size
+- [2B] value of 1 in hardware
+- [2B] accept data length
+- [2B] offset to accept data
+- [2B] connect flag, 在 Wireshark 中看来是两个相同的字节, 意义不大

```

接下来，在 `Wireshark` 中会看到两个 `SNS` 数据包，它们之间交换一些数据之后就进入正式的登陆流程。此处 `SNS` 数据包的类型也是 0x06，其判断依据是检查 `payload` 上前面四个字节，如果为 0xdeadbeef，则为 `SNS` 数据包。

### 2.1.2 Oracle 查询数据包的组成

从目前所观察的数据包来看，查询数据包是有一定规律的，如果只提取客户端的查询字符串，而不涉及 `session` 管理以及结果集的提取，那么这个过程是比较简单的。

首先，对于登陆后出现的第一个查询语句，其数据包结构是特殊的，其后续的查询数据包则以另外一种形态保持一致。下面以一系列 `SQL` 语句为例，来说明这个过程。

Listing 2.1: h-query.sql

```

1  --
2  -- Employee Tree: who is my manager.
3  -- This is a hierarchical query with ident
4  --
5  CREATE TABLE emp (
6      empno NUMBER(4) constraint E_PK primary key,
7      ename VARCHAR2(8),
8      init VARCHAR2(5),
9      job VARCHAR2(8),
10     mgr NUMBER(4),

```

<sup>2</sup>重传的目的尚不明确。在老版本 `Oracle` 的协议中（`Oracle 9i`），`Server` 端会将客户端定位到另一个端口进行通信，而不是默认的 1521。在最近的版本中，已经移除了这个设定。

```

11      bdate DATE,
12      sal NUMBER(6,2),
13      comm NUMBER(6,2),
14      deptno NUMBER(2) DEFAULT 10);
15
16 INSERT INTO emp VALUES (1,'Tom','N','TRAINER',12,DATE'1965-12-17',800,NULL,20);
17 INSERT INTO emp VALUES (2,'Jack','Jam','TRAINER',6,DATE'1961-02-20',1600,300,20);
18 INSERT INTO emp VALUES (3,'Wil','TF','Tester',6,DATE'1962-02-22',1250,500,30);
19 INSERT INTO emp VALUES (4,'Jane','JM','Designer',9,DATE'1967-04-02',2975,NULL,20);
20 INSERT INTO emp VALUES (5,'Mary','P','Tester',6,DATE'1956-09-28',1250,1400,30);
21 INSERT INTO emp VALUES (6,'Black','R','Designer',9,DATE'1963-11-01',2850,NULL,30);
22 INSERT INTO emp VALUES (7,'Chris','AB','Designer',9,DATE'1965-06-09',2450,NULL,10);
23 INSERT INTO emp VALUES (8,'Smart','SCJ','TRAINER',4,DATE'1959-11-26',3000,NULL,20);
24 INSERT INTO emp VALUES (9,'Peter','CC','Designer',NULL,DATE'1952-11-17',5000,NULL,10);
25 INSERT INTO emp VALUES (10,'Take','JJ','Tester',6,DATE'1968-09-28',1500,0,30);
26 INSERT INTO emp VALUES (11,'Ana','AA','TRAINER',8,DATE'1966-12-30',1100,NULL,20);
27 INSERT INTO emp VALUES (12,'Jane','R','Manager',6,DATE'1969-12-03',800,NULL,30);
28 INSERT INTO emp VALUES (13,'Fake','MG','TRAINER',4,DATE'1959-02-13',3000,NULL,20);
29 INSERT INTO emp VALUES (14,'Mike','TJA','Manager',7,DATE'1962-01-23',1300,NULL,10);
30
31 SELECT lpad(' ', 2 * level - 1) || ename as ename
32 FROM emp
33 START WITH mgr IS NULL
34 CONNECT BY nocycle PRIOR empno = mgr;
35
36 DROP TABLE emp;

```

将这些 SQL 存为文件 h-query.sql。登陆 Oracle 数据库库之后，在 sqlplus 控制台输入如下命令则执行该文件：

```
SQL> @h-query.sql
```

在 Wireshark 中，第一个数据包如图 2.1 所示：

橙色阴影下的数据即为应用层数据，除去前面的 10 个字节（头部 8 字节以及 2 字节的数据 flag 字段），就是 CREATE TABLE 命令的数据包载荷。

最先出现的字节是 0x035e，从表 2.3 可以得知，字节 0x5e 代表一个 Query 操作，后面的 0x11 代表数据包序列号，这点可以从后续的 INSERT 语句的载荷数据包中看出来，只是该序列号不是逐个递增的，而是间隔递增的，即后续的 Query 数据包的序列号为 0x13、0x15、0x17、…。至于其原因，在后面提到时再说明。

再往后的字节尚不知道其具体含义。从序列号字段开始 (payload + 3)，跳过 67 个字节之后，到达 0090 这一行的第三个字节 0x4f，它就是表示后面查询字符串的长度。该字节之后，就是具体的查询字符串，也即 CREATE TABLE 语句。

经测试，当 SQL 长度小于等于 0xfc 时 (252)，该字节就是后面数据包的长度。当 SQL 语句的长度超过 253 时，那么该字节固定为 0xfe，后面紧跟的那个字节才是 SQL 的长度。此处 0xfe 是一个标记字符，它说明此时 SQL 语句的长度已经大于 0xfc，例如当数据包长度为 0xfd 时，此时的数据包为

```

+-----+-----+-----+
| 0xfe | 0xfd | sql stmt |
+-----+-----+-----+

```

Figure 2.1: 登陆后第一条 SQL 语句的数据包

0000	08 00 27 3f 43 f8 94 de	80 ef 9a 4c 08 00 45 00	..?C... ..L..E.
0010	01 ad 1b e9 40 00 40 06	50 a1 c0 a8 25 ae c0 a8	....@. P...%...
0020	25 c2 bd 32 05 f1 e1 cc	32 e7 6a 75 34 cc 80 18	%..2.... 2.ju4...
0030	01 4b 80 f5 00 00 01 01	08 0a 00 01 27 bc 00 01	.K.....'....
0040	40 8e 01 79 00 00 06 00	00 00 00 00 03 5e 11 21	@..y.... ..^..!
0050	80 00 00 00 00 00 00 01	dc 02 00 00 01 0d 00 00	.....
0060	00 01 01 00 00 00 00 01	00 00 00 00 00 00 00 00	.....
0070	00 00 00 00 00 01 00 01	01 01 00 00 00 00 00 00	.....
0080	00 00 01 01 00 00 00 00	00 00 00 00 00 00 00 00	.....
0090	00 00 f4 43 52 45 41 54	45 20 54 41 42 4c 45 20	...CREAT E TABLE
00a0	65 6d 70 20 28 0a 20 20	20 20 65 6d 70 6e 6f 20	emp (. empno
00b0	4e 55 4d 42 45 52 28 34	29 20 63 6f 6e 73 74 72	NUMBER(4 ) constr
00c0	61 69 6e 74 20 45 5f 50	4b 20 70 72 69 6d 61 72	aint E P K primar
00d0	79 20 6b 65 79 2c 0a 20	20 20 20 65 6e 61 6d 65	y key,, ename
00e0	20 56 41 52 43 48 41 52	32 28 38 29 2c 0a 20 20	VARCHAR 2(8),.
00f0	20 20 69 6e 69 74 20 56	41 52 43 48 41 52 32 28	init V ARCHAR2(
0100	35 29 2c 0a 20 20 20 20	6a 6f 62 20 56 41 52 43	5),. job VARC
0110	48 41 52 32 28 38 29 2c	0a 20 20 20 20 6d 67 72	HAR2(8), mgr
0120	20 4e 55 4d 42 45 52 28	34 29 2c 0a 20 20 20 20	NUMBER( 4),.
0130	62 64 61 74 65 20 44 41	54 45 2c 0a 20 20 20 20	bdate DA TE,.
0140	73 61 6c 20 4e 55 4d 42	45 52 28 36 2c 32 29 2c	sal NUMB ER(6,2),
0150	0a 20 20 20 20 63 6f 6d	6d 20 4e 55 4d 42 45 52	. com m NUMBER
0160	28 36 2c 32 29 2c 0a 20	20 20 20 64 65 70 74 6e	(6,2), deptn
0170	6f 20 4e 55 4d 42 45 52	28 32 29 20 44 45 46 41	o NUMBER (2) DEFA
0180	55 4c 54 20 31 30 29 01	00 00 00 01 00 00 00 00	ULT 10). .....
0190	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
01a0	00 00 00 05 00 00 00 00	00 00 00 00 80 00 00 00	.....
01b0	00 00 00 00 00 00 00 00	00 00 00	.....

当 SQL 的长度超过 0xfe (254) 时, 此时这两个字节都不能用来断定后续 SQL 长度, 因为此时这两个字节的长度固定为 0xfeff, 无法用它来断定后续的长度是 0xff 还是大于 0xff。此时, TNS 协议对数据包做了特殊处理, 即如果 SQL 语句长度刚好 255 个字节, 那就直接拷贝这 255 个字节, 如果大于 255 字节, 则每隔 255 个字节会在数据包中插入一个分隔字节 (例如 0x80, 但该分隔字节不固定, 会有变动), 此时, 通过总长度以及固定位置的分隔符, 即可将被分割的 SQL 语句提取出来。

另一点值得注意的是, 此处提到的 SQL 语句长度并不是输入 SQL 语句的长度, 在 sqlplus 中会对 SQL 语句做预处理, 目前看来至少去掉了其中不必要的空白字符。而且 sqlplus 会对 SQL 语句做基本的 SQL 解析, 不合语法的 SQL 语句不会发送到服务器端。

以上是登陆后第一条查询语句的数据包分析。接下来分析第一条查询之后的其他查询语句的数据包。这种数据包如图 2.2 所示。

从图 2.2 中可以看出, 其载荷中的首部字节变成了 0x1169, 从表 2.2 可以得知, 这是一个扩展的 TTI 数据包, 其中 0x69 的意义为 **Cursor Close All**, 此处对数据包的分析意义不大, 但是后面紧跟着的是数据包的序列号 0x12。如何解释呢? 此处的意义即这是一条独立计数的数据包。截止到字节 0x035e 为止, 其长度为 12 字节。从 0x035e 开始, 又开始了一条独立计数的数据包, 其序列号为 0x13, 这就解释了上面 CREATE TABLE 数据包中所提到的序列号间隔的问题。

Figure 2.2: 第一条 SQL 语句之后的数据包

0000	08 00 27 3f 43 f8 94 de	80 ef 9a 4c 08 00 45 00	.. '?C... ..L..E.
0010	01 12 1b eb 40 00 40 06	51 3a c0 a8 25 ae c0 a8	....@.@. Q:..%...
0020	25 c2 bd 32 05 f1 e1 cc	34 60 6a 75 35 3b 80 18	%..2.... 4'ju5;...
0030	01 4b 50 f4 00 00 01 01	08 0a 00 01 28 7f 00 01	.KP..... ..({...
0040	c4 f3 00 de 00 00 06 00	00 00 00 00 11 69 12 01	.. ..1..
0050	01 00 00 00 03 00 00 00	03 5e 13 21 80 00 00 00	..... ^!....
0060	00 00 00 01 e7 00 00 00	01 0d 00 00 00 01 01 00	.....
0070	00 00 00 01 00 00 00 00	00 00 00 00 00 00 00 00	.....
0080	00 01 00 01 01 01 00 00	00 00 00 00 00 00 01 01	.....
0090	00 00 00 00 00 00 00 00	00 00 00 00 00 00 4d 49	.....MI
00a0	4e 53 45 52 54 20 49 4e	54 4f 20 65 6d 70 20 56	INSERT IN TO emp V
00b0	41 4c 55 45 53 28 31 2c	27 54 6f 6d 27 2c 27 4e	ALUES(1, 'Tom','N
00c0	27 2c 27 54 52 41 49 4e	45 52 27 2c 31 32 2c 44	','TRAIN ER',12,D
00d0	41 54 45 27 31 39 36 35	2d 31 32 2d 31 37 27 2c	ATE'1965 -12-17',
00e0	38 30 30 2c 4e 55 4c 4c	2c 32 30 29 01 00 00 00	800,NULL ,20)....
00f0	01 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
0100	00 00 00 00 00 00 00 00	04 00 00 00 00 00 00 00	.....
0110	00 80 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....

在 0x035e13 之后，同样跳过 67 个字节，即可拿到后面 SQL 语句的长度字节 0x4f，再之后就是具体的 INSERT 语句。

至此，基本的客户端 SQL 语句的数据包分析完毕。接下来是一些其它信息的数据包分析。

注意到，在上面的批量 SQL 语句中，第二个 INSERT 语句中的主键与第一条 INSERT 语句是重复的，所以服务器端会报错，在我们的 Wireshark 中，肯定不会错过这一幕，对此，服务器会发送两种数据包，一种是 Attention 数据包，一种是常规的类型为 0x06 的数据包，其中带有错误信息。这两个数据包分别如图 2.3 以及图 2.4 所示：

Figure 2.3: Oracle TNS 的 Attention 数据包

Transparent Network Substrate Protocol			
Packet Length: 11			
Packet Checksum: 0x0000			
Packet Type: Marker (12)			
Reserved Byte: 00			
Header Checksum: 0x0000			
Attention			
Marker Type: Data Marker - 1 Data Bytes (0x01)			
Marker Data Byte: 0x00			
Marker Data Byte: 0x01			
0000	94 de 80 ef 9a 4c 08 00	27 3f 43 f8 08 00 45 00	....L.. '?C...E.
0010	00 3f 5c bd 40 00 40 06	11 3b c0 a8 25 c2 c0 a8	.?\.@.@. ;;...%
0020	25 ae 05 f1 bd 32 6a 75	35 b8 e1 cc 36 1e 80 18	%...2ju 5...6...
0030	01 4a 31 b8 00 00 01 01	08 0a 00 01 c5 1d 00 01	.Jl.....
0040	28 80 00 0b 00 00 0c 00	00 00 01 00 01	(. ....

因为 Attention 数据包中并无实质性的数据，它顶多只能算是一个暗示。后面的 Error Message 数据包才是有价值的。

Figure 2.4: Oracle TNS Error Message 数据包

0000	94 de 80 ef 9a 4c 08 00	27 3f 43 f8 08 00 45 00	....L.. '?C...E.
0010	00 b5 5c bf 40 00 40 06	10 c3 c0 a8 25 c2 c0 a8	..\.@. ....%...
0020	25 ae 05 f1 bd 32 6a 75	35 ce e1 cc 36 29 80 18	%...2ju 5...6)..
0030	01 4a a6 1b 00 00 01 01	08 0a 00 01 c5 1d 00 01	.J.....
0040	28 89 00 81 00 00 06 00	00 00 00 00 04 02 00 00	(.....
0050	00 13 00 00 00 00 00 01	00 00 00 00 00 08 00 00	.....
0060	00 02 00 00 00 00 00 ff	ff ff ff 00 00 00 00 00	.....
0070	00 00 00 00 00 00 00 00	00 15 00 00 00 00 00 00	.....
0080	00 00 00 00 00 00 00 00	00 00 00 00 00 00 34 4f	.....40
0090	52 41 2d 30 30 30 30 31	3a 20 75 6e 69 71 75 65	RA-00001 : unique
00a0	20 63 6f 6e 73 74 72 61	69 6e 74 20 28 53 59 53	constra int (SYS
00b0	54 45 4d 2e 45 5f 50 4b	29 20 76 69 6f 6c 61 74	TEM.E PK ) violat
00c0	65 64 0a		ed.

此处载荷的头部字节为 0x0402，可以确定是一个“错误信息”，其具体的错误描述则在 64 个字节之后。0080 这一行的倒数第二个字节 0x34，即为错误信息的长度，后面紧跟的就是具体的错误信息。从描述中，可以得知，此处违反了主键的唯一性约束。

接下来分析一下 SELECT 语句的数据包，由于 SELECT 语句有一个获取结果的过程，所以会出现一个 **Fetch** 数据包。在上面的批量 SQL 中，有一个 SELECT 语句，其数据包和 INSERT 语句没有实质性的差别，关键是其后面紧跟的 **Fetch** 操作。其数据包如图 2.5 所示。

Figure 2.5: Oracle TNS Fetch 数据包

0000	08 00 27 3f 43 f8 94 de	80 ef 9a 4c 08 00 45 00	..'?C... ..L..E.
0010	00 49 1c 0f 40 00 40 06	51 df c0 a8 25 ae c0 a8	.I..@. Q...%...
0020	25 c2 bd 32 05 f1 e1 cc	50 2b 6a 75 44 e8 80 18	%..2.... P+juD...
0030	01 4b c5 ee 00 00 01 01	08 0a 00 1f f1 8b 00 7c	.K.....
0040	f2 cf 00 15 00 00 06 00	00 00 00 00 03 05 50 0c	.....P.
0050	00 00 00 0f 00 00 00		.....

此处的载荷数据包头部字节为 0x0305，从表 2.3 可以得知，这是一个 **Fetch** 操作，即 SELECT 语句之后获取查询结果的操作。这个操作之后，服务器会返回 SELECT 的结果，如图 2.6 所示。

上面提到了一个数据包序列号的问题，目前来看，只有一个字节用来保存数据包的序列号，如果上面的 INSERT 语句超过 256 个，那么这个序列号会回滚到 0 么？可以测试一下，写个 Python 脚本，生成这么堆 INSERT 语句：

Listing 2.2: gen-long-insert.py

```

1 f = open('long-insert.sql', 'w+')
2
3 # create table only include primary key
4 f.write('CREATE TABLE emp (empno NUMBER(4) constraint E_PK primary key);\n')
5
6 i = 0
7 while i < 256:
8     f.write('INSERT INTO emp VALUES(%d);\n' % i)
9     i += 1
10

```



Figure 2.6: Oracle TNS SELECT 结果数据包

0000	94 de 80 ef 9a 4c 08 00	27 3f 43 f8 08 00 45 00	.....L.. '?C...E.
0010	01 6d 5c e0 40 00 40 06	0f ea c0 a8 25 c2 c0 a8	.m\.@. ....%...
0020	25 ae 05 f1 bd 32 6a 75	44 e8 e1 cc 50 40 80 18	%....2ju D...P@..
0030	01 4a ad 75 00 00 01 01	08 0a 00 7c f2 e0 00 1f	.J.u....  ....
0040	f1 8b 01 39 00 00 06 00	00 00 00 00 06 02 01 00	..9....
0050	00 00 00 00 0f 00 00 00	00 00 00 00 00 00 00 00	.....
0060	00 00 07 07 20 20 20 4a	61 6e 65 15 01 00 01 07	.... J ane.....
0070	0a 20 20 20 20 20 53 6d	61 72 74 15 01 00 01 07	. Sm art.....
0080	0a 20 20 20 20 20 20 20	41 6e 61 15 01 00 01 07	. Ana.....
0090	09 20 20 20 20 20 46 61	6b 65 15 01 00 01 07 08	. Fa ke.....
00a0	20 20 20 42 6c 61 63 6b	15 01 00 01 07 09 20 20	Black .....
00b0	20 20 20 4a 61 63 6b 15	01 00 01 07 08 20 20 20	Jack. ....
00c0	20 20 57 69 6c 15 01 00	01 07 09 20 20 20 20 20	Wil. ....
00d0	4d 61 72 79 15 01 00 01	07 09 20 20 20 20 20 54	Mary..... T
00e0	61 6b 65 15 01 00 01 07	09 20 20 20 20 20 4a 61	ake..... Ja
00f0	6e 65 15 01 00 01 07 0a	20 20 20 20 20 20 20 54	ne..... T
0100	6f 6d 15 01 00 01 07 08	20 20 20 43 68 72 69 73	om..... Chris
0110	15 01 00 01 07 09 20 20	20 20 20 4d 69 6b 65 04	..... Mike.
0120	02 00 00 00 4e 00 0e 00	00 00 7b 05 00 00 00 00	....N.... {....
0130	0c 00 00 00 03 00 20 00	00 00 27 24 01 00 01 00	..... '\$....
0140	00 59 4e 01 00 0d 00 00	00 00 00 00 50 00 00 01	.YN..... P...
0150	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
0160	00 19 4f 52 41 2d 30 31	34 30 33 3a 20 6e 6f 20	..ORA-01 403: no
0170	64 61 74 61 20 66 6f 75	6e 64 0a	data fou nd.

```

11 f.write('DROP TABLE emp;\n')
12
13 f.close()

```

用 **sqlplus** 将这几百个 **INSERT** 语句喂给 **Oracle** 服务器之后，能拿到一大堆的数据包，其中，在第 214 条 **INSERT** 语句中，其序列号达到了 0xff。<sup>3</sup> 其前后数据包如图 2.7 所示。

在第一个数据包中，0x1169 后面的序列号字节为 0xff，到顶了，下一个 0x035e 的数据包序列号回滚到了 0x00。此即证明这种回滚的猜想是正确的，从其后面的那个数据包中，这一猜想也能得到印证。费了这么大劲搞序列号干什么用呢？

由于序列号的存在，使得客户端的请求必须按照“拿号”的顺序来，中间的插入者不能直接获取到这个序列号，有点类似于 **TCP** 中的 **ACK**，虽然此处服务器端并未将序列号传回给客户端，但是单边的序列号在一定程度上让服务器有一个验证依据。这期间发生的篡改也能被 **IPS** 所检测到。

### 2.1.3 Oracle 中其它数据包分析

为验证以上分析的严谨性，现收集一些其它数据包进行分析。

Listing 2.3: update.sql

```
1 CREATE TABLE emp (
```

<sup>3</sup>因为前面的测试数据包已经“占用”了一部分序列号。



Figure 2.7: Oracle TNS 数据包序列号回滚现场

0000	08 00 27 3f 43 f8 94 de	80 ef 9a 4c 08 00 45 00	..?C... ..L..E.
0010	00 e0 1c e9 40 00 40 06	50 6e c0 a8 25 ae c0 a8	....@.@. Pn..%...
0020	25 c2 bd 32 05 f1 e1 cc	e0 5c 6a 75 af 00 00 18	%..2.... \ju....
0030	01 4b d3 f4 00 00 01 01	08 0a 00 25 e1 93 00 94	.K..... %....
0040	b4 e0 00 ac 00 00 06 00	00 00 00 00 11 69 ff 01	.....i..
0050	01 00 00 00 09 00 00 00	03 5e 00 21 80 00 00 00	.....^!....
0060	00 00 00 01 51 00 00 00	01 0d 00 00 00 01 01 00	....Q.....
0070	00 00 00 01 00 00 00 00	00 00 00 00 00 00 00 00	.....
0080	00 01 00 01 01 01 00 00	00 00 00 00 00 00 01 01	.....
0090	00 00 00 00 00 00 00 00	00 00 00 00 00 00 1b 49	.....I
00a0	4e 53 45 52 54 20 49 4e	54 4f 20 65 6d 70 20 56	INSERT IN TO emp V
00b0	41 4c 55 45 53 28 32 31	33 29 01 00 00 00 01 00	ALUES(21 3).....
00c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
00d0	00 00 00 00 00 00 04 00	00 00 00 00 00 00 00 80	.....
00e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....

0000	08 00 27 3f 43 f8 94 de	80 ef 9a 4c 08 00 45 00	..?C... ..L..E.
0010	00 e0 1c ea 40 00 40 06	50 6d c0 a8 25 ae c0 a8	....@.@. Pm..%...
0020	25 c2 bd 32 05 f1 e1 cc	e1 08 6a 75 af 7d 80 18	%..2.... ju.}..
0030	01 4b cb cb 00 00 01 01	08 0a 00 25 e1 93 00 94	.K..... %....
0040	b4 e1 00 ac 00 00 06 00	00 00 00 00 11 69 01 01	.....i..
0050	01 00 00 00 0b 00 00 00	03 5e 02 21 80 00 00 00	.....^!....
0060	00 00 00 01 51 00 00 00	01 0d 00 00 00 01 01 00	....Q.....
0070	00 00 00 01 00 00 00 00	00 00 00 00 00 00 00 00	.....
0080	00 01 00 01 01 01 00 00	00 00 00 00 00 00 01 01	.....
0090	00 00 00 00 00 00 00 00	00 00 00 00 00 00 1b 49	.....I
00a0	4e 53 45 52 54 20 49 4e	54 4f 20 65 6d 70 20 56	INSERT IN TO emp V
00b0	41 4c 55 45 53 28 32 31	34 29 01 00 00 00 01 00	ALUES(21 4).....
00c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
00d0	00 00 00 00 00 00 04 00	00 00 00 00 00 00 00 80	.....
00e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....

```

2      id          NUMBER PRIMARY KEY,
3      fname VARCHAR2(50),
4      lname  VARCHAR2(50)
5  );
6
7  INSERT INTO emp (id, fname, lname)VALUES (1, 'A', 'B');
8  INSERT INTO emp (id, fname, lname)VALUES (2, 'C', 'D');
9  INSERT INTO emp (id, fname, lname)VALUES (3, 'Enn', 'F');
10 INSERT INTO emp (id, fname, lname)VALUES (4, 'G', 'H');
11 INSERT INTO emp (id, fname, lname)VALUES (5, 'G', 'Z');
12
13 SET SERVEROUTPUT ON
14 DECLARE
15 v_rowid ROWID;
16 v_rowcount NUMBER := 0;
17
18 CURSOR emp_cur1 IS SELECT rowid FROM emp WHERE id > 50;
19 CURSOR emp_cur2 IS SELECT rowid FROM emp WHERE id > 50;
20
21 BEGIN
22 OPEN emp_cur1;
23 DELETE FROM emp WHERE id > 50;
24 OPEN emp_cur2;
25 FETCH emp_cur1 INTO v_rowid;
26 IF emp_cur1%ROWCOUNT > 0
27 THEN
28     DBMS_OUTPUT.PUT_LINE('Cursor 1 includes the deleted rows');
29 ELSE
30     DBMS_OUTPUT.PUT_LINE('Cursor 1 does not include the deleted rows');

```

```

31     END IF;
32
33     v_rowcount := 0;
34
35     FETCH emp_cur2 INTO v_rowid;
36     IF emp_cur2%ROWCOUNT > 0
37     THEN
38         DBMS_OUTPUT.PUT_LINE('Cursor 2 includes the deleted rows');
39     ELSE
40         DBMS_OUTPUT.PUT_LINE('Cursor 2 does not include the deleted rows');
41     END IF;
42
43     CLOSE emp_cur1;
44     CLOSE emp_cur2;
45
46     ROLLBACK;
47
48     EXCEPTION
49     WHEN OTHERS
50     THEN
51         DBMS_OUTPUT.PUT_LINE(sqlerrm);
52 END;
53 /
54
55 DROP TABLE emp;

```

对于第一个 CREATE TABLE 语句而言，其遵循前面的分析，载荷数据以 0x1169 开头，后面再跟进另一个数据包，以 0x035e 开头。对第 13 行的 SET 语句而言，其在数据包中的字节并不是 SQL 语句，其数据包如图 2.8 所示。

Figure 2.8: Oracle TNS 数据包与 SQL 语句不一致现场

0000	08 00 27 3f 43 f8 94 de	80 ef 9a 4c 08 00 45 00	..?C... ..L..E.
0010	00 e9 1d 72 40 00 40 06	4f dc c0 a8 25 ae c0 a8	...r@.@. 0...%...
0020	25 c2 bd 32 05 f1 e1 cd	45 7f 6a 75 ee 50 80 18	%..2.... E.ju.P..
0030	01 4b 48 7c 00 00 01 01	08 0a 00 42 d0 de 01 08	.KH .... ..B....
0040	7b 31 00 b5 00 00 06 00	00 00 00 00 11 69 f2 01	{1..... ..i...
0050	01 00 00 00 0a 00 00 00	03 5e f3 21 00 04 00 00	..... ^!.....
0060	00 00 00 01 6c 00 00 00	01 0d 00 00 00 01 01 00	....l.....
0070	00 00 00 01 00 00 00 00	00 00 00 00 00 00 00 00	.....
0080	00 01 00 01 01 01 00 00	00 00 00 00 00 00 01 01	.....
0090	00 00 00 00 00 00 00 00	00 00 00 00 00 00 24 42	.....\$B
00a0	45 47 49 4e 20 44 42 4d	53 5f 4f 55 54 50 55 54	EGIN DBM S_OUTPUT
00b0	2e 45 4e 41 42 4c 45 28	4e 55 4c 4c 29 3b 20 45	.ENABLE( NULL); E
00c0	4e 44 3b 01 00 00 00 01	00 00 00 00 00 00 00 00	ND;.....
00d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 08	.....
00e0	00 00 00 00 00 00 00 00	80 00 00 00 00 00 00 00	.....
00f0	00 00 00 00 00 00 00 00		.....

从数据包中的字节来看，实际传送给服务器的是这样一段 SQL 语句

```
BEGIN DBMS_OUTPUT.ENABLE (NULL); END;
```

除此之外，字段偏移以及序列号，均遵循之前的分析。如果按照上文的做法，此处对 SQL 语句的记录可能会有偏差，虽然其结果可能是一样的。可以单独用这个 BEGIN 语句测试一下其数据包的情况。

通过数字节，发现这个数据包也遵循前面的分析结果。所以此处的第 13 行 SQL 语句是经 sqlplus 转换后再发送给服务器的。

Figure 2.9: Oracle TNS 被转换后的 SQL 语句数据包

0000	08 00 27 3f 43 f8 94 de	80 ef 9a 4c 08 00 45 00	..'?C... ..L..E.
0010	00 ee 1d 84 40 00 40 06	4f c5 c0 a8 25 ae c0 a8	...@. 0...%...
0020	25 c2 bd 32 05 f1 e1 cd	50 90 6a 75 f6 99 80 18	%..2.... P.ju....
0030	01 4b 85 ed 00 00 01 01	08 0a 00 4c fe b1 01 2f	.K..... ..L.../
0040	db 37 00 ba 00 00 06 00	00 00 00 00 11 69 ff 01	.7..... ..1..
0050	02 00 00 00 08 00 00 00	05 00 00 00 03 5e 00 21	..... ^!..
0060	00 04 00 00 00 00 00 01	6f 00 00 00 01 0d 00 00	..... 0.....
0070	00 01 01 00 00 00 00 01	00 00 00 00 00 00 00 00	.....
0080	00 00 00 00 00 01 00 01	01 01 00 00 00 00 00 00	.....
0090	00 00 01 01 00 00 00 00	00 00 00 00 00 00 00 00	.....
00a0	00 00 25 20 42 45 47 49	4e 20 44 42 4d 53 5f 4f	..% BEGI N DBMS 0
00b0	55 54 50 55 54 2e 45 4e	41 42 4c 45 28 4e 55 4c	UTPUT,EN ABLE(NUL
00c0	4c 29 3b 20 45 4e 44 3b	01 00 00 00 01 00 00 00	L); END; .....
00d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
00e0	00 00 00 00 08 00 00 00	00 00 00 00 00 80 00 00	.....
00f0	00 00 00 00 00 00 00 00	00 00 00 00	.....

接下来的数据包发生了异变。如图 2.10 所示。

Figure 2.10: Oracle TNS 载荷数据包再次以 0x035e 开头

0000	08 00 27 3f 43 f8 94 de	80 ef 9a 4c 08 00 45 00	..'?C... ..L..E.
0010	04 40 1d c3 40 00 40 06	4c 34 c0 a8 25 ae c0 a8	..@. 0...%...
0020	25 c2 bd 32 05 f1 e1 cd	89 8a 6a 76 1a 42 80 18	%..2.... ..jv.B..
0030	01 4b 72 07 00 00 01 01	08 0a 00 52 88 1c 01 47	.Kr..... ..R...G
0040	5d 1f 04 0c 00 00 06 00	00 00 00 00 03 5e 4b 21	1..... ^K!
0050	00 04 00 00 00 00 00 01	86 0a 00 00 01 0d 00 00	.....
0060	00 01 01 00 00 00 00 01	00 00 00 00 00 00 00 00	.....
0070	00 00 00 00 00 01 00 01	01 01 00 00 00 00 00 00	.....
0080	00 00 01 01 00 00 00 00	00 00 00 00 00 00 00 00	.....
0090	00 00 fe ff 44 45 43 4c	41 52 45 0a 76 5f 72 6f	....DECL ARE.v_ro
00a0	77 69 64 20 52 4f 57 49	44 3b 0a 76 5f 72 6f 77	wid ROWI D;.v_row
00b0	63 6f 75 6e 74 20 4e 55	4d 42 45 52 20 3a 3d 20	count NU MBER :=
00c0	30 3b 0a 0a 20 20 20 20	43 55 52 53 4f 52 20 65	0;... CURSOR e
00d0	6d 70 5f 63 75 72 31 20	49 53 20 53 45 4c 45 43	mp_curl IS SELEC
00e0	54 20 72 6f 77 69 64 20	46 52 4f 4d 20 65 6d 70	T rowid FROM emp
00f0	20 57 48 45 52 45 20 69	64 20 3e 20 35 30 3b 0a	WHERE i d > 50;..

至于载荷部分的数据再次以 0x035e 开头，我个人的理解是，不是所有的数据包都要以 0x1169 开头。<sup>4</sup> 前面已经提到过，0x1169 也是一种数据包，其操作意义是 *Cursor Close All*。如果当前 SQL 语句的前一条语句没必要执行该操作，则下一条 SQL 语句前面不会夹带 0x1169 数据包，而是直接以 0x035e 开头。这种情况下，解析数据包的时候要作特别处理：

- 如果数据包载荷以 0x035e 开头，则按照上面的偏移直接拿到 SQL 语句
- 如果载荷以 0x1169 开头，则简单记录一下数据包序列号之后，直接偏移到后面 0x035e 的地方，提取 SQL 语句。

对于上面的分析，可以再做一个实验。注意到，在 update.sql 文件中，第 13 行的 SQL 语句 SET SERVEROUTPUT ON 打开了服务器的一个选项，导致客户端每次

<sup>4</sup>前面提到的登陆后的第一条查询语句以 0x035e 开头是因为前面没有任何语句的需要处理，所以也就没有前面的 0x1169 数据包。

都要发送一条如下的 SQL 语句到服务器

```
BEGIN DBMS_OUTPUT.GET_LINES(:LINES, :NUMLINES); END;
```

包含这条语句的载荷数据包，其都是以 0x035e 开头的，不带 0x1169 数据包。而且在服务器端对该条数据包的返回载荷中，是以 0x0b05 开头的。如果将第 13 行去掉，则不会有这些信息，从而所有的数据包又再次携带 0x1169 数据包。

做这个实验的目的是为了说明某些 SQL 语句的执行会导致数据包载荷的头部发生一点变动，在解析的时候，需要注意一下。

#### 2.1.4 Oracle 12C TNS 协议的异同

Oracle 12C 中的 TNS 协议与 11g 中又有不同，现将其异同列举一二：

- 客户端登陆数据包中，第一条 connect 数据包格式是相同的，包括后面的 resend 数据包和 accept 数据包
- 表 2.1、2.2、2.3 的意义基本保持不变
- 后面双方的 0xdeadbeef 数据包开始不同，数据包的整体长度放在第三、四个字节，载荷数据包类型放在了第 11 个字节（这个与 11g 中相同），所以在解析载荷类型时，要偏移 10 个字节
- 0x035exx 之后偏移 73 个字节才到具体的 SQL 长度字节，而非 67 个字节
- 客户端发往服务器端的 0x02 类型的载荷数据包非常长，被分成三个数据包发送，而单个数据包的长度最大似乎为 1514 个字节。同样，服务器端返回的 0x02 数据包也很长，同样被分包。Oracle 11g 的情况是否如此，待定

更进一步的对比分析，后续再跟进。这样分析的目的是警告协议分析代码编写者，尽量分拆分析代码的逻辑，它们可能在不同的协议版本中都可以重用，只是组合方式的不同而已。

#### 2.1.5 Oracle 退出登陆数据包的组成

[TODO]

### 2.2 SQL Server 通信协议分析

SQL Server 数据库中采用的通信协议为 TDS<sup>5</sup>，该协议的具体规格基本已经公开，而且有 FreeTDS 的开源实现，即通过 FreeTDS 即可模拟客户端与 SQL

<sup>5</sup>TDS 有多个版本，各个版本之间有些微差别，目前的大版本号 7，如果 SQL Server 版本大于等于 2005，则都支持 TDS 7 了，所以优先支持 TDS 7。

Server 通信。

在 FreeTDS 中带有一个 `tsql` 工具（FreeTDS/src/apps/tsql），用它即可模拟客户端行为：

```
# tsql -S <sqlserver-ip> -U <username> -P <password>
```

此处服务器 IP 为 192.168.37.129，用户名和密码按照服务器的安装配置来选择。

TDS 的协议流程基本上比较简单，就目前从 Wireshark 中抓取的数据包而言，其严格遵守基本的“一来一回”规则，在单次通信中（比如登录，输入查询语句等），基本就两个数据包就完成任务。不像 DRDA 那样，在单个操作中来回折腾多个数据包。

TDS 的数据包基本上格式如下

```
tcp_pkt
|
+-- type          : 2B
+-- status        : 1B
+-- length        : 2B
+-- pkt_number    : 1B
+-- window        : 1B
+-- data          : nB
```

基本上前面 7 个字节为 TDS 数据包头，里面对一些基本信息进行了描述，便于后面对 data 的解析。

对登录的数据包而言，其 data 部分的格式如下：

```
data
|
+-- login_pkt_hdr : 36B
+-- len_and_offset : 50B
+-- client_name   : nB
+-- username      : nB
+-- password      : nB
+-- app_name      : nB
+-- server_name   : nB
+-- lib_name      : nB
+-- locale        : nB
```

data 部分数据的特点是

- data.login\_pkt\_hdr 部分列出了很多关于数据包的属性信息，由于涉及字段较多，在此不详列
- data.len\_and\_offset 则列出了自 data.client\_name 到 data.locale 各个字段在数据包中的偏移量以及长度
- 自 data.client\_name 到 data.local，所有的数据均为 Unicode 编码，对于 ASCII 字符，其所占字节为 2，低字节补上 0x00，例如，字符串 test 的数据包格式为 0x74 0x00 0x65 0x00 0x73 0x00 0x74 0x00

对查询数据包而言，其 data 部分的格式很简单，直接包含具体的查询语句，同理，其字符编码也为 Unicode 编码。

### 2.3 IBM DB2 数据包协议分析

DB2 数据库中采用的通信协议为 DRDA (*distributed rational database architecture*), 其数据包结构为

```
tcp_pkt
|
+-- DRDA
|  |
|  + DDM
|  |
|  + parameter(optional)
|  + ...
|
+-- DRDA
|
+-- ...
```

在单个 TCP 数据包中可能包含多个 DRDA 数据包, 在单个 DRDA 数据包中包含有一个 DDM (*distributed database management*) 数据包和可选的一个或多个 parameter 数据包。各个 DRDA 包以及各个 parameter 包之间均无任何分隔字符, DRDA、DDM 以及 parameter 之间亦无分隔字符。

DDM 数据包长度固定为 10 字节, 其格式如下:

```
DDM
|
+-- length      : 2B
+-- magic       : 1B
+-- format      : 1B
+-- correl_id   : 2B
+-- length2     : 2B
+-- code_point  : 2B
```

其中

- DDM.length 指当前 DRDA 数据包总长度, 包含一个 DDM 包长度以及若干 parameter 包长度
- DDM.magic 基本上固定为 0xD0
- DDM.format 包含一些属性信息, 权作参考
- DDM.correl\_id 目前不知道有什么用
- DDM.length2 目前不知道有什么用
- DDM.code\_point 指当前 DRDA 的数据包类型, 根据该数据包类型可以知道这个 DRDA 是干什么用的

其中 DDM.format 的位划分为:

```
format
|
+-- reserved      : 1b
+-- chained       : 1b
+-- continue      : 1b
+-- same_correlation : 1b
+-- dss_type      : 4b
```

DRDA.parameter 的包格式为：

```
parameter
|
+-- length      : 2B
+-- code_point  : 2B
+-- data        : nB
```

其中 parameter.data 在当前版本的协议中，如果用 DB2 自带的 db2 程序与服务器通信，于原始数据的头尾分别会附加一个字节，可能是用于服务器端的识别，可以在解析时将其剪除。

这里面最重要的字段为两个 code\_point 字段，分别为 DDM.code\_point 以及 parameter.code\_point，它们分别代表具体的数据包类型，根据该类型，即可按照需求解析所需数据。

另外需要注意的一点是，在目前的分析情景中使用的是 DB2 自带的 db2 工具作为客户端与 Windows 上安装的 DB2 服务器通信，在 Wireshark 中看到的是，对于标准 ASCII 字符，有些（并非全部，根据 parameter.code\_point 而定）parameter.data 的编码格式为 EBCDIC，而非通常的 ASCII 格式，比如，对于 ASCII 字符 'd'，EBCDIC 将其编码为 \204，其十六进制为 0x84。如果需要解析这些数据，则需要将这些字节做相应的转换<sup>6</sup>。

## 2.4 MySQL 数据包协议分析

[TODO]

## 2.5 超长 SQL 语句的处理

为保持协议分析的正确性，分别对 SQL Server、MySQL 以及 Oracle 中超长 SQL 语句进行了测试，结果如下：

- SQL Server 会对数据包进行分包，单个包的大小（包括协议头部字节）为 4Kb
- MySQL 对长度为 6Kb 的 SQL 语句没做任何处理，直接发送过去了，在服务器收到的是单个数据包
- 用 SQLPlus 测试时，Oracle 11g R2 直接忽略过长的 SQL 语句并报错：SP2-0027: Input is too long (> 2499 characters) - line ignored，据实测，SQL 语句的长度不能超过 2499（不含 2499，最长为 2498），此长度包括 SQL 语句后的 ';' 字符，但是如果换做其它客户端，可能该长度有所改变，即该长度是 sqlplus 工具的限制，因为当长度为 2499 时，Wireshark 根本没有抓到该数据包，即 sqlplus 没有将该长度的数据包发送出去，客户端本身就截留了该 SQL 语句。

---

<sup>6</sup>关于转换映射，参见 [这里](#)





## Chapter 3

# Suricata 开发手册

**Suricata** 是一个开源的、高性能入侵检测/防护系统，由 **OISF** 主导开发，其主要具有以下特性：

- 相对于 **Snort**，它是一个多线程模型，使得其可尽可能充分地利用硬件资源来进行检测。
- 支持多层次的通用网络协议，如 **IP**、**TCP**、**UDP**，应用层的 **HTTP**、**SSH**、**FTP** 等。
- 能识别网络上传输的几千种文件类型，可以进行 **MD5** 检测，扩展名提取等等。

由于其支持多层次的网络协议，故可以对其进行扩展，基本上所有运行于 **IP** 层上的通信协议都能通过扩展 **Suricata** 来实现对其的检测和防护。**MySQL** 常用的通信协议为 **TCP**，故可以在 **Suricata** 中加入对 **MySQL** 协议的检测。

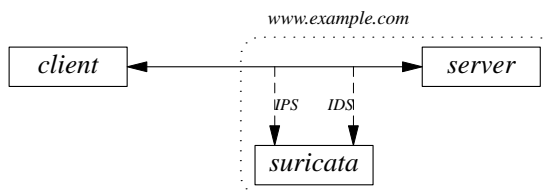
### 3.1 Suricata 的整体架构

**Suricata** 的工作原理为旁路抓包，即它并不妨碍客户端和服务端的主体通信流程，在 **IDS** 模式下，它只对过往的数据进行检测，如果发现异常数据则会对该数据做详细的记录，在 **IPS** 模式下，**Suricata** 则会干预主体通信流程，一旦发现异常情况，则采取相应的措施，可能会阻止该数据的进一步传送。

**Suricata** 的部署拓扑结构为：

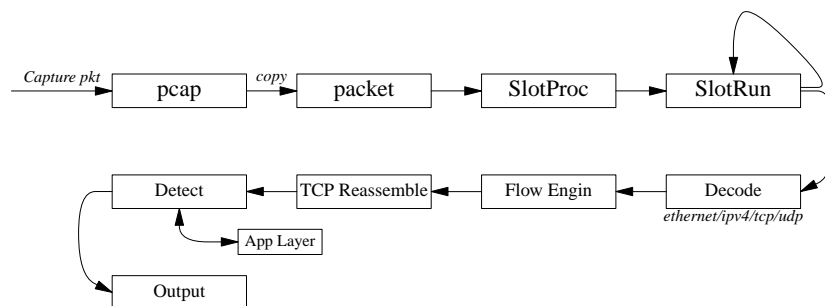
**Suricata** 提供多种工作模式，主要有 **auto**、**autofp** 以及 **worker** 三种，其中

- 三种模式中数据包都是线性处理的，数据包的状态处于不断变化之中，在不同的模块中，对数据包进行不同的切分和处理，最终完成检测和保护功能。



- `auto` 和 `autofp` 是类似的，各个模块以线程进行划分，每个模块分别有一个线程对应处理，线程之间以同步的信号量进行通信。数据包都是从一个线程模块转移到另一个线程模块，在数据包转移的过程中，会对数据包进行不同的处理。所不同的是 `autofp` 提供了负载均衡功能（*auto flow pinned load balancing*）。
- `worker` 模式和前两者不同，它在同一个线程中完成对所有模块的处理，也即流水线上的所有模块都运行于同一个线程中，而在整个 **Suricata** 内部，有多个这样的流水线，分别处理各自的数据，线程彼此之间没有数据包传递。

Suricata 中单个流水线的处理流程为：



下面以 **Mysql** 协议为例，详细说明如何在 **Suricata** 框架中添加自定义协议识别、关键字识别、日志模块等内容。

### 3.2 在 Suricata 中加入 MySQL 协议检测

在 **Suricata** 中添加 **MySQL** 协议的检测，只需要在应用层（*App Layer*）增加相应的模块即可。具体分为以下几个步骤：

1. 增加协议号。
2. 加入新的模块，将其命名为 `app-layer-mysql.c`。
3. 增加编译设置，将新加入的模块编译进 **Suricata** 中。

在头文件 `src/app-layer-protos.h` 的枚举 `AppProto` 中加入一个新的成员 `ALPROTO_MYSQL` 即完成了协议号添加，下面具体说明一下新模块的编写。

在 **Suricata** 中，所有的协议注册都是在 `src/app-layer-parser.c` 模块中完成的，所有的第三方协议模块必须提供一个注册 API 给该模块，以完成其协议注册。在这里，**MySQL** 的协议支持模块为 `src/app-layer-mysql.[ch]`，注册函数为 `RegisterMysqlParsers()`。

```
/* src/app-layer-mysql.c */
void RegisterMysqlParsers(void) {
    ...
}

/* src/app-layer-mysql.h */
void RegisterMysqlParsers(void);
```

在 `RegisterMysqlParsers()` 函数中，至少需要做以下几个步骤：

- 调用 `AppLayerProtoDetectConfProtoDetectionEnabled` 检测协议支持是否启用。
- 调用 `AppLayerProtoDetectRegisterProtocol` 添加对应用协议的支持。
- 调用 `AppLayerProtoDetectPPParseConfPorts` 函数将对应的协议号以及协议检测回调函数注册到 `src/app-layer-parser.c` 模块中。
- 调用 `AppLayerParserRegisterParser` 注册（2 个）协议解析函数。
- 调用 `AppLayerParserRegisterStateFuncs` 注册两个内存管理函数（`alloc/free`），用以管理协议状态对象的内存。

这里需要注意以下协议检测函数的注册，在 **Suricata** 中，有两类协议注册函数，一类为“带关键字的协议识别注册函数”以及“不带关键字的协议识别注册函数”。对前者而言，比如在 **HTTP** 协议中，常见的关键字有 `GET`、`POST` 等，如果要识别这样的数据包，可调用函数 `AppLayerProtoDetectPMRegisterPatternCS` 或函数 `AppLayerProtoDetectPMRegisterPatternCI`；<sup>1</sup> 当某些通信协议没有关键字时，比如 **DNS** 协议或 **MySQL** 协议，则使用函数 `AppLayerProtoDetectPPParseConfPorts`。所不同的是，由于后者并未提供关键字识别，所以需要另行传入一个函数来检测应用层数据流，并且，当该检测函数返回时，将对应的协议号（也即 `ALPROTO_MYSQL`）作为其返回值，否则返回 `ALPROTO_FAILED`，表示数据流和所支持的协议不匹配<sup>2</sup>。

除此之外，**Suricata** 会使用协议模块中一个重要的数据结构，即协议状态对象。在该对象中，你可以设置所需的任何数据，因为它对 **Suricata** 而言是一个 `void *` 对象。在一个通信的生命周期内，该状态会一直存在，所以你至少可以在其中保存通信的状态信息，以便于通信协议的解析。由于该对象由 **Suricata** 主框架初始化，所以你需要将该对象的内存管理函数告知 **Suricata**。

<sup>1</sup>这两者的区别在于是否对关键字的大小写进行区分。

<sup>2</sup>可以通过在 `suricata.yaml` 中增加协议以及端口来匹配发送给服务端的数据，这样更为可靠。

完成代码后，需要将其加入到 **Suricata** 的编译规则中来，在 `src/Makefile.am` 中，于 `suricata_SOURCES` 列表中加入新加的代码文件名即可。

对于新加入的代码，**Suricata** 主框架代码会调用它们，对此需要添加相应的头文件包含。对新加入的应用层模块，需要在 `src/app-layer-parser.c` 的 `AppLayerParserRegisterProtocolParsers` 函数中增加其注册函数的调用。

### 3.3 修改 Suricata 配置文件

如果要想新的模块可以正常运行，还需要将该模块的配置添加到配置文件中。在 **Suricata** 的配置文件 `suricata.yaml` 中，在 `app-layer.protocols` 中加入 `mysql`：

```
app-layer:
  protocols:
    mysql:
      enabled: yes
      detection-ports:
        tcp:
          toserver: 3306
```

注意，这里指定了具体的协议以及端口，客户端的数据只有通过 **TCP** 发送到 **3306** 端口才能被 **MySQL** 所注册的模块捕获，这样做的好处就是不会遗漏发送给服务器端的数据，即使该数据包的格式可能不对。

### 3.4 在 Suricata 中增加关键字识别

关键字的识别主要用于 **detect** 模块对数据报文的分析。这些关键字通常都配置在规则文件中，在规则文件中，可以针对关键字上发生的动作做不同的行为选择，比如，如果定义了与 **MySQL** 相关的规则识别，当有用户以 `root` 用户名登陆数据库时，则处罚 **detect** 模块中的一个警告信息，其规则大概为

```
alert mysql any any -> any any (msg:"mysql user(root) detected";
  flow:to_server,established; mysql-user:root; sid:2240000; rev:1;)
```

其中，`alert` 即 **detect** 模块的一种行为，其它几种行为有 `pass`、`reject`、`drop`。后面的 `mysql` 即规则的种类，在 **Suricata** 中，默认内置了多种规则种类，如 **TLS**、**HTTP**、**DNS** 等等。后面的四个 `any` 即表示一个 **socket** 连接元素，分别代表源端和目的端的 **IP** 和端口，箭头 `->` 表示数据流向。后面 `()` 中的类容就是具体规则的文法表示；此处的 `msg` 就是警告信息最后写入日志文件的字符串，`flow` 表示数据流，此处它有两个属性，一个表述数据流向，一个表述连接已经建立。后面的 `mysql-user` 也是一个即将添加的关键字，它用来承接规则中指定的用户名。后面的 `sid` 表示当前规则的全局 **ID**，在 **Suricata** 中，通常会划分一个 **ID** 区间给特定种类的规则（类似 **IP** 段划分）；最后的 `rev` 应该表示版本，因为规则文件可能时常变迁（根据不同的安全环境），用它来记录一些版本的情况，便于维护。

要使得上面的这条规则生效，可以先在 `rules/` 中增加 `mysql-events.rules` 文件。下面以关键字 `mysql-user` 为例，分析一下如何在 **Suricata** 中增加新的关键字。

在 `mysql-events.rules` 中，增加上面列出那个示例规则，它表示当用户 `root` 登陆时，记录一条 `alert` 信息到日志文件中，此处的日志信息记录在文件 `log/suricata/fast.log` 中。下面是一个 **log** 例子：

```
01/09/2014-15:34:21.196162  [**] [1:2240001:1] mysql user(root) detected
[**] [Classification: (null)] [Priority: 3] {TCP} 192.168.36.131:54107
-> 192.168.37.119:3306
```

接下来就是在代码中添加具体的模块，来实关键字的识别。

### 3.5 增加关键字识别的实现

以增加关键字 `mysql` 以及 `mysql-user` 为例，需要做以下几个工作：

1. 增加协议关键字识别，此处指前面的 `mysql` 关键字，它是 **MySQL** 协议的标识，它之下可以涵盖其它与 **MySQL** 相关的关键字。
2. 增加关键字识别模块，此处即各种与 **MySQL** 相关的关键字。
3. 在 `src/detect.h` 的枚举中增加一些宏定义，这些宏定义就是各种 **MySQL** 关键字的索引。

从上一节中可以看到，在动作 `alert` 后面加入了一个 `mysql` 关键字，这就使得该条规则只会匹配 **MySQL** 的通信数据，而不会干扰其它应用层（如 **HTTP**）或协议层数据（如 **IP** 或 **TCP**）。由于前面已经加入了 **MySQL** 应用层，所以这里使用 `mysql` 作为匹配 **MySQL** 关键字的协议识别，注意，如果要修改协议识别（如将其改成 `mysql-4.1`），那么在 `RegisterMysqlParsers` 函数中也需要更新，将参数 `proto_name` 改成相应值。之所以它们之间有强烈的关联，是因为 `app-layer` 模块的初始化先于签名规则的初始化，所以，后者能在 `app-layer` 模块中找到与关键字匹配的协议。

此处可以视作模块间的耦合，如果在 `detect` 模块也加入一种识别，而不借助 `app-layer` 模块的关键字，更易于理解和实现。

下一步就是增加关键字识别模块 `src/detect-mysql-keywords.[ch]`，该子模块被 **Detect** 模块调用，以检测规则签名中的规则是否匹配。

若在该子模块中增加关键字 `mysql-user` 的识别和检测，需要做以下几个步骤：

1. 编写注册函数 `DetectMysqlKeywordsRegister`，在其中追加几个回调函数：

- (a) 关键字 `mysql-user` 的设置函数：`DetectMysqlUserSetup()`
- (b) 匹配关键字 `mysql-user` 的函数：`DetectMysqlUserALMatch()`，其中 `AL` 指应用层。
- (c) 如果关键字值的匹配过程中需要分配新的对象，则另需注册一个该对象的资源管理函数，用于释放内存。在 `mysql-user` 关键字中，该函数为 `DetectMysqlUserFree()`。

2. 在 `detect` 模块的 `SigTableSetup` 函数中增加对该注册函数的调用。

函数 `DetectMysqlKeywordsRegister` 是这样子的：

```
void DetectMysqlKeywordsRegister(void) {
    sigmatch_table[DETECT_AL_MYSQL_USER].name = "mysql-user";
    sigmatch_table[DETECT_AL_MYSQL_USER].desc = "mysql TBD";
    sigmatch_table[DETECT_AL_MYSQL_USER].url = "mysql TBD";
    sigmatch_table[DETECT_AL_MYSQL_USER].Match = NULL;
    sigmatch_table[DETECT_AL_MYSQL_USER].AppLayerMatch = DetectMysqlUserALMatch;
    sigmatch_table[DETECT_AL_MYSQL_USER].alproto = ALPROTO_MYSQL;
    sigmatch_table[DETECT_AL_MYSQL_USER].Setup = DetectMysqlUserSetup;
    sigmatch_table[DETECT_AL_MYSQL_USER].Free = DetectMysqlUserFree;
    sigmatch_table[DETECT_AL_MYSQL_USER].RegisterTests = NULL;
    sigmatch_table[DETECT_AL_MYSQL_USER].flags |= SIGMATCH_PAYLOAD;

    /* ... you can add more keywords here */
}
```

这里要注意的是，匹配的回调函数有两个，一个是 `Match`，一个是 `AppLayerMatch`，前者主要用于协议层，如校验和检查，后者用于应用层。这两个回调函数的参数很不同，直接影响匹配的结果。

从上面的代码中可以看到，主要的字段是 `name`，将其设置成 `mysql-user`，它就是关键字的名称；其中 `desc` 和 `url` 主要做日志使用，便于调试。`alproto` 即之前应用层中注册的宏 `ALPROTO_MYSQL`，最后在 `flags` 上追加属性 `SIGMATCH_PAYLOAD`，便于 `Detect` 模块的识别。

这里的几个回调函数都是由 `Detect` 模块来调用。`DetectMysqlUserSetup` 用来设置关键字，它会新建一个 `DetectMysqlUser` 对象，该对象由具体的关键字逻辑定义，此处只是将关键字 `mysql-user` 作为其唯一的字段，在分析规则文件中的规则时，会将关键字传入该函数。对象新建完以后，它以 `void *` 的形式被加入到 `SigMatch` 对象中，作为关键字 `mysql-user` 检测的辅助对象 (`ctx`)。

函数 `DetectMysqlUserFree` 用来释放 `DetectMysqlUser` 对象。

函数 `DetectMysqlUserALMatch` 就是具体的匹配函数，它带入的参数非常多，常常只需要用几个行了。此处简单地将 `DetectMysqlUser` 中的字段与 `alstate` 中的用户名做一下对比即可直到是否匹配该规则。同时此处会代入 `Signature` 对象，其中有该条规则的动作字段，可以将其设置在 `MysqlTransaction` 中，用于将具体的动作记录到日志文件中。

接下来在 `src/detect.h` 中增加宏定义 `DETECT_AL_MYSQL_USER`<sup>3</sup>。在 `src/detect.c` 中增加 `MySQL` 关键字模块的头文件包含：

<sup>3</sup>注意，如果增加其它关键字，则需要增加新的宏定义。

```
#include "detect-mysql-keywords.h"
```

然后在 src/Makefile.am 的 suricata\_SOURCE 中增加新模块编译规则即可。

### 3.6 增加 Json 输出模块

在现有的 Suricata 中，log 模块依旧使用其自定义的格式记录，但是 event 时间则采用 JSON 格式记录。本节只涉及 event 相关的 JSON 记录。在 Suricata 中，以文件 output-json-xxx.[ch] 命名的模块有 DNS、HTTP、SSH、TLS 等。此处仿照其实现，以 MySQL 协议为例，说明一下如何在 Suricata 中添加 JSON 的日志输出功能。

### 3.7 Suricata 开发过程中碰到的疑难问题

在 Suricata 的日常开发中，碰到一些比较难的问题，这些问题需要深挖 Suricata 的底层实现。现将其一一列举出来，以备日后查看。

#### 3.7.1 Suricata 只能将小部分的 TNS 协议数据包送到应用协议层

具体的表现症状就是当抓取登录数据包时，只有前面几个数据包被抓取到了，后面的数据包全部没有上浮到应用层，导致应用层无法解析数据包。但是登陆的过程以及后续的 SQL 执行都能正常进行，这表明数据确实从 Suricata 中穿过去了，但是没有上报到应用层。

一步步跟踪发现，数据没有到达应用层是因为从到应用层中间的处理应为某种原因中断了，从而数据没有上报。具体的地点是：

在函数 StreamTcpReassembleInlineAppLayer 的 for (; seg != NULL;) 中，有如下代码

```
1 ...
2 for (; seg != NULL;) {
3     if (p->flow->flags & FLOW_NO_APPLAYER_INSPECTION) {
4         if (seg->flags & SEGMENTTCP_FLAG_RAW_PROCESSED) {
5             SCLogDebug("removing seg %p seq %"PRIu32
6                 " len %"PRIu16"", seg, seg->seq, seg->payload_len);
7
8             TcpSegment *next_seg = seg->next;
9             StreamTcpRemoveSegmentFromStream(stream, seg);
10            StreamTcpSegmentReturntoPool(seg);
11            seg = next_seg;
12            continue;
13        } else {
14            break;
15        }
16    }
17 ...
```



其中，第三行的条件成立，但是第四行不成立，导致退出当前的 for 循环，但是这个 for 循环是非常重要的，其中涉及将 Packet 对象中的数据拷贝出来，并调用 AppLayerHandleTCPData 进入应用层处理。for 循环退出之后，函数 StreamTcpReassembleInlineAppLayer 基本退出，导致这个数据包不能进入应用层处理。

发现现场之后，可以进一步挖掘，为何会进入第三行这个 if 语句，宏 FLOW\_NO\_APPPLAYER\_INSPECTION 是在什么时候设置上去的？通过 grep 一下源码树，发现在文件 src/flow.h 中定义了一个函数 FlowSetSessionNoAppplayerInspectionFlag，就是通过它来设置这个标志位的。接下来再 grep 这个函数，发现了如下一些地方

```
src/app-layer.c:244:      FlowSetSessionNoAppplayerInspectionFlag(f);
src/app-layer.c:273:      FlowSetSessionNoAppplayerInspectionFlag(f);
src/app-layer.c:327:      FlowSetSessionNoAppplayerInspectionFlag(f);
src/app-layer.c:351:      FlowSetSessionNoAppplayerInspectionFlag(f);
src/app-layer-parser.c:835: FlowSetSessionNoAppplayerInspectionFlag(f);
src/app-layer-parser.c:862: FlowSetSessionNoAppplayerInspectionFlag(f);
src/detect.c:11060:      FlowSetSessionNoAppplayerInspectionFlag(p2->flow);
src/detect.c:11243:      FlowSetSessionNoAppplayerInspectionFlag(p2->flow);
src/flow.h:426:static inline void FlowSetSessionNoAppplayerInspectionFlag(Flow *);
src/flow.h:494:static inline void FlowSetSessionNoAppplayerInspectionFlag(Flow *f) {
src/stream-tcp.c:4201:      FlowSetSessionNoAppplayerInspectionFlag(p->flow);
```

在该函数处设置断点，第一次触发该断点的调用栈为

```
#0 FlowSetSessionNoAppplayerInspectionFlag (f=0x8ef76a8) at flow.h:495
#1 0x080a561c in AppLayerParserParse (alp_tctx=0xb6512c20, f=0x8ef76a8, alproto=14, flags=4 '\004', inp=0x080a561c) at app-layer-parser.c:862
#2 0x0805e771 in AppLayerHandleTCPData (tv=0x9125f20, ra_ctx=0xb6512af8, p=0x8d3e5c0, f=0x8ef76a8, ssn=0xb65126e0) at app-layer.c:351
#3 0x082ce657 in StreamTcpReassembleInlineAppLayer (tv=0x9125f20, ra_ctx=0xb6512af8, ssn=0xb6595160, stream=0x082d7afd) at stream-tcp-reassemble.c:11060
#4 0x082d7afd in StreamTcpReassembleHandleSegment (tv=0x9125f20, ra_ctx=0xb6512af8, ssn=0xb6595160, stream=0x08296ab9) at stream-tcp-reassemble.c:11243
#5 0x08296ab9 in HandleEstablishedPacketToServer (tv=0x9125f20, ssn=0xb6595160, p=0x8d3e5c0, stt=0xb65126e0) at stream-tcp.c:4201
#6 0x0829aed3 in StreamTcpPacketStateEstablished (tv=0x9125f20, p=0x8d3e5c0, stt=0xb65126e0, ssn=0xb6595160) at stream-tcp.c:4201
#7 0x082b4426 in StreamTcpPacket (tv=0x9125f20, p=0x8d3e5c0, stt=0xb65126e0, pq=0x9126188) at stream-tcp.c:4201
#8 0x082b5588 in StreamTcp (tv=0x9125f20, p=0x8d3e5c0, data=0xb65126e0, pq=0x9126188, postpq=0x0) at stream-tcp.c:4201
#9 0x082ef0a4 in TmThreadsSlotVarRun (tv=0x9125f20, p=0x8d3e5c0, slot=0x9126088) at tm-threads.c:559
#10 0x08276651 in TmThreadsSlotProcessPkt (tv=0x9125f20, s=0x9126088, p=0x8d3e5c0) at tm-threads.h:142
#11 0x08277f0a in NFQCallBack (qh=0xb6500620, nfmsg=0xb6500648, nfa=0xb6e908ac, data=0x8430000 <nfq_t>) at nfq.c:142
...
```

从 #6 开始，从此处建立连接，开始接收数据包，到 #5 中处理数据包，在 #4 中重组接收到的数据包，进入 #3，在函数 StreamTcpReassembleInlineAppLayer 中进入 #2，注意，此时进入了 app-layer 逻辑层。然后进一步进入函数 AppLayerParserParse，最终在该函数中调用 FlowSetSessionNoAppplayerInspectionFlag。之所以调用该函数，是因为某个 goto 语句发现了某个条件不符合，这几个条件是

- 发现了 stream gap
- AppLayerParserState 对象分配失败
- app-layer 的 state 对象分配失败
- app-layer 的 parser 函数执行失败



## Chapter 4

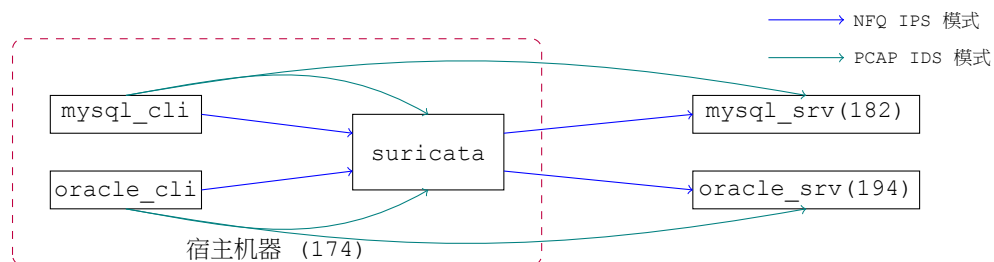
# Suricata 预研 Demo

### 4.1 主要内容

- 环境搭建
- 协议分析（在前面的章节中，此处不表）
- 效果展示（IDS/IPS）

### 4.2 网络结构

Suricata 和数据库客户端均位于同一台机器。数据库服务器位于 182 以及 194。其网络结构如图所示：



### 4.3 Suricata IDS/IPS 对现有数据库协议支持的展示

目前对 MySQL 的协议以及基本支持，能获取对应的 SQL 语句以及更详细的信息。对 Oracle 11g 数据库而言，目前可以拿到其基本的 SQL 语句信息，其它更深入的信息（session 管理、返回的数据分析）尚不能获取。

### 4.3.1 IDS 功能演示

以命令 `./suricata r -c yaml/demo.yaml -i eth1` 启动 IDS 模式的 Suricata，此处使用 `eth1` 作为嗅探网卡。

对 Oracle 以及 MySQL 数据库而言，分别对应如下两条检测规则：

```
alert oracle1lg any any -> any any (msg:"oracle user(coanor) detected";
  flow:to_server,established; oracle1lg-user:coanor; sid:2250000; rev:1;)
alert oracle1lg any any -> any any (msg:"oracle database(orcl1lg) detected";
  flow:to_server,established; oracle1lg-sid:orcl1lg; sid:2250001; rev:1;)

alert mysql any any -> any any (msg:"mysql user(root) detected";
  flow:to_server,established; mysql-user:root; sid:2240000; rev:1;)
alert mysql any any -> any any (msg:"mysql database(aap_log) detected";
  flow:to_server,established; mysql-database:aap_log; sid:2240001; rev:1;)
```

对这两条规则而言，如果满足其检测条件，就会在 `fast.log` 文件中记录相应的 **alert** 信息。同时，在 MySQL 以及 Oracle 的 JSON 文件日志中，也会记录这一行为 (JSON 文件中的 `action` 字段)。对其它几种行为而言，分别更改前面的 `alert` 即可。

**alert** 日志在常规的 JSON 日志中为：

```
{
  "timestamp":"2014-06-03T10:09:54.203021",
  "event_type":"oracle1lg",
  "src_ip":"192.168.37.174",
  "src_port":45827,
  "dest_ip":"192.168.37.194",
  "dest_port":1521,
  "proto":"TCP",
  "oracle1lg":{"user":"coanor",
    "db_name":"orcl1lg",
    "action":"ALERT",
    "meta_info":{"sql":null, "cmd":"unkonw"}
  }
}
```

常规的 SQL 日志输出示例为：

```
{
  "timestamp":"2014-06-03T10:12:30.534439",
  "event_type":"oracle1lg",
  "src_ip":"192.168.37.174",
  "src_port":45827,
  "dest_ip":"192.168.37.194",
  "dest_port":1521,
  "proto":"TCP",
  "oracle1lg":{"user":"coanor",
    "db_name":"orcl1lg",
    "action":"UNKNOWN",
    "meta_info":{"sql":"insert into employee (EMPLOYEE_ID, MANAGER_ID,FIRST_NAME,
      LAST_NAME, TITLE, SALARY)values( 1 , 0 , 'James' , 'Smith' , 'CEO', 800000)",
      "cmd":"query"
    }
  }
}
```

在 `fast.log` 文件中的示例为

```
06/03/2014-13:38:41.827922  [**] [1:2250000:1] oracle user(coanor) detected [**]  
[Classification: (null)] [Priority: 3] {TCP} 192.168.37.174:46697 -> 192.168.37.194:1521
```

#### 4.3.2 IPS 功能演示

设置如下 **iptables** 规则后，再启动 **Suricata** `./suricata -c yaml/demo.yaml -q 0`。

```
sudo iptables -A INPUT -j NFQUEUE  
sudo iptables -A OUTPUT -j NFQUEUE  
  
# 允许 SSH 访问，便于调试  
sudo iptables -I INPUT -p tcp -m tcp --dport 22 -j ACCEPT  
sudo iptables -I OUTPUT -p tcp -m tcp --sport 22 -j ACCEPT
```

连接建立后，显示的结果数据和 **IDS** 模式下的一致。