Plugin MySQL dissector per nProbe

Emanuele Tomasi

6 ottobre 2010

Indice

1 Uno sguardo a *nProbe*

Sviluppato da Luca Deri, *nProbe* è una sonda che implementa il protocollo *NetFlow* (il nome infatti sta per *NetFlow probe*). A partire dalla versione 9 di *NetFlow*, le informazioni esportate non sono più standardizzate secondo quanto stabilito da Cisco (creatore del protocollo), ma sono scelte dall'utente attraverso un *template*. Grazie all'uso dei *template*, quindi, le informazioni generate da *nProbe* non sono solo le classiche 7-tuple:

1. indirizzo IP sorgente
2. indirizzo IP destinatario
3. porta sorgente
4. porta destinazione
5. protocollo IP
6. interfaccia di ingresso (SNMP ifIndex)
7. IP Type of Service (*ToS*)

ma ve ne possono essere anche altre aggiuntive e/o sostitutive.

1.1 La struttura a plugin di *nProbe*

nProbe implementa questi nuovi protocolli con un insieme di plugin esterni al codice principale e tutti inseriti nella directory `plugins`. Ogni plugin è in grado di esportare una o più informazioni identificate da un *id* e un *nome* univoci. Al momento dell'invocazione, l'utente sceglie cosa vuole esportare usando l'opzione:

```
[--flow-templ|-T] <flow template>   | Specify the NFv9 template
```

nProbe, quindi, va alla ricerca dei plugin che implementano almeno una tra le informazioni richieste nel *template* e li attiva.

Ad esempio, un plugin attualmente presente è l'*SMTP Protocol Dissector* che nell'help di *nProbe* riporta quanto segue:

```
Plugin SMTP Protocol Dissector templates:
[NFv9 57657][IPFIX 35632.185] %SMTP_MAIL_FROM           Mail sender
[NFv9 57658][IPFIX 35632.186] %SMTP_RCPT_TO             Mail recipient
```

In questo esempio, il plugin può esportare sia il mittente che il destinatario di una email. L'*id* univoco che attiva il plugin richiedendogli di esportare il mittente è 57657 (per *NetFlow v.9*. Invece è 35632[1].185 per il protocollo *IPFIX*[2]), mentre il *nome* univoco da inserire nel *template* è

[1] 35632 è il *Private Enterprise Number (PEN)* assegnato a *ntop* dall'*Internet Assigned Numbers Authority (IANA)*. Anche *ntop* è stato sviluppato da Luca Deri.

[2] L'*Internet Protocol Flow Information Export (IPFIX)* è un protocollo standardizzato dall'*Internet Engineering TaskForce (IETF)* basato su *NetFlow v.9*.

`%SMTP_MAIL_FROM`. Se si vuole attivare il plugin richiedendogli di esportare il destinatario della email, basta inserire nel *template* la stringa `%SMTP_RCPT_TO`. Il comando seguente, quindi, fa attivare il plugin *SMTP Protocol Dissector* e gli chiede di esportare nel flusso sia il mittente che il destinatario di una email[3]:

```
nprobe -T "%SMTP_MAIL_FROM %SMTP_RCPT_TO"
```

1.2 Cosa scrivere nei plugin

Dovendo stare "staccato" dal codice principale, un plugin deve implementare delle funzioni dai prototipi ben definiti ed essere in grado di fornire gli indirizzi di queste funzioni a *nProbe*, in modo tale che quest'ultimo sia poi in grado di eseguirle ogniqualvolta lo ritenga necessario.

1.2.1 L'interfaccia dei plugin

Tutte le informazioni su di un plugin sono inserite nella struttura dati `PluginInfo`, dichiarata come segue nel file `engine.h`:

```
typedef struct pluginInfo {
  char *nprobe_revision, *name, *version, *descr, *author;
  u_char always_enabled, enabled;
  PluginInitFctn initFctn;
  PluginTermFctn termFctn;
  PluginConf pluginFlowConf;
  PluginFctn deleteFlowFctn;
  u_char call_packetFlowFctn_for_each_packet;
  PluginPacketFctn packetFlowFctn;
  PluginGetPluginTemplateFctn getPluginTemplateFctn;
  PluginCheckPluginExportFctn checkPluginExportFctn;
  PluginCheckPluginPrintFctn checkPluginPrintFctn;
  PluginSetupFctn setupFctn;
  PluginHelpFctn helpFctn;
} PluginInfo;
```

Le variabili il cui tipo inizia con "Plugin" sono dei puntatori a funzione il cui prototipo è dichiarato sempre nello stesso file. Le prime cinque variabili di tipo stringa sono informazioni sul plugin:

nprobe_revision la revisione di *nProbe* per cui è scritto il plugin. Se questo viene aggiornato al pari del programma principale si può anche usare la costante `NPROBE_REVISION`.

name il nome del plugin.

version la versione.

descr la sua descrizione.

author le informazioni sullo sviluppatore.

[3]In realtà questo *template* è veramente minimale ed è a solo scopo di esempio. Se non si inseriscono nel *templata* le informazioni relativa agli indirizzi, alle porte e al protocollo IP, *nProbe* ignora totalmente queste informazioni e non riesce a riconoscere i flussi in maniera corretta.

Dopo queste informazioni seguono due variabili booleane:

always_enabled indica se il plugin deve essere attivo indipendentemente dal *template* utente (settata a 1) oppure deve essere attivato esclusivamente se richiesto nel *template* (settata a 0).

enabled è una variabile per uso interno. Indica se il plugin è stato attivato o meno.

Infine gli altri elementi della struttura che, come già detto, sono principalmente dei puntatori a funzione. Se si decide di non implementare una o più di queste funzioni, basta settare il loro valore a `NULL`:

initFctn viene invocata per inizializzare il plugin. *nProbe* passa a questa funzione l'intera command line, in modo tale che un plugin possa andare alla ricerca di qualche parametro che gli interessa (si veda la funzione `helpFctn()`).

termFctn opposta alla precedente `initFctn()`, viene invocata per terminare il plugin.

pluginFlowConf tutte le informazioni che un plugin può esportare, come detto, sono caratterizzate da un *id* e da un *nome* ma oltre a questi due attributi ne esistono altri che *nProbe* usa internamente. Questa funzione, ritorna un'array terminato da un entry particolare, con tutti gli attributi associati ad ogni informazione che il plugin è in grado di esportare. Si capirà meglio quando si vedrà come fa il plugin ad informare *nProbe* su quali informazioni è in grado di esportare.

deleteFlowFctn viene invocata quando si elimina un flusso. Ad esempio, in questa funzione il plugin può liberare eventuale memoria allocata associata al flusso.

call_packetFlowFctn_for_each_packet questa variabile, di tipo booleano, indica se si vuole chiamare la funzione `packetFlowFctn()` per ogni pacchetto del flusso (settata ad 1), oppure solo per il primo pacchetto del flusso (settata a 0).

packetFlowFctn è la funzione principale che viene invocata al ricevimento di ogni pacchetto appartenente al flusso (si veda la variabile precedente `call_packetFlowFctn_for_each_packet`). È in questa funzione che il plugin estrae le informazioni.

getPluginTemplateFctn questa funzione prende il nome di un elemento del *template* utente (un elemento della stringa passata con l'opzione `-T`) e, se il plugin lo gestisce, ritorna tutti gli attributi ad esso associati, altrimenti ritorna `NULL`. Anche questa funzione sarà più chiara una volta che si vedrà come fa il plugin ad informare *nProbe* su quali informazioni è in grado di esportare.

checkPluginExportFctn viene invocata al momento dell'esportazione dei dati. In questa funzione il plugin scrive in un buffer limitato, quelle informazioni richieste nel *template* che è in grado di gestire.

checkPluginPrintFctn *nProbe* su richiesta può salvare le informazioni esportate anche in un file temporaneo e, quando lo fa, invoca questa funzione. È simile alla precedente funzione `checkPluginExportFctn()`, solo che in questo caso siccome i dati sono scritti in un file, non c'è limite

alle informazioni che si possono scrivere. Ad esempio, se il mittente di una email è troppo lungo, potrebbe essere necessario troncare questa informazione per farla entrare nel buffer da esportare, mentre non vi sono restrizioni quando l'informazione è scritta nel file temporaneo.

setupFctn viene invocata subito dopo l'inizializzazione di tutti i plugin, poco prima che *nProbe* inizi la cattura dei pacchetti.

helpFctn viene invocata durante l'esecuzione dell'help di *nProbe* (parametro `--help|-h`) per mostrare le opzioni necessarie al plugin. Se un utente, invocando *nProbe*, gli passa sulla command line una o più tra queste opzioni, il plugin le può intercettare durante l'esecuzione della funzione `initFctn()`.

1.2.2 Far avere l'interfaccia ad *nProbe*

Una volta che nel codice del plugin viene definita la struttura `PluginInfo`, si deve farla avere ad *nProbe*. Per fare questo ci sono due modi, a seconda che i plugin vengano inseriti staticamente nel binario compilato, oppure vengano caricati dinamicamente a tempo di esecuzione. In ogni caso, si deve implementare una funzione senza parametri che ritorna un puntatore alla struttura `PluginInfo` definita, questa sarà la cerniera tra il plugin e il codice principale. Nel primo caso, in cui i plugin sono inseriti nel binario a tempo di compilazione, si deve inserire nel file `plugin.c` la dichiarazione della funzione esterna (definita nel plugin) che restituirà la struttura dati. Ad esempio, per il plugin di SMTP nel file c'è inserita questa dichiarazione:

```
#ifdef MAKE_STATIC_PLUGINS
...
extern PluginInfo* smtpPluginEntryFctn(void)
...
#endif
```

Invece, nel caso in cui i plugin sono caricati in maniera dinamica a tempo di esecuzione, *nProbe* deve sapere a priori il nome della funziona da invocare una volta caricato il codice, così il plugin deve definire, al suo interno, la funzione:

```
PluginInfo *PluginEntryFctn(void)
```

La figura 1 mostra come vengono prelevate le interfacce dei plugin, quando questi vengono inseriti a tempo di esecuzione.

1.2.3 Le informazioni esportate

Una volta che il plugin è correttamente legato al codice di *nProbe* non resta che una cosa da fare, indicare a quest'ultimo quando attivarlo. Come già detto, *nProbe* si basa sul *template* utente passato attraverso il parametro `-T` della linea di comando per sapere quale plugin attivare e quale no. Per ogni informazione che il plugin è in grado di esportare, si deve definire una struttura `V9V10TemplateElementId`, dichiarata nel file `nprobe.h` come segue:

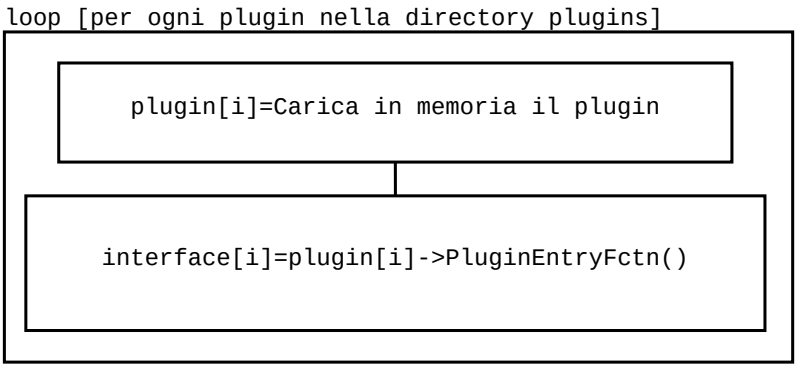

Figura 1: Prelevamento interfacce plugin a run-time

```
typedef struct flow_ver9_ipfix_template_elementids {
  u_int8_t  isOptionTemplate; /* 0=flow template, 1=option template */
  u_int8_t  useLongSnaplen;
  u_int32_t templateElementEnterpriseId;
  u_int16_t templateElementId;
  u_int8_t variableFieldLength;
  u_int16_t templateElementLen;
  ElementFormat elementFormat; /* Only for elements longer than 4 bytes */
  ElementDumpFormat fileDumpFormat; /* Hint when data has to be printed on
                                       a human readable form */
  char      *templateElementName, *templateElementDescr;
} V9V10TemplateElementId;
```

Le variabili della struttura hanno il seguente significato:

isOptionTemplate Specifica se il *template* è un *option template* (settata a 1) o un *flow template* (settata a 0). Si possono usare le due costanti `FLOW_TEMPLATE` e `OPTION_TEMPLATE`, definite in `nprobe.h` come segue:

```
#define FLOW_TEMPLATE       0
#define OPTION_TEMPLATE     1
```

useLongSnaplen *nProbe* è in grado di passare al plugin tutto il *payload* del pacchetto oppure solo una piccola parte. Nel caso in cui un plugin voglia fare *packet inspection* e necessiti di tutto il *payload* contenuto nel pacchetto deve settare questa opzione a 1, altrimenti se necessita solo di una piccola parte iniziale può settare questa opzione a 0. Alternativamente ai valori numerici si possono usare le costanti definite nel file `nprobe.h`:

```
#define SHORT_SNAPLEN       0
#define LONG_SNAPLEN        1
```

Si noti che anche nel caso in cui questa opzione è settata a `LONG_SNAPLEN`, il *payload* ha comunque una dimensione massima che non può andare oltre il valore della costante:

```
#define PCAP_LONG_SNAPLEN       1600
```

definita nel file `engine.h`.

templateElementEnterpriseId l'identificativo univoco dell'azienda che ha creato il plugin (ad esempio il *PEN* rilasciato dalla *IANA*).

templateElementId l'identificativo univoco per questa informazione (il suo *id*).

variableFieldLength specifica se l'informazione ha dimensione fissa o variabile. Anche in questo caso si può usare una delle due costanti `STATIC_FIELD_LEN` o `VARIABLE_FIELD_LEN`, definite in `nprobe.h` come segue:

```
#define STATIC_FIELD_LEN    1
#define VARIABLE_FIELD_LEN  2
```

templateElementLen la dimensione massima, in byte, che occupa quest'informazione. Anche se la precedente variabile `variableFieldLength` è stata settata a `VARIABLE_FIELD_LEN`, occorre comunque informare `nProbe` della sua dimensione massima.

elementFormat specifica il formato di questa informazione. Il tipo di questa variabile è `ElementFormat`, che è un *enumerate* definito sempre in `nprobe.h`:

```
typedef enum {
  ascii_format = 0,
  hex_format,
  numeric_format,
  ipv6_address_format
} ElementFormat;
```

fileDumpFormat indica come dev'essere stampata l'informazione in un formato interpretabile dall'uomo. Il tipo di questa variabile è `ElementDumpFormat`, che è un *enumerate* definito in `nprobe.h` come segue:

```
typedef enum {
  dump_as_uint = 0, /* 1234567890 */
  dump_as_formatted_uint, /* 123'456 */
  dump_as_ip_port,
  dump_as_ip_proto,
  dump_as_ipv4_address,
  dump_as_ipv6_address,
  dump_as_mac_address,
  dump_as_epoch,
  dump_as_bool,
  dump_as_tcp_flags,
  dump_as_hex,
  dump_as_ascii
} ElementDumpFormat;
```

templateElementName il nome univoco che identifica questa informazione (quello che viene usato nel *template* utente passato con l'opzione `-T`).

templateElementDescr una descrizione sull'informazione che si esporta.

È molto importante scegliere bene sia la variabile `templateElementId` che `templateElementName` perché, come detto, devono essere univoche. Per la seconda si usa anteporre il nome del plugin in modo da stare relativamente sicuri (ad esempio: `SMTP_MAIL_FROM` o `SMTP_RCPT_TO`). Per la prima invece bisogna assicurarsi che l'*id* non sia già stato usato. L'unica informazione usabile è una base di partenza, definita nel file `nprobe.h` dalla costante `NTOP_BASE_ID`:

```
#define NTOP_BASE_ID 57472
```

Dato che un plugin generalmente esporta più di un'informazione, queste strutture `V9V10TemplateElementId` devono essere inserite in un'array terminato dalla particolare struttura:

```
{ FLOW_TEMPLATE, NTOP_ENTERPRISE_ID, 0, STATIC_FIELD_LEN, 0, 0, 0, NULL, NULL }
```

Ora si possono capire meglio le due funzioni dell'interfaccia dei plugin, `pluginFlowConf()` e `getPluginTemplateFctn()`. La prima ritorna l'array contenente tutte le strutture dati `V9V10TemplateElementId` per le informazioni definite dal plugin, mentre la seconda prende come parametro il nome di un'informazione, controlla per ogni struttura la variabile `templateElementName`, e se ne trova una uguale al parametro passatogli, allora ritorna il puntatore alla struttura `V9V10TemplateElementId` associata, altrimenti ritorna `NULL`.

nProbe, per sapere se deve attivare un plugin o meno, esegue l'algoritmo seguente (si veda la figura 2): per ogni elemento passato nel *template* utente tramite l'opzione `-T`, invoca la funzione `getPluginTemplateFctn()` di un plugin e, se questo ritorna una struttura `V9V10TemplateElementId` valida (cioè diversa da `NULL`), lo attiva.

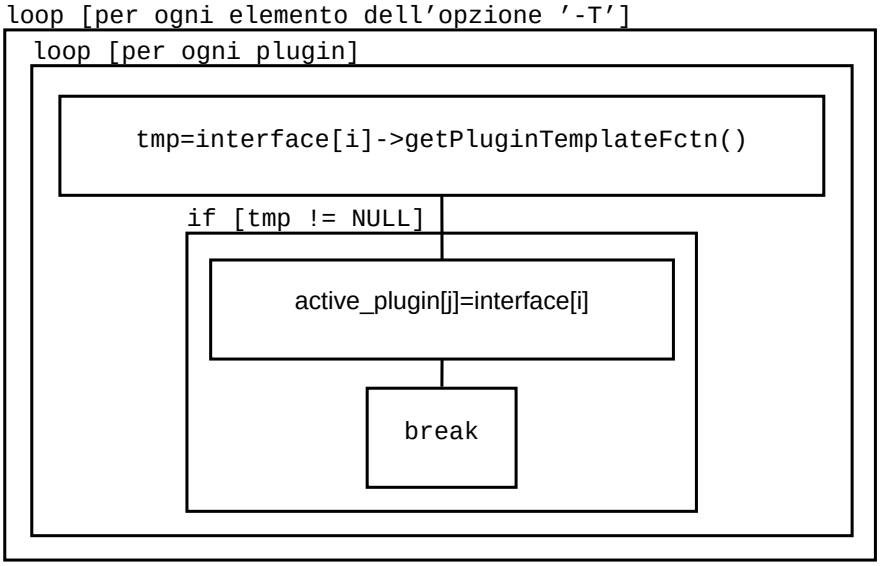

Figura 2: Algoritmo di attivazione dei plugin

1.2.4 I dati privati

Alla creazione di un nuovo flusso, *nProbe* crea una struttura dati di tipo `FlowHashBucket` dichiarata nel file `bucket.h`. Questa stuttura dati è molto complessa ed è quasi esclusivamente di uso interno. Per

quanto riguarda i plugin tutto ciò che importa sapere è la presenza all'interno di questa struttura dati della variabile `plugin`: una lista concatenata di strutture dati di tipo `PluginInformation`. Se un plugin è interessato ad analizzare il flusso appena creato, deve definire un oggetto di tipo `PluginInformation` e concatenarlo alla lista puntata dalla variabile `plugin`.

Prima di vedere più in dettaglio cosa avviene dal punto di vista dei plugin durante la creazione di un nuovo flusso, è importante vedere la struttura dati `PluginInformation` dichiarata sempre nel file `bucket.h`:

```
typedef struct pluginInformation {
  struct pluginInfo *pluginPtr;
  void *pluginData;
  struct pluginInformation *next;
} PluginInformation;
```

La struttura ha tre variabili:

pluginPtr: un puntatore all'interfaccia del plugin (la sua struttura dati `PluginInfo`).

pluginData: un puntatore a dei dati privati che il plugin vuole associare al flusso.

next: il prossimo elemento della lista.

Durante la creazione di un flusso avviene quanto segue:

- *nProbe* crea la struttura dati `FlowHashBucket` settando la variabile `plugin` a `NULL`.
- Per ogni plugin attivo, invoca la funzione `packetFlowFctn()` passandogli la struttura `FlowHashBucket` e una variabile (`new_bucket`) settata appositamente (a 1) per far capire al plugin che si tratta di un nuovo flusso.
- Il plugin, se è interessato a trattare il flusso:
  - definisce una variabile di tipo `PluginInformation`.
  - collega alla variabile `pluginPtr` la sua struttura dati `PluginInfo`.
  - eventualmente crea una struttura dati privata da collegare alla variabile `pluginData`.
  - si inserisce nella lista concatenata `plugin` di `FlowHashBucket`.

In questo modo *nProbe* ha nella variabile `plugin` di ogni flusso, la lista concatenata dei plugin attivi per quello specifico flusso.

Per rendere più chiaro ciò che avviene, ecco un estratto dal *MySQL protocol dissector*:

```
/* La struttura per i dati privati del plugin */
struct plugin_data
{
  ...
};

static void
mysqlPlugin_packet(u_char new_bucket,
```

```
   FlowHashBucket* bkt,
                   u_short sport,
                   u_short dport,
                   ...)
{
  struct plugin_data *p_data; /* dati privati del plugin */

  /* Testa se il flusso è interessante:
     ovvero se almeno una delle porte è quella del server MySQL */
  if ( sport != MYSQL_SERVER_PORT && dport != MYSQL_SERVER_PORT )
    return; /* Nothing to be done */

  ...

  if ( new_bucket ) /* Se è un nuovo flusso */
    {
      /* Crea la struttura dati PluginInformation */
      PluginInformation *info;
      if( ! (info = (PluginInformation*)malloc(sizeof(PluginInformation))) )
        {
          traceEvent(TRACE_ERROR, "Not enough memory?");
          return; /* Not enough memory */
        }

      /* Colloga la sua PluginInfo alla variabile pluginPtr */
      info->pluginPtr  = (void*)&mysqlPlugin;

      /* Crea la struttura per i dati privati e la collega alla variabile pluginData */
      if ( ! (p_data = info->pluginData = malloc(sizeof(struct plugin_data))) )
        {
          traceEvent(TRACE_ERROR, "Not enough memory?");
          free(info);
          return; /* Not enough memory */
        }

      /* Collega le sue informazioni alla lista dei plugin attivi per questo flusso */
      info->next = bkt->plugin;
      bkt->plugin = info;
    }
```

1.2.5 Esportazione forzata delle informazioni

L'esportazione delle informazioni su di un flusso è generalmente segnata dalla "morte" del flusso stesso o quando lo si ritiene scaduto. Questo però permette di ottenere solo delle informazioni aggregate sul flusso e non delle informazioni specifiche: si può sapere quanti byte in totale si sono scambiati server e client e non cosa si sono scambiati in un dato momento.

nProbe mette a disposizione dei plugin una funzione che forza l'esportazione delle informazioni pur eventualmente mantenendo in memoria i dati interni relativi al flusso stesso. La funzione è definita nel file `engine.c` ed ha il seguente prototipo:

```
void exportBucket(FlowHashBucket *myBucket, u_char free_memory);
```

Dove `myBucket` è il flusso che si vuole esportare, mentre `free_memory` è una variabile booleana che,

se settata a 1 indica di rimuovere dalla memoria le informazioni sul flusso[4], mentre se viene settata a 0 i dati relativi al flusso rimarranno comunque in memoria.

L'esportazione di un flusso implica l'invocazione per ogni plugin attivo su quel flusso della sua funzione `checkPluginExportFctn()`, ed eventualmente della funzione `checkPluginPrintFctn()`. In questo modo il plugin è in grado di dire al collezionatore cosa si sono scambiati in un determinato periodo il server e il client.

1.2.6 Un plugin di base

Per riassumere tutto ciò che si è visto in questo paragrafo, ecco il corpo vuoto di un plugin:

```
#include "nprobe.h" /* Include anche engine.h */

#define BASE_ID            NTOP_BASE_ID+195
#define STRING_MAX_LEN     32     /* Massima dimensione per le stringhe */

static V9V10TemplateElementId myPlugin_template[] =
  {
    { FLOW_TEMPLATE, LONG_SNAPLEN, NTOP_ENTERPRISE_ID, BASE_ID, VARIABLE_FIELD_LEN,
      STRING_MAX_LEN, ascii_format, dump_as_ascii, "MY_INFO", "Info from packet" },
    { FLOW_TEMPLATE, NTOP_ENTERPRISE_ID, 0, STATIC_FIELD_LEN, 0, 0, 0, NULL, NULL }
  };

static PluginInfo myPlugin; /* Definita successivamente */

/* Invocata quando il plugin viene inizializzato */
static void
myPlugin_init(int argc, char *argv[])
{}

/* Invocata quando il plugin viene terminato */
static void
myPlugin_term(void)
{}

/* Invocata per prelevare gli attributi sulle informazioni esportate dal plugin */
static V9V10TemplateElementId *
myPlugin_conf(void)
{
  return(myPlugin_template);
}

/* Invocata quando un flusso viene eliminato */
static void
myPlugin_delete(FlowHashBucket *bkt, void *pluginData)
{}

/* Invocata quando un pacchetto viene ricevuto */
static void
myPlugin_packet(u_char new_bucket, void *pluginData,
   FlowHashBucket *bkt,
   u_short proto, u_char isFragment,
```

[4]Attenzione a settare la variabile a 1. Questo implica la chiamata per ogni plugin attivo su quel flusso della funzione `deleteFlowFctn()` e la rimozione da parte di *nProbe* della struttura dati `FlowHashBucket`. Se però il flusso non è effettivamente terminato, l'arrivo di nuovi dati implicheranno la creazione di un nuovo flusso con tutto l'iter che ne consegue.

```
   u_short numPkts, u_char tos,
   u_short vlanId, struct eth_header *ehdr,
   IpAddress *src, u_short sport,
   IpAddress *dst, u_short dport,
   u_int len, u_int8_t flags, u_int8_t icmpType,
   u_short numMplsLabels,
   u_char mplsLabels[MAX_NUM_MPLS_LABELS][MPLS_LABEL_LEN],
   char *fingerprint,
   const struct pcap_pkthdr *h, const u_char *p,
   u_char *payload, int payloadLen)
{}

/* Invocata all'inizio per capire se il plugin dev'essere attivato */
static V9V10TemplateElementId *
myPlugin_get_template(char *template_name)
{
  int i;

  for(i=0; myPlugin_template[i].templateElementId != 0; i++)
    if(!strcmp(template_name, myPlugin_template[i].templateElementName))
      return(&myPlugin_template[i]);

  return(NULL); /* Unknown */
}

/* Invocata quando il flusso viene esportato */
static int
myPlugin_export(void *pluginData, V9V10TemplateElementId *theTemplate,
   int direction /* 0 = src->dst, 1 = dst->src */,
   FlowHashBucket *bkt, char *outBuffer,
   uint* outBufferBegin, uint* outBufferMax)
{}

/* Invocata quando il flusso viene stampato su di un file */
static int
myPlugin_print(void *pluginData, V9V10TemplateElementId *theTemplate,
  int direction /* 0 = src->dst, 1 = dst->src */,
  FlowHashBucket *bkt, char *line_buffer, uint line_buffer_len)
{}

/* FIXME(invocata quando?) */
static void
myPlugin_setup(void)
{}

/* FIXME(invocata quando?) */
static void
myPlugin_help(void)
{}

/* La struttura PluginInfo da far avere a nProbe per fargli conoscere
 * gli indirizzi delle funzioni di interfacciamento */
static PluginInfo myPlugin =
  {
    NPROBE_REVISION,
    "My Protocol Dissector",
    "0.1",
    "Handle My protocol",
    "My <my.email@my.domain.org>",
```

```
    0 /* non sempre abilitato */, 1, /* abilitato */
    myPlugin_init,
    myPlugin_term,
    myPlugin_conf,
    myPlugin_delete,
    1, /* invoca la funzione packetFlowFctn per ogni pacchetto del flusso */
    myPlugin_packet,
    myPlugin_get_template,
    myPlugin_export,
    myPlugin_print,
    myPlugin_setup,
    myPlugin_help
  };

/* La "funzione cerniera" tra nProbe e il plugin (ritorna la struttura PluginInfo) */
#ifdef MAKE_STATIC_PLUGINS
PluginInfo *myPluginEntryFctn(void)  /* Se il plugin è inserito staticamente nel sorgente di nProbe */
#else
PluginInfo *PluginEntryFctn(void) /* Se il plugin viene caricato dinamicamente a run-time */
#endif
{
  return(&myPlugin);
}
```

1.3 Cosa tenere a mente durante la stesura di un plugin

Va ricordato che *nProbe* è una sonda e cattura i pacchetti che passano dalla rete. Durante lo sviluppo di un plugin per l'analisi di un particolare protocollo, si deve tener sempre presente che il plugin:

- `Deve essere stateless`. La sonda in generale non riceve i pacchetti così come li ricevono gli host che generano il flusso. Il client e il server sanno in ogni istante cosa aspettarsi dal pacchetto successivo, ma la sonda potrebbe essere stata attivata, ad esempio, nel mezzo della comunicazione. Pertanto è un errore scrivere un plugin che implementa un automa a stati per l'analisi del contenuto. Affermazioni come: "dato che prima è passato il pacchetto A, questo è sicuramente il pacchetto B", sono, in generale, sbagliate.

- `Deve stare attento alla frammentazione dei pacchetti`. Un flusso di informazioni potrebbe non essere inviato tutto in un solo pacchetto. Se, ad esempio, il client e il server usano una connessione TCP e il client invia una richiesta superiore al *maximum transmission unit (MTU)*, allora il kernel del client suddividerà il flusso di richiesta in più pacchetti TCP. Il kernel del server dall'altra parte "fonderà" questi pacchetti e li farà ricevere all'applicazione come un unico flusso di dati continuo. Questa suddivisione del flusso è, quindi, completamente trasparente alle applicazioni client e server, ma il plugin, per quanto detto al punto precedente, deve occuparsi anche di questa eventualità. Non solo la sonda potrebbe aver perso uno dei pacchetti, ma tenendo conto dell'inaffidabilità della rete, potrebbe prendere i pacchetti in un ordine casuale, oppure prendere un duplicato del pacchetto precedente[5].

- `Non si deve fidare dei pacchetti`. Il plugin non deve basarsi solo sulle specifiche del protocollo per analizzare i pacchetti, ma deve stare sempre attento che ciò che analizza sia

[5] Ad esempio se il protocollo di trasporto è TCP e un pacchetto non viene ricevuto.

effettivamente un pacchetto valido. Ad esempio, se nelle specifiche del protocollo viene detto che una stringa e null-terminated, il plugin potrebbe usare una strlen() per contare quanto è grande la stringa, ma se non tiene conto del fatto che il pacchetto che sta analizzando potrebbe non essere di quel protocollo e che, quindi, il carattere di fine stringa potrebbe non esserci affatto, allora la strlen() potrebbe andare in buffer overflow determinando un segmentation fault del plugin e il conseguente crash della sonda.

2 Il protocollo MySQL

In questo paragrafo verrà analizzato, a grandi linee, il protocollo MySQL[6]. Questo permetterà di capire in che modo funziona il plugin per l'analisi dei pacchetti MySQL e da dove preleva le informazioni che esporta (l'username dell'utente che si collega al server, il database che sta interrogando, etc...).

È importante notare fin da subito che esistono delle differenze sostanziali tra le versioni minori e quelle maggiori o uguali alla 4.1 del protocollo. Alcune di queste differenze sono trasparenti allo scopo del plugin, altre, come si vedrà, saranno gestite.

2.1 L'handshake

Dopo l'handshake a livello TCP, viene effettuato un secondo handshake a livello di protocollo MySQL. In questa fase vengono inviati esattamente i 3 pacchetti[7] seguenti:

1. Il server, tra le altre cose, informa il client sulla propria versione, sulla versione del protocollo che è in grado di gestire, sul linguaggio accettato e sull'identificativo del thread che gestisce la connessione. Questo pacchetto è conosciuto come *greeting packet*.

2. Il client a sua volta invia al server delle informazioni utili per la connessione, come ad esempio la dimensione massima dei pacchetti, in byte, che è in grado di accettare. Oltre a queste informazioni, invia anche le credenziali dell'utente (username e password) necessarie per l'autenticazione[8]. Per quest'ultimo motivo il pacchetto viene conosciuto come *client authentication packet* o *login packet*. Se entrambi, client e server, gestiscono la versione 4.1 del protocollo[9], allora in questo pacchetto il client può inviare anche il nome del database sul quale intende lavorare. Ancora, il formato del pacchetto è differente a seconda che si usino delle versioni superiori o uguali alla 4.1 del protocollo, oppure no.

3. Il server conclude l'handshake inviando al client un pacchetto chiamato *result packet*. Se è in grado di soddisfare le richieste del client e l'autenticazione è avvenuta con successo, la connessione va avanti, altrimenti il server nega l'accesso e chiude la connessione.

[6] Maggiori informazioni possono essere prelevate dal sito:
http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol

[7] In questo contesto i pacchetti sono intesi come pacchetti del protocollo MySQL.

[8] L'autenticazione in realtà inizia dal *greeting packet* ed è differente a seconda che vengano usati protocolli inferiori al 4.1 oppure no.

[9] Si noti che in questa fase, il client sa già quale versione del protocollo sarà usata, perché conosce sia la sua, che quella del server (che gli è stata inviata nel *greeting packet*).

Sia il server (nel *greeting packet*) che il client (nel *login packet*) inviano una maschera di bit di 2 byte che contiene svariate informazioni, tra cui un flag che indica se sono in grado di gestire il protocollo 4.1 o superiore. La maschera inviata dal server viene chiamata *server_capabilities*, mentre quella inviata dal client prende il nome di *client_flag* e, nel caso in cui il protocollo usato è il 4.1 o superiore, dopo *client_flag* il client invia altri 2 byte conosciuti come *client_extended_flag*.

2.2 L'header dei pacchetti

Tutti i pacchetti MySQL hanno un'intestazione fissa di 4 byte. I primi 3 byte indicano la dimensione, in byte, del pacchetto stesso e, dato che questa informazione è un numero intero non negativo, la dimensione massima di un pacchetto MySQL è di $2^{24} = 16$MB. Ai fini del plugin, questo vuol dire che un pacchetto MySQL può essere frammentato in più pacchetti TCP.

L'ultimo byte dell'intestazione indica il numero di sequenza del pacchetto. Questo numero viene azzerato alla fine di ogni comunicazione. Quindi, ad esempio, i tre pacchetti dell'handshake sono numerati 0, 1 e 2 e, dopo di ciò, la sequenza viene riazzerata.

2.3 I pacchetti di richiesta (*command packet*)

Se l'handshake è avvenuto con successo, il client può iniziare a interrogare il server inviando dei pacchetti conosciuti come *command packet*. Il primo byte di questo pacchetto identifica il comando, mentre i rimanenti byte ne sono l'argomento.

2.4 I pacchetti di risposta (*result packet*)

Ricevuta la richiesta dal client, il server risponde con quello che viene chiamato genericamente un *result packet* (come quello che termina l'handshake). Il primo byte di questo pacchetto è chiamato *field_count* e identifica il tipo di risposta che il server da al client. A seconda del valore del *field_count* questo pacchetto viene chiamato:

Ok Packet: indica che tutto è andato bene, lo si riconosce perché il *field_count* è 0x00.

Error Packet: indica che c'è stato un errore, lo si riconosce perché il *field_count* è 0xFF.

Result Set Header Packet: è l'intestazione del risultato di una query. Il *field_count* ha un valore compreso tra 0x01 a 0xFD e indica il numero di colonne della tabella risultante[10].

Nel caso di un `Result Set Header Packet`, il server invierà tutti i pacchetti (incrementando sempre il numero di sequenza) che permetteranno al client di ottenere l'intera tabella del risultato della query. Il server invierà dapprima tanti pacchetti per quanto era il valore del *field_count* del `Result Set Header Packet`. Questa prima serie di pacchetti, ognuno dei quali è chiamato *Field Packet*, definisce i campi delle colonne della tabella (ad esempio, id di tipo intero, nome di tipo stringa, etc...). La fine di questa prima serie di pacchetti viene decretata dall'invio del pacchetto chiamato *EOF Packet* (il cui primo byte vale 0xFE) che, a volte, può servire per trasportare

[10] Potrebbe sembrare che il numero di colonne del risultato sia al più 253 (0xFD). In realtà questo singolo byte può anche indicare quanti byte successivi devono essere presi per calcolare un numero intero che arriva sino a 2^{64}.

delle informazioni. Il server quindi invia al client un'altra serie di pacchetti chiamati *Data Packet*, ognuno dei quali contiene una riga della tabella (quindi il valore della prima colonna, il valore della seconda colonna, etc...). Ed infine, per terminare la sequenza dei *Data Packet* il server invia un altro pacchetto di *EOF Packet*.

La figura 3 mostra un esempio di connessione MySQL in cui, dopo l'handshake, il client seleziona un database ed esegue una SELECT. Si noti come, grazie alle informazioni ricevute, il client è in grado di creare la tabella risultante.

3 Il plugin *MySQL Protocol Dissector*

Dopo il background acquisito su *nProbe* e sul protocollo MySQL si può passare a vedere il plugin *MySQL protocol dissector*.

3.1 Le informazioni esportate

Attualmente il plugin è in grado di esportare le seguenti informazioni sul traffico MySQL (sono i nomi univoci da passare a *nProbe* come parte del *template*):

MYSQL_SERVER_VERSION: la versione del server MySQL. Viene passata dal server al client nel *greeting packet*.

MYSQL_USERNAME: il nome dell'utente che si è collegato al server. Viene passato dal client al server nel *login packet*.

MYSQL_DB: il database al quale si è attualmente collegati. Ci sono due modi in cui il client seleziona un database: se server e client gestiscono il protocollo di versione superiore o uguale al 4.1, il client può selezionare un database direttamente nel *login packet*; altrimenti il client invia un *command packet* il cui comando è *use database*.

MYSQL_REQUEST: la versione human-readable della richiesta fatta dal client. Come detto, il primo byte del *command packet* identifica il comando che il client chiede di eseguire al server. Qui viene esportata una versione più leggibile di questa richiesta così, ad esempio, se il primo byte vale 0x03, e cioè se il client sta chiedendo al server di eseguire una query al database, l'informazione esportata sarà la stringa: "Query".

MYSQL_QUERY: l'eventuale query che deve essere eseguita. Sono tutti i byte successivi al primo del *command packet* nel caso in cui il primo byte valga 0x03.

MYSQL_REQUEST: la risposta del server. E' il *field_count* (il primo byte) del *result packet* che, come si è visto, ha significati differenti a seconda della richiesta fatta dal client.

Queste informazioni vengono esportate dal plugin ad ogni *result packet*. Se l'handshake viene catturato tutto correttamente si può quindi sapere la versione del server MySQL, l'username dell'utente, l'eventuale database selezionato nel *login packet* e la risposta del server che informa il client se la connessione è stata accettata o meno. Durante l'esecuzione normale di richiesta-risposta tra

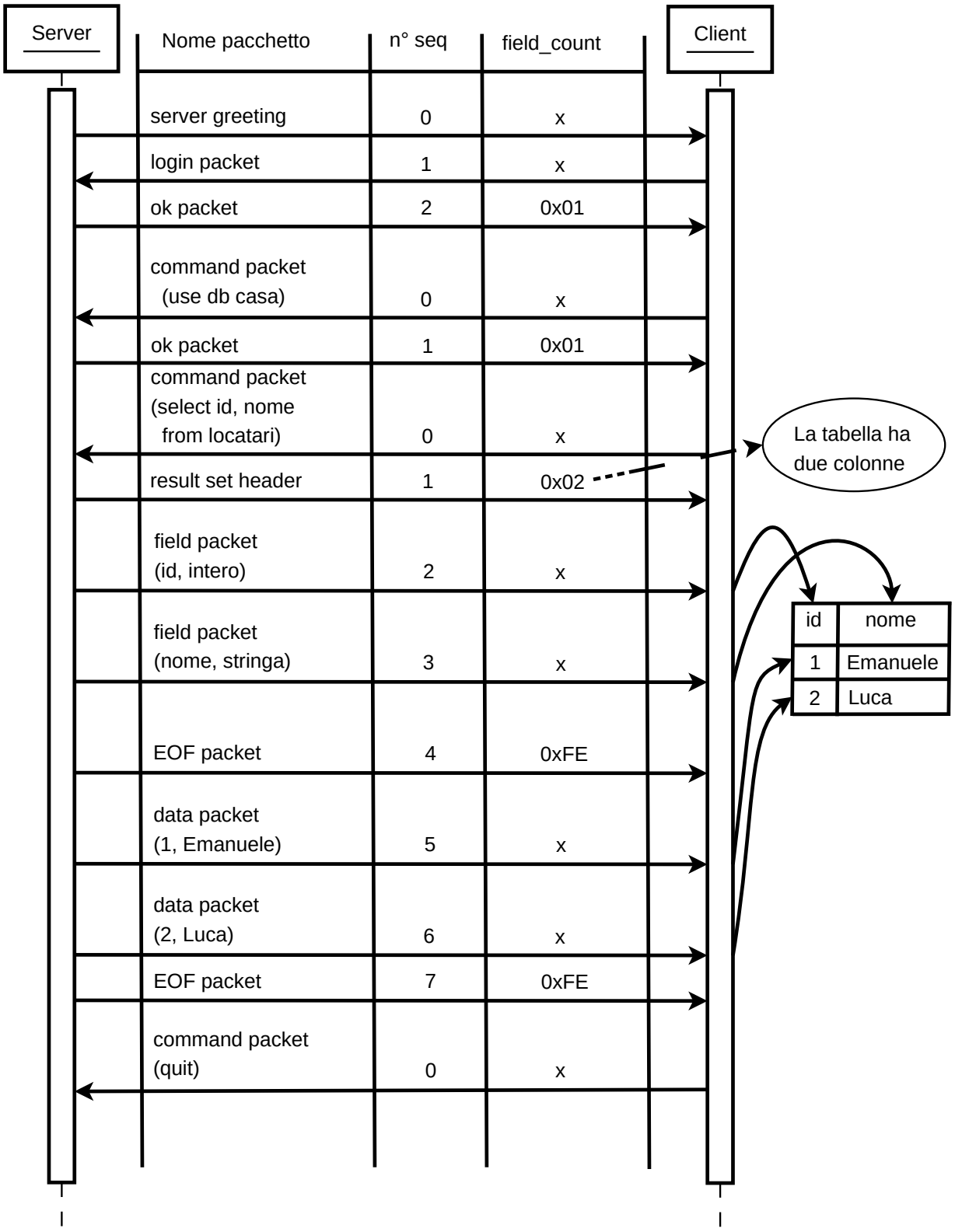

Figura 3: Esempio di una connessione MySQL

client e server invece, si possono analizzare la richiesta fatta dal client con l'eventuale query e il suo esisto.

3.2 L'analisi del traffico

Dalle specifiche del protocollo MySQL e dalle informazioni che il plugin è in grado di esportare, si nota che tutto ciò che serve si trova all'inizio del pacchetto. Per risolvere il problema della frammentazione del pacchetto MySQL in più pacchetti TCP e di tutto quello che ne consegue (ritrasmissione dei pacchetti TCP errati, riordino dei pacchetti TCP, etc...), il plugin tiene in memoria solo il primo (o meglio parte del primo) degli eventuali pacchetti TCP diretti dal client al server, o viceversa.

Il plugin riconosce la fine di una serie di pacchetti TCP quando la direzione cambia; questo perché il server non risponde sino a che tutti i pacchetti TCP non gli sono stati recapitati con successo (ovvero fino a che non ha ricostruito tutto il pacchetto MySQL), e viceversa, il client non richiede l'esecuzione di un altro comando se non ha letto tutte le risposte del server. Sino a che la direzione del "discorso" non cambia, il plugin si preoccupa solo di controllare che il pacchetto TCP che la sonda cattura non sia precedente a quello che lui pensa essere il primo della serie; questo avviene quando la rete mischia i pacchetti TCP facendoli arrivare in disordine alla sonda. Quando poi la direzione cambia, il plugin effettua l'analisi del pacchetto in memoria ed eventualmente esegue l'esportazione delle informazioni. Vale la pena notare che il pacchetto precedente non è sempre quello con il numero di sequenza minore, infatti, il numero di sequenza è un numero positivo intero ma a 32 bit, il fatto che sia limitato superiormente implica che prima o poi questo numero ricomincerà dall'inizio e, se questo avviene, il primo pacchetto non è più quello che ha il numero di sequenza più piccolo in assoluto, ma quello con il numero di sequenza più piccolo tra quelli più grandi. Per tenere conto di questa eventualità, il controllo effettuato dal plugin è il seguente:

```
if ( tcpSeqNum < p_data->tcp_info.firstPktSeqNum &&
     (tcpSeqNum + 8388608) > p_data->tcp_info.firstPktSeqNum )
  /* E' un nuovo primo */
```

Si controlla che il numero di sequenza attuale (`tcpSeqNum`) sia effettivamente più piccolo di quello che il plugin pensa essere il primo pacchetto della serie (`p_data->tcp_info.firstPktSeqNum`). Inoltre si controlla che aggiungendo al numero di sequenza del pacchetto attuale, un numero relativamente grande ($2^{23} = 8388608$), questo risulti effettivamente più grande di quello salvato, cosa che non avviene nel caso in cui il numero di sequenza è stato resettato[11].

Ovviamente osservare il flusso del "discorso" implica che il plugin non è completamente stateless in quanto si aspetta di vedere un cambio di direzione. L'analisi del traffico TCP però è molto complicata e potrebbe portare via molte risorse. Si ricordi che il plugin gira su una sonda che potrebbe avere attivi innumerevoli flussi nello stesso periodo di tempo, i plugin non devono quindi "rubare" molte risorse per compiere il loro lavoro. L'algoritmo scelto è sembrato essere un buon compromesso tra efficienza e coerenza dei dati esportati, infatti, se ad esempio la sonda dovesse

[11] Si ricordi che il numero di sequenza va di pari passo con i byte inviati. Un pacchetto MySQL come detto in precedenza non può essere più grande di 2^{24} byte quindi, nell'eventualità che il numero di sequenza sia stato resettato, aggiungendo 2^{23} a questo numero non lo farà mai essere più grande del più piccolo tra i più grandi.

perdere l'unico pacchetto TCP che segna il cambio del "discorso", quello che succede è che il plugin pensa che la direzione non sia mai cambiata e quindi le informazioni che esporta in un dato momento potrebbero essere errate. Questo però non ha impatto sul futuro, dove se non ci sono altri errori di cattura, il plugin si riallineerà al "discorso".

3.2.1 In ritardo di uno

L'analisi del pacchetto viene effettuata, come detto, solo al cambio di direzione del "discorso" tra il client e il server. Nel momento in cui il flusso scade però, questo cambio di direzione non avviene, ed il plugin deve tener conto anche di questa eventualità. *nProbe* invoca comunque le funzioni `exportFcnt()` e `printFcnt()` quando il flusso è scaduto e durante l'esecuzione di queste funzioni, quindi, va controllato se il pacchetto conservato in memoria è stato analizzato oppure no, altrimenti si rischia di esportare delle informazioni vecchie.

3.3 Le strutture dati private

Durante l'analisi del traffico MySQL si possono individuare tre tipi di dati distinti:

I dati statici della connessione MySQL: quei dati che vengono scambiati durante l'handshake e che rimangono uguali per tutta la connessione[12].

I dati relativi alla connessione TCP: sono i dati necessari a seguire il "discorso" a livello di pacchetti TCP.

I dati dinamici della connessione MySQL: quali ad esempio, il comando richiesto dal client, l'eventuale query, la risposta al comando, etc....

Per ognuno di questi tipi di dati il plugin ha una struttura dati ben definita:

```
/* Informazioni statiche della connessione MySQL (prelevate durante l'handshake) */
struct mysql_connection_info
{
  char *mysql_server_version;   /* la versione del server */
  char *mysql_username;         /* l'username dell'utente collegato */
  u_int16_t s_caps;             /* server capability */
  u_int16_t c_caps;             /* client capability (client flag) */
  u_int16_t c_ext_caps;         /* ciente extended capability (client extended flag) */
};

/* Informazioni del flusso TCP */
struct mysql_tcp_info
{
  u_short src_port;                                 /* porta sorgente */
  u_int32_t firstPktSeqNum;                         /* numero di sequenza del primo pacchetto */
  u_char firstPktPayload[MYSQL_TCP_PKT_MAX_SIZE+1]; /* primo pacchetto del "discorso" (primi byte) */
  int firstPktPayloadLen;                           /* dimensione, in byte, del primo pacchetto salvato */
  u_char wasDissected;                              /* il pacchetto è stato analizzato (1) o no (0)? */
};
```

[12]Tra queste informazioni il plugin mette anche l'username dell'utente collegato al database. In realtà esiste un comando chiamato *change user* che permette di cambiare l'utente a connessione avviata.

```
/* Struttura dati privata collegata al flusso
 * (contiene le informazioni dinamiche della connessione MySQL).
 */
struct plugin_data
{
  char *mysql_db;                       /* il database attualmente in uso */
  u_int8_t mysql_command;               /* comando di cui client chiede l'esecuzione */
  char *mysql_query;                    /* eventuale query fatta dal client */
  int16_t mysql_response;               /* risposta del server (si notino i 2 byte) */
  struct mysql_connection_info c_info;  /* informazioni statiche sul flusso MySQL */
  struct mysql_tcp_info tcp_info;       /* informazioni sul flusso TCP */
};
```

Il plugin collega solo la struttura `plugin_data` alla variabile `pluginData` della `PluginInformation` relativa al flusso (si veda 1.2.4), le altre strutture dati come si vede sono contenute in essa sotto forma di variabili.

C'è un altra struttura dati che il plugin usa durante l'analisi del pacchetto MySQL, quella che contiene le informazioni sull'header:

```
/* Header del pacchetto MySQL */
struct mysql_packet_header
{
  u_int32_t length; /* dimensione del pacchetto (solo 24 bit) */
  u_char number;    /* numero di sequenza del pacchetto */
};
```

3.4 Il plugin in azione

Ecco l'output prodotto dall'esecuzione di *nProbe* su un pacchetto MySQL grazie all'ausilio del plugin:

```
$> nprobe -p 1/1/1/1/0/1 -T "%PROTOCOL %IPV4_SRC_ADDR %IPV4_DST_ADDR %L4_SRC_PORT %L4_DST_PORT\
   %MYSQL_SERVER_VERSION %MYSQL_USERNAME %MYSQL_DB %MYSQL_REQUEST %MYSQL_QUERY %MYSQL_RESPONSE"\
   -P /tmp -i ../samples/mysql_complete.pcap
05/Oct/2010 11:00:28 [nprobe.c:2548] Welcome to nprobe v.5.8.2 ($Revision: 1700 $) for\
                                    x86_64-unknown-linux-gnu
05/Oct/2010 11:00:28 [nprobe.c:2600] Dumping flow files every 60 sec into directory /tmp
05/Oct/2010 11:00:28 [nprobe.c:2606] WARNING: -n parameter is missing. 127.0.0.1:2055 will be used.
05/Oct/2010 11:00:28 [plugin.c:137] Loading plugins [.so] from ./plugins
05/Oct/2010 11:00:28 [dbPlugin.c:72] Initializing DB plugin
05/Oct/2010 11:00:28 [util.c:295] GeoIP: loaded AS config file GeoIPASNum.dat
05/Oct/2010 11:00:28 [plugin.c:580] 1 plugin(s) enabled
05/Oct/2010 11:00:28 [plugin.c:511] Enabling plugin MySQL Protocol Dissector
05/Oct/2010 11:00:28 [engine.c:1695] Saving flows into temporary file '/tmp/2010/10/05/11/00.flows.temp'
05/Oct/2010 11:00:29 [engine.c:1554] Flow file '/tmp/2010/10/05/11/00.flows' is now available

$> sed -n '1 b print;/|56162|3306|/b print;d;:print {s/\([^|]*|\)\{5\}//;p;}' /tmp/2010/10/05/11/00.flows
MYSQL_SERVER_VERSION|MYSQL_USERNAME|MYSQL_DB|MYSQL_REQUEST|MYSQL_QUERY|MYSQL_RESPONSE
5.0.54|tfoerste||Query|select @@version_comment limit 1|1
5.0.54|tfoerste||Query|SELECT DATABASE()|1
5.0.54|tfoerste|test|Use Database||0
5.0.54|tfoerste|test|Query|show databases|1
5.0.54|tfoerste|test|Query|show tables|1
5.0.54|tfoerste|test|Show Fields||3
5.0.54|tfoerste|test|Query|create table foo (id BIGINT( 10 ) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,\
                          animal VARCHAR(64) NOT NULL, name VARCHAR(64) NULL DEFAULT NULL) ENGINE = MYISAM|0
```

```
5.0.54|tfoerste|test|Query|insert into foo (animal, name) values ("dog", "Goofy")|0
5.0.54|tfoerste|test|Query|insert into foo (animal, name) values ("cat", "Garfield")|0
5.0.54|tfoerste|test|Query|select * from foo|3
5.0.54|tfoerste|test|Query|delete from foo where name like '%oo%'|0
5.0.54|tfoerste|test|Query|delete from foo where id = 1|0
5.0.54|tfoerste|test|Query|select count(*) from foo|1
5.0.54|tfoerste|test|Query|select * from foo|3
5.0.54|tfoerste|test|Query|delete from foo|0
5.0.54|tfoerste|test|Query|drop table foo|0
5.0.54|tfoerste|test|Quit||-1
```

Alcuni piccoli commenti sull'output:

- Per le prime due righe[13] di output manca il nome del database in uso, questo vuol dire che durante l'handshake il client non ha selezionato nessun database.
- La terza riga indica che l'utente ha chiesto di utilizzare il database *test* e il server ha risposto con un *Ok packet* (*field_count* = 0x00).
- Una query interessante è riportata dal seguente output:

  ```
  5.0.54|tfoerste|test|Query|select * from foo|3
  ```

  dal quale si riesce a capire che la tabella *foo* contiene tre colonne (coerentemente con la *CREATE TABLE* precedente), pur non essendo state direttamente selezionate dal client durante la *SELECT*.
- L'ultima riga di output, riporta una risposta come '-1' mentre tutte le risposte del server vanno, come visto nel paragrafo 2.4, da 0 (*Ok packet*) a 255 (*Error packet*). Il plugin riserva due byte per le risposte del server in modo da contenere non solo tutti i valori da 0 a 255 ma anche altri con significato speciale. Ogni volta che esporta delle informazioni, il plugin resetta alcuni dati delle strutture private ed uno di questi è proprio la variabile che contiene la risposta del server che viene settata ad un valore sconosciuto di '-1'; questo valore quindi indica che il server non ha risposto, ed in effetti, se si guarda la figura 3, si nota che alla richiesta di fine sessione da parte del client, il server non risponde ma chiude direttamente la connessione.

[13] Queste due query sono eseguite automaticamente dal client MySQL senza che l'utente abbia ancora iniziato a fare richieste.