Unsniff Plugin Developer's Guide

For use with Unsniff Network Analyzer

Version 1.3
Feb 3, 2006

Unsniff Evaluation Copy

Version 1.3 Feb 3, 2006

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Unleash Networks Pvt Limited
5, Nehru Street, Gowrivakkam
Chennai 601302, India
http://www.unleashnetworks.com
support@unleashnetworks.com

This document and the software described by this document are Copyright © 2003-05 by Unleash Networks Pvt Ltd. All rights reserved. Use of the software described herein may only be done in accordance with the License Agreement provided with the software. This document may not be reproduced in full or partial form except for the purpose of using the software described herein in accordance with the License Agreement provided with the software. Information in this document is subject to change without notice. Companies, names and data used in the examples herein are fictitious unless otherwise noted. The API may only be used with the UNSNIFF NETWORK ANALYZER. You may not reverse engineer or copy the API to work with any other product.

Windows, Visual Studio are registered trademarks of Microsoft Corporation. All other trademarks are the property of their respective owners.

UNLEASH NETWORKS PVT LTD WILL NOT BE LIABLE FOR (A) ANY BUG, ERROR, OMISSION, DEFECT, DEFICIENCY, OR NONCONFORMITY IN MERCHANT OR THIS DOCUMENTATION; (B) IMPLIED MERCHANTIBILITY OF FITNESS FOR A PARTICULAR PURPOSE; (C) IMPLIED WARRANTY RELATING TO COURSE OF DEALING, OR USAGE OF TRADE OR ANY OTHER IMPLIED WARRANTY WHATSOEVER; (D) CLAIM OF INFRINGEMENT; (E) CLAIM IN TORT, WHETHER OR NOT ARISING IN WHOLE OR PART FROM UNLEASH NETWORKS PVT LTD's FAULT, NEGLIGENCE, STRICT LIABILITY, OR PRODUCT LIABILITY, OR (F) CLAIM FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES, OR LOSS OF DATA, REVENUE, LICENSEES GOODWILL, OR USE. IN NO CASE SHALL UNLEASH NETWORKS PVT LTD LIABILITY EXCEED THE PRICE THAT LICENSEE PAID.

Unsniff Evaluation Copy

Version 1.3 Feb 3, 2006

Revision History

June 20, 2005	Rev 1.0	Initial Release
Sep 27, 2005	Rev 1.1	Updated before public Beta release with comments from pre Beta
Oct 20, 2005	Rev 1.2	Minor updates from web reviews
Feb 3, 2006	Rev 1.3	Updates from Beta 1

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

1 Introduction

This guide explains how to write plugins for the Unsniff Network Analyzer. This guide must be used along with the API documentation and the code samples provided with the Unsniff API Developers Pack.

1.1 About Unsniff

Unsniff is the next generation network analyzer software from Unleash Networks. It features never before seen graphical representations of packets, a new storage format that can store entire sessions, PDUs, User Objects, comments and more. All conventional features of a network/protocol analyzer such as filters, statistics are also present in Unsniff in a better form.

One of the key features of Unsniff is that it is also a framework for developing your own analysis solutions. You can develop advanced decoders for your own protocols or you can choose to extend the user interface of Unsniff by adding your own views of the network data. In order to take advantage of these features, you have to write plugins.

This document along with the samples and API documentation contains all the information you need.

1.2 Intended audience

This document is intended for developers who want to:

- Add decoders for new protocols
- Add basic user interface elements to Unsniff
- Add custom sheets
- Integrate their application into Unsniff via the Eavesdrop interface
- Create custom name resolvers

1.2.1 Skills required

Unsniff Evaluation Copy

Protocol plugins are written in C++ or XML. All other plugins are written in C++.

If you wish to write protocol plugins (also known as decoders or dissectors) – you are expected to have a working knowledge of XML or C++. The choice of using C++ or XML depends on the complexity and "stateful"ness of the protocol.

The Unsniff API goes to great lengths to make development of Protocol Plugins easy. A large number of protocols can get away with *no* C++ code at all. The XML API is sufficient in those cases. Even when you have to dive into C++, the Visual Studio Wizards and the helper classes make it really easy to write protocol plugins.

Other types of plugins (user inteface, name resolvers, eavesdroppers) are written in C++. A knowledge of Microsoft ATL/COM will be helpful, but it is not required. Custom sheets are full blown ActiveX controls that directly integrate into Unsniff as another sheet.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

1.3 Getting Started

In addition to this guide; you need several other resources before you can get down to business and start writing plugins.

1. A licensed copy of ***Unsniff Network Analyzer***
    - You can purchase a licensed copy or download a trial from http://www.unleashnetworks.com
2. The ***Unsniff Plugin Developers Pack***
    - You can download the developers pack from http://www.unleashnetworks.com
    - The pack consists of the API library and header files, sample code, the Unsniff Plugin Developers Guide (this document), API reference documentation, the Unsniff Scripting Guide, and Visual Studio Wizards
    - Install the Unsniff Plugin Developers Pack
3. A copy of Microsoft Visual C++ 6.0 or higher
4. Your favorite XML Editor. We recommend the free Microsoft XML Notepad See MSDN Online XML Developer Center (http://msdn.microsoft.com/xml)

1.3.1 The Unsniff DevZone

Unleash Networks maintains the Unsniff DevZone at http://www.unleashnetworks.com .You can access the latest developer tips, articles, patches, and sample code from the DevZone. The DevZone also contains a developer message board where you can talk to Unleash Networks engineers.

The Unsniff DevZone is your most valuable resource for writing plugins for Unsniff.

Unsniff Evaluation Copy

1.3.2 Platforms

The Unsniff Plugin API works only on the following platforms:

- Windows 2000
- Windows XP

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

1.4 Typographical Conventions

This guide uses the typographical conventions shown below:

Notation	Description
`Lucida Console`	Code samples
`Ludica Italics`	Inline code comments
Times Italics	References to other documents or links
ⓘ Info	Additional advanced information
Tip	Power Tip
Warning	Warning

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

2 Unsniff Plugin Framework

A plugin is an add-on program that enhances the functionality of Unsniff Network Analyzer. Unsniff plugins are designed to integrate seamlessly into the Unsniff application framework. Unsniff plugins can be classified broadly into two categories:

1. Protocol Plugins
   a. Custom decoders for protocols

2. Advanced Plugins
   a. Using the Eavesdrop interface
   b. Custom Name Resolvers
   c. Custom User Objects and Renderers
   d. User Interface plugins (which add buttons, toolbars to Unsniff)
   e. Custom Sheets

Before we dive into a discussion of plugins – let us first talk about GUIDs and how they are used in Unsniff. These are central to the Unsniff plugin framework.

2.1 GUIDs

A GUID is a globally unique 128-bit (16 byte) number. GUIDs play a major role in the Unsniff plugin framework. Every protocol must have a unique GUID. GUIDs are also used to identify *User Object* types and *Name-Resolution* types.

2.1.1 GUID Formats

GUIDs are usually written in two formats on the Windows platform.

- "Registry" format
  `{974FB098-DE46-45db-94DA-8D64A3BBCDE5}`

- "Define GUID" format – shown below

```
DEFINE_GUID(..,0x974fb098, 0xde46, 0x45db, 0x94, 0xda, 0x8d, 0x64,
                                                 0xa3, 0xbb, 0xcd, 0xe5);
```

Tip

Every protocol in Unsniff must have a unique GUID.

Unsniff Evaluation Copy

2.1.2 Predefined GUIDs

Each protocol must have a unique GUID. Unsniff already defines GUIDs for many common protocols. See the files *USNFProtocols.h* and *USNFProtocols.c* in the include directory.

Some examples from the file:

```
DEFINE_GUID(UPID_IP,0xa2c724b, 0x5b9f, 0x4ba6, 0x9c, 0x97, 0xb0, 0x50, 0x80,
                                                     0x55, 0x85, 0x74);
DEFINE_GUID(UPID_IPV6,0x85c0cced, 0xda8d, 0x4029, 0x92, 0x4a, 0xa6, 0xab, 0x87,
                                                     0xf6, 0x2e, 0xf8);
```

For example: If you are planning to write a plugin for IP use the `UPID_IP` GUID in your plugin.

For your custom protocols or if your protocol is not in the pre-defined list, you have to generate your own GUID using a tool such as GUIDGEN.exe provided with Microsoft Visual Studio. See *Chapter 5.3* for an example of generating your own GUID.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

2.2 The Unsniff Plugin Framework

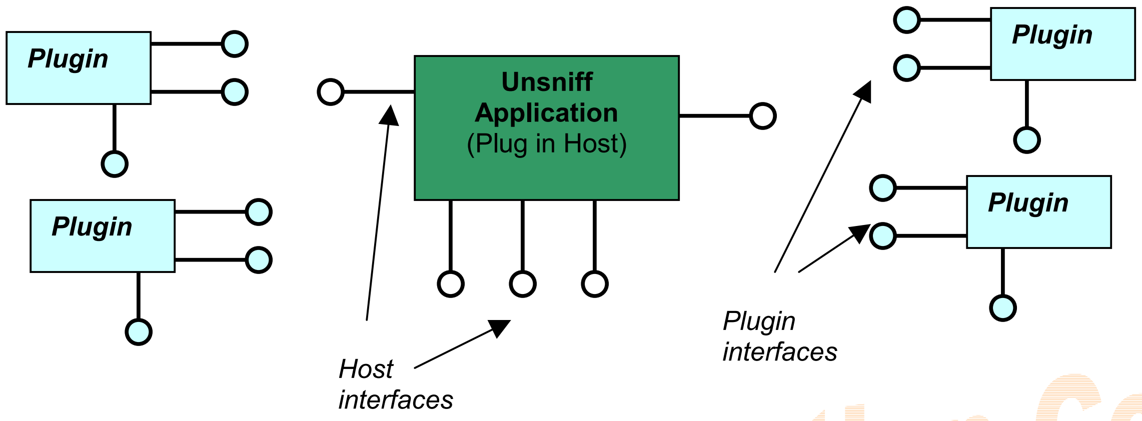
Unsniff uses Microsoft COM as its plugin framework. At the most basic level, every plugin is a COM component that implements a certain set of interfaces. These interfaces depend on the functionality being offered by the plugin.

A major part of the main Unsniff application is devoted to managing plugins. Unsniff provides the following facilities to all plugins.

- Installation and Discovery of plugins
- Configuration
- Activation and Deactivation
- Persistence support
- Reassembly
- Logging and Tracing support

2.2.1 The COM environment

The picture below shows the environment of a plugin.



The main Unsniff application is not aware of any protocols. All intelligence is distributed in various plugins. This design enables plugins to be developed and deployed independent of the main application.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

The Unsniff API and COM

The Unsniff API shields you from all of COMs gory details. You will use a wizard to generate all the skeleton code for you. You just have to write C++ code in specified areas.

Info

For **protocol plugins**, the Unsniff API provides you with a lot of support. Unless you look closely you will not even know that you are writing a COM object using the Active Template Library (ATL). So take heart !

Here are some additional considerations while writing Unsniff plugins:

- All plugins must be housed in In-Process Servers (DLL)
- All plugins must use the Apartment Threaded model
- Use the ATL (Active Template Library)

XML Plugins

The Unsniff API also features a very rich XML specification for writing protocol decoders. Most protocols can be specified using this XML schema. This requires no C++ code at all.

If your protocol is stateful and complex – you need to use C++ or a mix of XML and C++. You can take advantage of XML to define your fields and records. Field definitons are the most time consuming and error-prone tasks while developing protocol decoders. C++ can be used to handle the stateful part of your protocol. There are many samples in the API Developers Pack that illustrate how a C++ plugin can take advantage of XML to define fields and records.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

2.3 The Development Environment

In order to effectively write,debug, and deploy plugins you need to have access to some development tools. Your development environment consists of:

- Microsoft Visual Studio *(if writing a C++ plugin)*
- XML Notepad or another XML Editor
- The Unsniff Network Analyzer application

2.3.1 Microsoft Visual Studio ™

This section provides tips on leveraging Visual Studio for maximum productivity.

Tip 1 - Use the Unsniff API Wizards
All C++ plugins are COM components written using the Active Template Library. Always use the wizards to generate the plugin project as well as the skeleton of the plugin object.

Tip 2 – Build Configurations
Choose the *Unicode Debug* build for developing your plugin.This enables you to have access to Trace and Assert facilities. Switch to "*Win32 - Unicode Release Min Dependency* "build for a release version of your plugin. Both these configurations are supported via the Unsniff Plugin App Wizard.

Tip 3 – Debug Trace
Use the ***ATLTRACE(. .)*** function generously to add trace statements to your code. The output from these statements will appear in the Output window in Visual Studio. The ATLTRACE function is ignored in the release configuration.

Tip 4 – Unicode support
Use the macros defined in tchar.h to perform string operations. This will help you compile with UNICODE support with very little effort. For example : Use *_tcscpy()* instead of *strcpy*(), *_stprintf*(. . .) instead of *sprintf*.

Unsniff Evaluation Copy

Tip 5 – Class declaration
Unless you have already treaded the path of ATL COM, you are likely to get a serious headache when you look at the plugin class declaration. DO not get disheartened[1] at the apparent twisted use of C++ templates. The Unsniff API uses a clever technique to hide all these from you[2]. You are not expected to know anything about template programming or COM. A basic level of C++ knowledge is sufficient.

Tip 6 – Your own log messages
You can generate your own log messages that appear in the Unsniff Log Window. Refer to the *UserPrintLogMessage*(. . .) method in the Unsniff API Reference.

[1] The headache will disappear after you have written a few plugins. You will either learn to ignore the class declarations or learn more about ATL
[2] You plugin class actually inherits a lot of COM specific behavior from a base class. This is called implementation inheritance

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

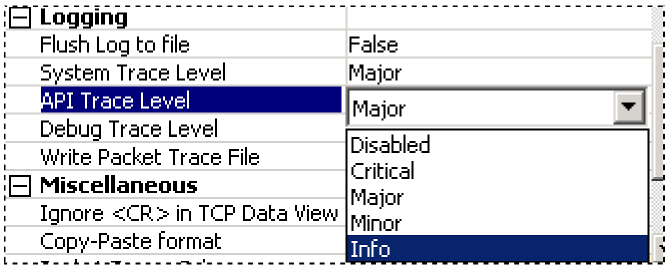
2.3.2 Unsniff Network Analyzer

This is one of your most powerful debugging aids while developing a plugin. The Unsniff API spews out very verbose and well defined error messages every time it encounters a error situation. These messages appear in the Unsniff log window.

The Log Window

If not already visible; the Unsniff Log Window must be made visible by selecting *View->Log Window* item from the main menu. All log messages from Unsniff application and the API are shown in this window in different colors.

Tip 1 – Set the API Trace Level to Info
While debugging your plugin you want to set the API trace level to INFO. This ensures that all log messages including minor warnings make it to the log window.



Logging	
Flush Log to file	False
System Trace Level	Major
API Trace Level	Major
Debug Trace Level	
Write Packet Trace File	
Miscellaneous	
Ignore <CR> in TCP Data View	
Copy-Paste format	

Tip 2 – Log Messages
The log window is a dockable / resizable window. This window contains a list of all log messages that was generated in this session. The following log message was generated because we tried to push a 38 byte field on to the field stack when only 22 bytes was left in the frame.

```
* 05-25-2005 04:55:01 00000538 01ab8ea0        [NB-NS]  [0x80040207]
UAPI BreakoutFields [ Details ]: [0x80040207] Field "Question
Records"[39] (size 38) attempted to read beyond frame. Expected Size <=
22
```

Unsniff Evaluation Copy

The format of the log message is:
[Level] [Timestamp] [Internal Use 1][Internal Use 2] [Protocol Name] [Error Code] [Error Text]

All the error codes are described in detail in *Appendix-A*.

Tip 3 – Use Import while testing
While testing protocol plugins you can either use a live capture or by just importing a capture file from another format (e.g., tcpdump). You can save some time and also get a stable test input by just importing already captured data.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

3 Protocol Plugins Overview

The most common type of plugin in Unsniff. A protocol plugin is used to decode[3] a message or protocol. Depending on the skill level of the implementer, the complexity of the protocol, and performance requirements – you can choose to write a plugin using any of the following methods.

Type	Explanation
Pure XML Plugin *(Chapter 6*)	The entire plugin is written using the Unsniff XML Specification. A large number of protocols can be handled in this method.
Pure C++ Plugin *(Chapter 5)*	Use the Unsniff API. This offers the maximum performance, features, and customizability.
A mixture of C++ and XML	Routine tasks such as field and help definitions can be written in XML. C++ can be used for the more interesting or performance critical parts

This chapter is an overview of Unsniff packet analysis concepts. If you are looking for specific information about C++ and/or XML plugins; refer to Chapter 5 (for C++) and Chapter 7(XML).

3.1 Protocol plugin tasks

A protocol plugins main task is to breakout individual fields of a protocol message from a raw byte array. There are other important tasks a plugin may need to perform. Using the Unsniff API you can write plugins that can do all these tasks.

- Handle datagram based protocols (such as UDP, BOOTP)
- Handle stream based protocols (such as LDAP, BGP)
- Control how each field is displayed in the visual breakout
- Control how each field is displayed in the tree view
- Call another protocol plugin to decode a portion of the data
- Pass control to another plugin
- Filter based on any field(s) of the protocol
- Allow any field to be shown on the *Protocol Details Sheet* in Unsniff
- Perform accounting (e.g.. Count HTTP Get/Post/Response messages)
- Provide balloon help text for any field
- Provide bit-level breakdown of bitfields
- Construct PDUs
- Extract User Objects such as files, images, video, audio from payload
- Allow user to enter configuration data for the plugin
- Enable all of Unsniff rich graphical features (such as visual breakout, TCP ladder diagrams)
- Enable scripting using Ruby/VBScript
- Decryption of data if approprite keying material is supplied
- All the above features with minimum code

Unsniff Evaluation Copy

[3] Also known as dissection

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

3.2 API error checking

One of the main headaches while writing protocol decoders is to constantly watch out for buffer overruns. Using the Unsniff API to write a protocol plugin will get you all the following error checking for free.

- Bounds checking (automatically stop if you try to read more data than captured)
- Loop checking (detects if you are in an infinite loop, for example with zero length fields)
- Alignment (detects if you try to read a mis-aligned 16,32,64, bit field)
- Invalid records (if records are not nested properly)
- Referring to an invalid field (if a message type is not known)
- Invalid ASN.1 length and structure (if an ASN.1 message is not structured properly)

In addition to the above basic error checking, Unsniff API can detect over 80 separate error conditions which are explained in *Appendix-A : API Error Codes*. When the Unsniff API detects an error it stops decoding at that point and prints a log message with a detailed error message.

3.3 Which method should I use ?

The Unsniff XML specification is really powerful. Almost any protocol can be described using XML only. Even stream based protocols such as LDAP, BGP can be expressed in XML. However, there are some restrictions to consider:

1. Your plugin must be the top-most in the protocol stack
2. You cannot use any custom configuration parameters
3. You cannot express protocols that require a look-ahead of more than 32 bits. For example: If a *Message type* field (that determines all the other fields) is in byte # 20, you have a look-ahead of 20 bytes. These protocols are extremely rare and is considered bad protocol design. If you ever run into a protocol that requires this kind of look – ahead, you have to go the C++ route.
4. You will not be able to extract user objects
5. You cannot build very advanced real time packet descriptions
6. You cannot support accounting for sub-protocols or messages
7. Your XML file can be read by the user of Unsniff. Therefore it is not suitable for proprietary or secret protocols

Unsniff Evaluation Copy

Tip

XML excels in field and help definitions. Even if you are forced to go the C++ route due to various reasons as described in Sec 3.3, you can still define your fields in XML. Then you can write code in C++ to *utilize* the fields that have been defined in XML.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

3.3.1 When only C++ will do

There are some scenarios where only a C++ plugin can be used. They are worth mentioning here.

1. You want your protocol to remain proprietary so you cannot use an XML file
2. You need direct access to the packet payload
3. Your protocol has further lower level protocols
4. You want to create PDUs and User Objects
5. You want the highest performance possible

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

3.4 Unsniff Packet Analysis Process

A protocol plugin needs to support two distinct packet analysis functions. They are *Quick Parse* and *Field Breakout*. Both these packet analysis functions have only two inputs (1) an array of BYTEs corresponding to the captured packet data (2) a USHORT (16 bit) number containing the size of the array.

3.4.1 Main packet analysis functions

The protocol plugin has to answer these questions for each of the two packet analysis functions. *In these questions, "you" refers to the plugin; "me" refers to the main Unsniff application.*

- Quick Parse
  - How many bytes of the array can you handle ?
  - Give me a short description of the packet
  - Is there a next protocol? If yes, give me values for the access points for the next protocol or tell me directly what the next protocol is
  - Perform any accounting now

- Field Breakout
  - Breakout the array into its constituent protocol fields
  - Provide a mini-description of the packet

3.4.2 Stream analysis

Many protocols are stream based. They depend on a lower transport layer such as TCP to provide a reliable transmission stream independent of the lower layer. Stream based protocols do not respect the message boundaries at the data link layer imposed due to the MTU of the physical medium.

Unsniff Evaluation Copy

Stream based protocols define PDUs or Messages which can span packet boundaries. One packet may contain many PDUs or a single PDU may span multiple packets. Examples of stream-based protocols are LDAP, BGP, and HTTP. Using the Unsniff API you no longer have to worry about reassembly or keeping track of segments.

Stream based plugins must answer the following questions (in addition to *Quick Parse* and *Field Breakout* functions mentioned in Sec 3.2.1)

- Stream Function
  - Do you want to be notified as bytes accumulate on a stream? If yes, how many bytes must accumulate before you are notified.
  - Handle one or more *"stream events"* (example events are: bytes ready in stream, stream started, stream closed, stream incomplete)

    - Construct a PDU (which will be displayed in the *PDU Sheet*)
    - Extract user objects (if you want)

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

3.5 The Field Breakout process

The primary work of a protocol plugin is to breakout a raw array of BYTEs into the constituent protocol fields. The breakout model used by Unsniff is a unique *Frame – Stack* model.

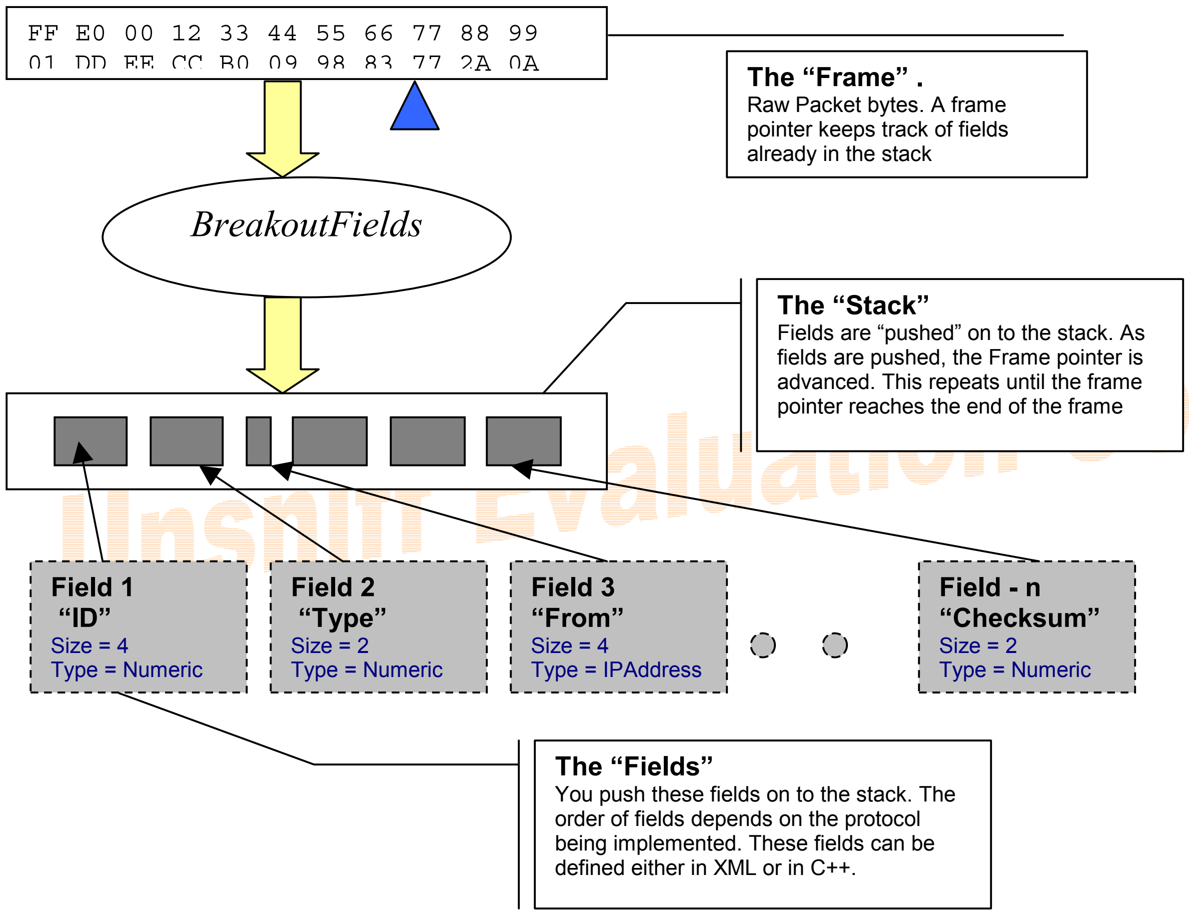
3.5.1 The Frame – Stack model

The Frame – Stack model is an innovation of Unsniff. It is designed to tremendously ease plugin development. Captured packet data is presented to a plugin as a *Frame.* Unsniff maintains a *Frame Pointer* that is automatically updated as fields are accounted for. The stack is nothing but a set of fields that represent the protocol data.

Your tasks while writing a protocol plugin are:

- Provide definitions of each field (you can use XML or C++ to provide these definitions)
- Push some or all of the defined fields on to the *stack* in the correct order

The "Frame / Stack" model is shown below.



FF E0 00 12 33 44 55 66 77 88 99
01 DD EE CC B0 09 98 83 77 2A 0A
The "Frame" .
Raw Packet bytes. A frame pointer keeps track of fields already in the stack
BreakoutFields
The "Stack"
Fields are "pushed" on to the stack. As fields are pushed, the Frame pointer is advanced. This repeats until the frame pointer reaches the end of the frame
Field 1 "ID" Size = 4 Type = Numeric
Field 2 "Type" Size = 2 Type = Numeric
Field 3 "From" Size = 4 Type = IPAddress
Field - n "Checksum" Size = 2 Type = Numeric
The "Fields"
You push these fields on to the stack. The order of fields depends on the protocol being implemented. These fields can be defined either in XML or in C++.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

3.5.2 Example

To quickly grasp the idea of the Frame-Stack breakout model, see the example shown below.

In the example below: we are trying to decode an RTP packet. (RTP stands for Real Time Protocol, the actual mechanics of the protocol are not important to us for this purpose)
Field breakout is a two step process.

1. Provide definitions of all fields used by the protocol
2. Push the fields onto the stack using the frame as a guide

3.5.2.1 Provide Field Definitions

Fields are the foundation upon which packet analysis is built on. See *Chapter 4 : Fields* for more detail on fields. Fields can be described in XML or C++. The pros and cons of both approaches are discussed in detail in Chapter 4.

```
/////////////////////////////////////////////////////////////////////////
//      ProvideFieldDefs  - Predefined field definitions
//
/////////////////////////////////////////////////////////////////////////
BOOL         CPIRTP::ProvideFieldDefs()
{

       USNF_BEGIN_ENUM_DEF(RTPProfiles)
             ENUM_ENTRY(0,   "PCMU,audio,8 khz,1 chan")
             ENUM_ENTRY(3,   "GSM,audio,8 khz,1 chan")
             ENUM_ENTRY(33,   "MP2T,aud/vid,90 khz")

             //.. similarly define all profiles here ..

             ENUM_ENTRY(34,   "H263,video,90 khz")
       USNF_END_ENUM_DEF()
```

An Enum List

```
       CUSNFFlagsField * pPTF = new CUSNFFlagsField("M/Payload Type",
                                       "M/PT",
                                       FW_8BITS,
                                       FS_SUBLAYOUT);

             pPTF->AddChild(FID_PT_F_M, new CUSNFNumericField("Marker",
                                       "M", FW_1BITS, FS_PLAIN));

             pPTF->AddChild(FID_PT_F_PT, new CUSNFEnumField("Payload Type",
                                       "PT", FW_7BITS,
                                 FS_LABEL|FS_PROTOCOL,
                                 USE_ENUM(RTPProfiles)));

       UserAddFieldDef(FID_PT_FLAGS,pPTF);
```

Flags Field

```
       UserAddFieldDef(FID_SEQ, new CUSNFNumericField( Sequence",
                                              FW_16BITS,
                                              FS_PROTOCOL));
```

A Numeric field

```
       // . . define all other fields similarly .. .
       return TRUE;
}
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

3.5.2.2 Breakout Fields

Assuming all the fields are defined as shown in the example above using C++ or XML. You are now ready to specify which fields appear in the given frame. You do this by pushing fields onto the field stack. The *FieldStm* is a C++ stream operator which can be used to push fields onto the stack. Alternately you can call *UserPushField(..)*

```
/////////////////////////////////////////////////////////////////////////
//     BreakoutFields  - Parse the Data buffer completely
//
/////////////////////////////////////////////////////////////////////////
BOOL         CPIRTP::BreakoutFields(UCHAR * Data, USHORT DataLength)
{

    FieldStm       << FID_CC_FLAGS
                   << FID_PT_FLAGS
                   << FID_SEQ
                   << FID_TS
                   << FID_SSRC
                   << FID_CSRC;
```

These fields always present

```
    // Has extension header (depends on the X bit in the CC_FLAGS field)
    pv = UserGetFieldValue(FID_CC_F_X);
    if(pv->GetNumericVal()==1) {
          FieldStm      << FID_EXT_DEF_PROFILE
                         << FID_EXT_LENGTH
                         << FID_EXT_DATA;
    }
```

These 3 only present if X bit is set in CC flags

```
    //.. more code..similar to above.. //

    return TRUE;
}
```

You can use the Frame access methods like *UserGetNextOctet ()* to write conditional code. You can also use intelligent fields such as ChoiceField and ConditionalField to accomplish the same thing. We will examine this in detail in *Chapter 4 : Fields*

Unsniff Evaluation Copy

Tip

The above example uses C++ to define fields. You could have also defined these fields in XML. You could also group the fields shown into a *CUSNFRecord* field and just push that one field onto the stack. The Unsniff API will allow you to perform the same task in different ways.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4 Fields

Fields are the building blocks of the Frame-Stack model. The richness or depth of a protocol decode depends on the accuracy of the fields defined. Using the Unsniff API you can define a protocol field in great detail – with minimum effort.

What is a field?

A field represents a sequence of bytes that mean something to a network protocol. A protocol message is nothing but a well-defined sequence of fields. Examples of fields for the IP protocol are: *"Source IP Address" , "Destination IP Address", "Type of Service"*

As a protocol plugin developer your task is to:

- Read the relevant RFCs or other documents that specify the protocol
- Make a note of all the fields that are present in the protocol
- Express each field using the Unsniff API

4.1 Goals

The primary (over 90%) of the work involved in writing a protocol decoder is defining fields. The Unleash Networks team had these goals while designing the support infrastructure for fields.

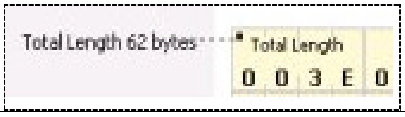
1. Unsniff Fields (*Fields*) must allow the user to specify as much detail or as little detail as they want
2. Fields must be extensible. The user can add their own field types using C++
3. Fields must also capture the visual representation of the object
4. Fields have filtering support built in
5. Fields must enable the *Protocol Details* view in Unsniff
6. All fields must automatically be scriptable via the Unsniff Scripting API
7. The user must be able to use XML or C++ to specify fields
8. Fields must automatically avail of the security and error checking features built into the Frame-Stack model
9. Fields must be self-documenting. The user should be able to provide field level help
10. Fields must handle all common networking issues like ASN.1, endian-ness, alignment, records, name resolution
11. Above all Fields must be easy to use

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.2 Properties of Fields

A field has the following properties

Property	Explanation
ID	An integer ID that uniquely identifies the field
Type	The type of field. You can use the built in types such as Numeric, IPAddress, Binary, String, etc or your own user defined types. See *Sec 4.4 : Standard Field Types* for details about Types Sec *Sec 4.5 : Defining Fields* for details on adding your own field types
Name	The full name of the field. For example: "`Destination IP Address`"
Short Name	A short form of the name. The Visual Breakout will use this name if there are space constraints while displaying the full Name. E.g. ""`Dest IP`" It is recommended that the length of the short name is not more than 10 characters
Size	The size of the field. You can use a size value of `F_AUTOSIZE` if the size is not known ahead of time. **Note:** In Unsniff all sizes are in Bits (not bytes)
Help ID	An integer ID that identifies the field level help text. Many fields can share a common help text by using the same Help ID
Styles	Determines how a field is interpreted, and displayed. See *Sec 4.3 : Styles* for details on styles
Icon	Determines what icon is shown for the field. You can select from a list of pre-defined Icons.
Value	A variant structure that holds the field value
Display Value	The string representation of the field value Example : the string "`192.168.1.1`" for a 32-bit IP address
Display Label	The string used for the field label in the Unsniff Visual Breakout. "`Total Length 62 bytes`" in the example shown below 

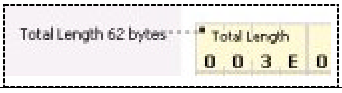
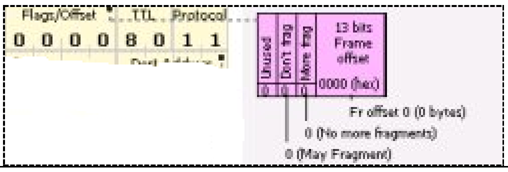
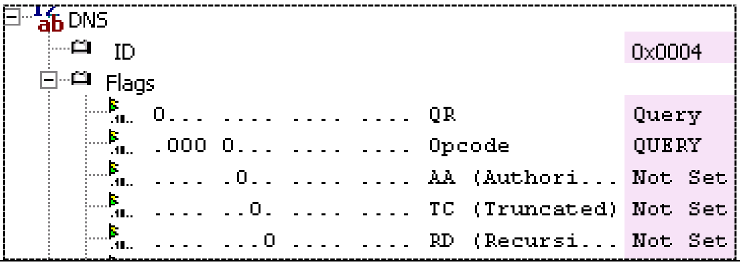
Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.3 Styles

Styles determine how a field is interpreted and displayed.

- In C++ ; combine styles by OR-ing them together (e.g. `FS_LABEL|FS_FILTER`)
- In XML : combine styles using a comma (e.g. `<styles> label,filter </styles> )`

Style (C++, XML)	Explanation
FS_PLAIN plain	The normal style. The visual breakout field is shown along with the tree view.
FS_DRAW_LABEL label	Show a label in the visual breakout 
FS_DRAW_SUBLAYOUT sublayout	For bit-fields; show the bits in a separate mini frame. The example below shows the Flags/Frame Offset field sublayout for the IP Protocol  Flags/Offset TTL Protocol 0 0 0 0 8 0 1 1 Unused Don't frag More frag 13 bits Frame offset 0000 (hex) Fr offset 0 (0 bytes) 0 (No more fragments) 0 (May Fragment)
FS_ENABLE_FILTER filter	The user can construct a display filter based on this field
FS_ENABLE_FILTER_VALUE filtervalue	The display filter must be based on the field value type. So numeric fields will allow the user to enter numeric expressions, enum will allow user to select values from a set, etc.
FS_ENABLE_FILTER_DISPLAYSTRING filterdisplaystring	The display filter must be based on the display string of the value. This allows the user to enter string regular expressions even for numeric or enum fields.
FS_DRAW_SEPARATOR	Draw a separator in the visual breakout. You can use this to delineate records
FS_NO_VISUAL nodetail	Do not draw this field in the visual breakout. Use this while displaying text based protocols such as HTTP, where the actual visual breakout has limited value – but will take up lot of space
FS_NO_DETAIL novisual	Do not draw this field in the tree view
FS_NO_VALUE novalue	This field has no value. You may use this style on `Reserved` or `Pad` fields
FS_PROTOCOL_ITEM protocolitem	This field will appear on the *Protocol Details Sheet* in Unsniff
FS_USE_HOSTORDER hostorder	This field is a numeric field in host order.
FS_ETHEREAL_STYLE_BITFLAGS showbitflags	For bit-fields; this style indicates that we want to see a traditional bit wise display in the tree view. 

```
DNS
  ID                                                  0x0004
  Flags
    0... .... .... ....  QR                           Query
    .000 0... .... ....  Opcode                       QUERY
    .... .0.. .... ....  AA (Authori...               Not Set
    .... ..0. .... ....  TC (Truncated)               Not Set
    .... ...0 .... ....  RD (Recursi...               Not Set
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

FS_UNICODE unicode	For string fields. Indicates the contents are in Unicode
FS_SAVE_TO_VARIABLE variable	Save the value of this field in a user-defined variable
FS_ALIGN_NATURAL align	Enforce natural alignment rules for numeric fields. If this style is in effect, the API will expect 16 bit fields to be aligned on a SHORT boundary and 32-bit fields on a LONG boundary. If this is not the case an error is flagged.
FS_IGNORE ignore	This field must be ignored. You can use this style to dynamically disable certain fields.
FS_REVERSE reverse	The bit fields are specified in reverse order. This style is useful when implementing IEEE standards.
FS_FILL fill	This field will auto-repeat till the end of the packet frame or the end of the current record.
FS_HEX_FORMAT hex	Display this numeric field in Hexadecimal format
FS_OPTIONAL optional	ASN.1 field only. Use this for ASN.1 fields that are tagged `OPTIONAL`
FS_HIDDEN hidden	This field will be hidden in the tree view. Use this for `Record Fields` if you do not want a separate tree node for the record
FS_COMPRESSED_VISUAL compressed-visual	The visual breakout for this field will be compressed.
FS_SIGNED signed	Treat the field as a signed entity.
FS_CONDITIONAL	This field is conditional. It will be present only if a user defined logical expression is true. (Example: `$MsgType != 25`)
FS_SIZE_EXPRESSION	The size of the field is determined by evaluating an user-defined expressions. (Example: `8 * ( $MsgLen – 2 )`)
FS_CHOICE_EXPRESSION	One of the child fields is selected based on a user-defined expression. See "`ChoiceField`" for more detail.
FS_REPS_EXPRESSION	This field is auto-repeat. The number of times this field repeats is determined by evaluating an expression (Example: `$RecordCount`)

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.4 Standard Field Types

The Unsniff API provides you with a set of standard fields. These fields are designed to take care of the majority of protocol plugin needs. If none of these fields suit you – you can define your own field by deriving from an appropriate base class using C++.

Using fields in C++

You define fields by instantiating the appropriate class. The class names are of the form ***CUSNF****<FieldType>****Field***. Eg.*CUSNF*Numeric*Field*, *CUSNF*String*Field*

An example C++ entry

```
UserAddFieldDef(FID_YIADDR, new CUSNFIPAddressField( "Yiaddr",
                                                     FS_FILTER|FS_PROTOCOL));
```

Using fields in XML

In XML you define fields by adding a <FieldDef> entry. The field names are simple English names. E.g., "numeric" "string"

An example XML entry
(Refer to Chapter 6 for more detail about the Unsniff XML Specification)

```
    <FieldDef name="Yiaddr" >
       <fieldtype>ipaddress</fieldtype>
       <styles>filter,protocolitem</styles>
       <helptext>"Your" (client) IP Address</helptext>
     </FieldDef>
```

Unsniff Evaluation Copy

Field Types

The Unsniff API features the following built-in field types:

- CUSNFNumericField
- CUSNFEnumField
- CUSNFBinaryField
- CUSNFGroupField
- CUSNFRecordField
- CUSNFFlagsField
- CUSNFStringField
- CUSNFDelimitedStringField
- CUSNFIPAddressField
- CUSNFIPv6AddressField
- CUSNFMACAddressField
- CUSNFCommentField
- CUSNFChoiceField

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

- CUSNFFloatField
- CUSNFBigNumericField
- CUSNFGUIDField
- CUSNFExternalField
- CUSNFWeakRefField
- CUSNFAsnField
- CUSNFAsnGroupField
- CUSNFAsnHeaderField
- CUSNFAsnChoiceField
- CUSNFAsnSetField
- CUSNFAsnWeakRefField
- CUSNFPadField

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5 Defining Fields

The *ProvideFieldDefs*() method is where you must all the fields used by your protocol. In the *ProvideFieldDefs*() method; you can use C++ to define your fields or load field definitions from an XML document.

This section deals with these issues in detail
.

- XML vs. C++ issues
- *UserAddFieldDef* function
- *ProvideFieldDefs* function
- simple fields
- bit fields
- records
- using variables
- fields of dynamic length
- auto-repeat fields
- choice fields
- conditional
- external fields
- ASN.1 fields
- padding fields
- using delay load
- using name resolution
- user-defined fields

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.1 XML vs. C++

You can define fields in XML as well as in C++. These following scenarios arise:

1. You are writing a pure C++ plugin
   a. In the method *ProvideFieldDefs*() , define all the fields directly using *UserAddFieldDef*()[4]

2. You are writing a pure XML plugin
   a. Define all your fields in the <*FieldDefs*> … </*FieldDefs*> block. Your protocol must have only one field as the root field. Specify the name of this root field in the <rootfield> tag.

3. You are writing a C++ plugin with only Field definitions specified in XML. This is a very powerful combination.
   a. Define all your fields in the <*FieldDef*> …. </*FieldDef*> block.
   b. In the method *ProvideFieldDefs*() , instruct the API to load fields from the XML document using *UserLoadFieldDefsFromXML("myprot.xml")* function.
   c. You can also continue to define more fields in C++ using the regular *UserAddFieldDef()* method.

4.5.2 ProvideFieldDefs function

You must define all fields within this method. This method can be used to load field definitions in both XML and C++ formats. See examples below.

C++ Field Definitions

```
BOOL CmyPlug::ProvideFieldDefs()
{
    UserAddFieldDef(FID_1, - - - );

    UserAddFieldDef(FID_2, - - - );

    // more fields

    UserAddFieldDef(FID_12, - - - );

    return TRUE;
}
```

XML Field Definitions

```
BOOL CmyPlug::ProvideFieldDefs()
{
    UserLoadFieldDefsFromXML("xmldoc');

    return TRUE;
}
```

Unsniff Evaluation Copy

More detail about *ProvideFieldDefs* can be found in *Sec 5.5 : Defining Fields*

[4] See the Unsniff API Reference Documentation

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.3 UserAddFieldDef function

This is the method that is used to add fields defined using C++. This is discussed in detail in *Chapter 5.5 : Defining Fields*

4.5.4 Simple Fields

In this section – we will look at the usage of simple fields. Simple fields are of fixed length and do not contain nested fields. Some examples of simple fields are Numeric, Enumerated, Strings, IPAddress etc.

Let us take an example of a numeric field and see how it is defined in C++ and XML. You can construct numeric fields using the following constructor.

```
CUSNFNumericField(LPCTSTR pszName,
                  LPCTSTR pszShortName,
                  DWORD   dwSizeBits,
                  DWORD   dwStyle
                  );
```

Refer to the Unsniff API Reference documentation for a definition of all the constructors available for CUSNFNumericField.

C++ Field Definitions	XML Field Definitions
`new CUSNFNumericField( "ProfileID", "PID", FW_16BITS, FS_LABEL|FS_FILTER));`	`<fielddef name="ProfileID" shortname="PID"> <fieldtype>numeric</fieldtype> <sizebits>16</sizebits> <styles>label,filter</styles> </fielddef>`

Other simple fields are:

Class Name	Description
CUSNFNumericField	A numeric field of upto 32 bits wide
CUSNFBigNumericField	A numeric field upto 64 bits wide
CUSNFStringField	An ANSI or Unicode string field. It can be of fixed length or null terminated
CUSNFDelimitedStringField	An ANSI or Unicode string field terminated by a sequence of characters (eg. "\r\n")
CUSNFBinaryField	Arbitrary length field
CUSNFFloatField	A field in 32-bit IEEE Floating Point Format
CUSNFIPAddressField	An IP Address
CUSNFMACAddressField	A MAC Address
CUSNFIPv6AddressField	An Ipv6 Address
CUSNFGUIDField	A 128-bit GUID
CUSNFCommentField	A user comment. This field can be used to insert comments into your breakout.

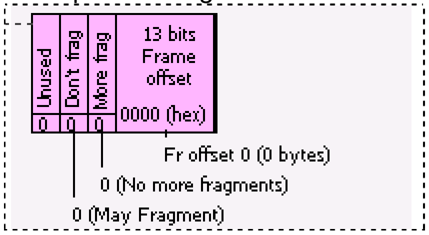
Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.5 Bit Fields

Unsniff supports bit fields of upto 32 bits wide. Bit fields are shown in a separate frame with a detailed breakout of individual bits. An example of a bit-field is shown below.

Example : IP Flags/Frame Offset Field



The C++ class *CUSNFFlagsField* is used to model bit fields. You must first define a *CUSNFFlagsField* – then add details about the individual bits within the bit-field.

Let us try to define the "IP Flags/Frame Offset" field shown above in both C++ and XML to make it clear.

C++ Field Definition

```
CUSNFFlagsField * pF;
pF = new CUSNFFlagsField("Flags/Frame Offset",
                          "Flg/FO",
                           FW_16BITS, FS_SUBLAYOUT);

    pF->AddChild(FID_UNUSED,   new CUSNFNumericField("Unused",
                                       FW_1BITS, FS_PLAIN));

    pF->AddChild(FID_DONTFRAG, new CUSNFNumericField("Don't Fragment","DF",
                                       FW_1BITS, FS_LABEL));

    pF->AddChild(FID_MOREFRAG, new CUSNFNumericField("More Fragments","M",
                                       FW_1BITS, FS_LABEL));

    pF->AddChild(FID_FROFF,    new CUSNFNumericField("Frame Offset","Off",
                                       FW_13BITS, FS_LABEL));
UserAddFieldDef(FID_IP_FLAGS,pF);
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

XML Field Definition

```
<FieldDef name="Flags/Frame Offset" shortname="Flg/Fo">
     <fieldtype>flags</fieldtype>
     <styles>sublayout</styles>

     <FieldDefs>

        <FieldDef name="Unused" >
           <fieldtype>Numeric</fieldtype>
           <sizebits>1</sizebits>
        </FieldDef>

        <FieldDef name="Don't Fragment" shortname="DF">
           <fieldtype>Numeric</fieldtype>
           <sizebits>1</sizebits>
           <styles>label</styles>
        </FieldDef>

        <FieldDef name="More Fragments" shortname="M">
           <fieldtype>Numeric</fieldtype>
           <sizebits>1</sizebits>
           <styles>label</styles>
        </FieldDef>

        <FieldDef name="Frame Offset" shortname="Off">
           <fieldtype>Numeric</fieldtype>
           <sizebits>13</sizebits>
           <styles>label</styles>
           <helptext>"Field Help for Frame Offset"</helptext>
        </FieldDef>

      </FieldDefs>
   </FieldDef>
```

Usage Notes:

1. Define the individual bit fields MSB to LSB. You can also define in the reverse order LSB to MSB, but then you must specify the FS_REVERSE field style
2. You must account for every bit in the bitfield. If there are unused bits, create a corresponding "unused" or "reserved" field
3. You cannot have any further nesting of the bit field. A bit field cannot contain records, or other bitfields
4. The bit field must be of a fixed size between 4-32 bits

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.6 Enumerations

Enumerated fields are nothing but numeric fields with a meaningful text attached to each possible value. Since this type of field is very commonplace, the Unsniff API provides special support for it.

If you are defining fields in C++, you must use the *CUSNFEnumField* class. An enum field consists of {integer value} → {name, long name} mappings. You must also use the macros *USNF_BEGIN_ENUM_DEF, USNF_END_ENUM_DEF, ENUM_ENTRY, ENUM_ENTRY_LONG* to define your enum block.

For example consider the "Option Code" field in the BOOTP protocol. An excerpt of possible values is shown below.

Option	Name	Long Name[5]
0	Pad	-not set-
1	End	End of Options
2	Time Offset	-not set-
3	Router	-not set-
4	Time Server	-not set-
5	Name Server	-not set-
-	-	-
116	DHCP Auto	DHCP Auto Configuration

C++ Field Definition

```
// Define all enums in this block
USNF_BEGIN_ENUM_DEF(OptionCodes)
   ENUM_ENTRY(0, "Pad" )
   ENUM_ENTRY_LONG(1, "End", "End of Options")
   ENUM_ENTRY(2, "Time Offset" )
   ENUM_ENTRY(3, "Router" )
   ENUM_ENTRY(4, "Time Server" )
   ENUM_ENTRY(5, "NameServer" )
   - - -
   ENUM_ENTRY_LONG(116, "DHCP Auto", "DHCP Auto Configuration" )
USNF_END_ENUM_DEF()

// Create an enum field using the block defined above
UserAddFieldDef( FID_OPTION_CODE,
                    new CUSNFEnumField("Option Code",
                                        "Opt",
                                        FW_8BITS,
                                        FS_LABEL|FS_FILTER,
                                        USE_ENUM(OptionCodes) );
```

Unsniff Evaluation Copy

[5] In most cases, you can leave the long name blank. Unsniff will re-use the short name. If you specify both long and short names, Unsniff will display either of the two based on available space.

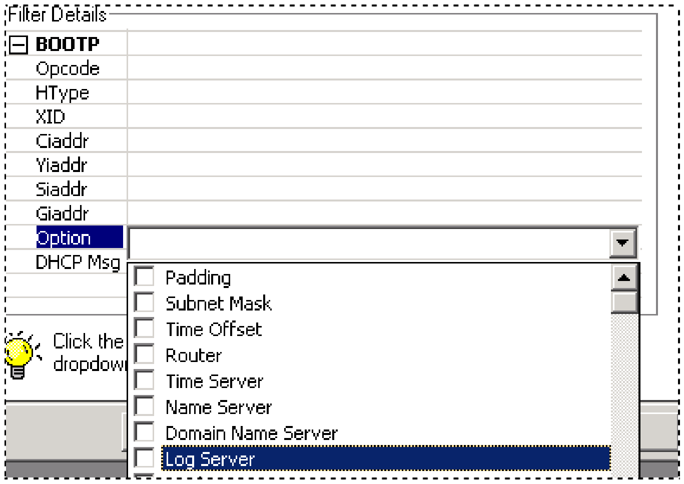
Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

XML Field Definition

```
 <FieldDef name="Option Code" shortname="Opt">
      <fieldtype>enum</fieldtype>
      <styles>filter,protocolitem,label</styles>
      <sizebits> 8 </sizebits>

      <EnumList>

         <Enum value="0" name="Pad" />
         <Enum value="1" name="End" longname="End of Options"/>
         <Enum value="2" name="Time Offset" />
         <Enum value="3" name="Router" />
         <Enum value="4" name="Time Server" />
         <Enum value="5" name="Name Server" />
         . . . . .
         <Enum value="116" name="DHCP Auto" longname="DHCP Auto Config"/>
      </EnumList>

</FieldDef>
```

Usage Notes:

Filtering
If an enumerated field also has the FS_FILTER style, then Unsniff automatically allows the user to construct a display filter using a specially designed combo box that allows multiple selections.



uation Copy

Tip
Performance Hint
We recommend that you specify your enumeration lists in a sorted ascending order. This can speed up your plugin – because the Unsniff API can search the list faster.

Tip
Binary enumerations
You can also define enumerated values for bit fields. This can greatly increase the user-friendliness of the decode. *For example: { 1= Enable, 0= Disable} { 1= Do Discovery; 0= Do not do discovery}.*

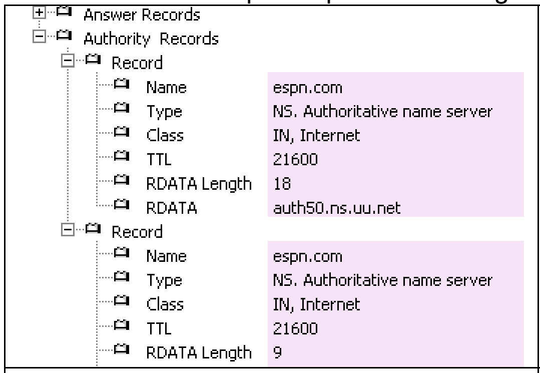
Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.7 Records

Many protocols consists of a header and a series of records. These records are nothing but a structured group of fields. It makes a lot of sense to keep the concept of records intact while displaying protocols.

- Records are displayed in the tree view as separate collapsible nodes containing the child fields
- Records are shown the visual breakout using a novel coloring scheme that makes them jump out
- Records can be nested

Examples of records:
A DNS QUERY response packet showing records



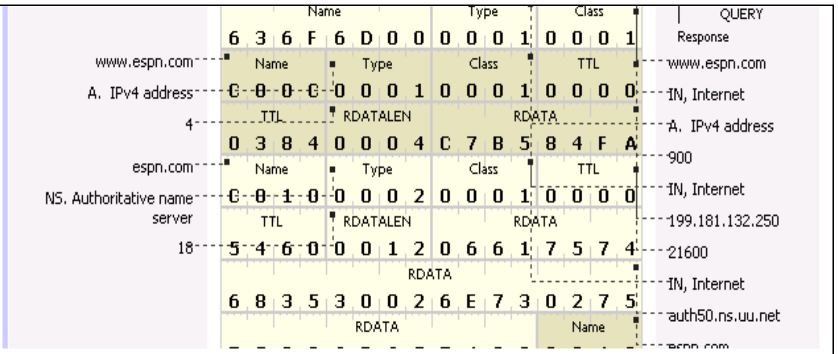
Answer Records
Authority Records
Record

Name	espn.com
Type	NS, Authoritative name server
Class	IN, Internet
TTL	21600
RDATA Length	18
RDATA	auth50.ns.uu.net

Record

Name	espn.com
Type	NS, Authoritative name server
Class	IN, Internet
TTL	21600
RDATA Length	9

Tree view of record



Visual Breakout showing record coloring in action

The C++ class *CUSNFRecordField* is used to define records. It can hold nested records, so you can add another *CUSNFRecordField* class to it as a child.

In the example below: we will create a record called *My Record* with two children *name* and *ID*.

C++ Field Definition

```
CUSNFRecordField * pRec;
pRec = new CUSNFRecordField("My Record",
                            "MyRec");

    pRec ->AddChild(FID_1,   new CUSNFStringField("Name",
                                      FW_AUTO, FS_LABEL));

    pRec ->AddChild(FID_2,   new CUSNFNumericField("ID",
                                      FW_16BITS, FS_LABEL));

UserAddFieldDef(FID_MY_RECORD, pRec);
```

Unsniff Evaluation Copy

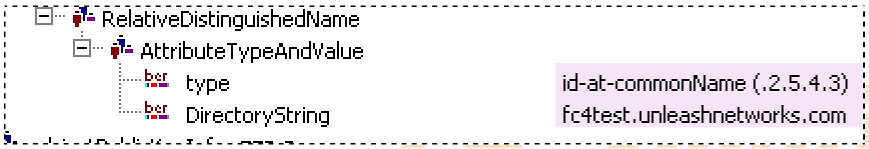
Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

XML Field Definition

```
<FieldDef name="My Record" shortname="MyRec">
     <fieldtype>record</fieldtype>

     <FieldDefs>

        <FieldDef name="Name" >
           <fieldtype>string</fieldtype>
           <styles>label</styles>
        </FieldDef>

        <FieldDef name="ID" >
           <fieldtype>Numeric</fieldtype>
           <sizebits>16</sizebits>
           <styles>label</styles>
        </FieldDef>

      </FieldDefs>
   </FieldDef>
```

Record Display Format
A record is nothing but a collection of child fields. However, records themselves may have meanings as an aggregate entity. Unsniff allows you optionally construct a string value for record fields based on the values of the child fields. This is useful when dealing with TLV style records, or ASN.1 style records. This is best illustrated by an example.

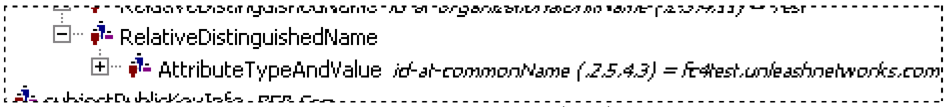
Consider the ASN.1 field from the X.509v3 Digital Certificate named *attributeTypeAndValue*, this is a record field with two children. The first child is a type and second type is a value corresponding to that type.


RelativeDistinguishedName
AttributeTypeAndValue
type id-at-commonName (.2.5.4.3)
DirectoryString fc4test.unleashnetworks.com

Record with two child fields

Unsniff Evaluation Copy

Now you may wish to assign a value to the *attributeTypeAndValue* based on the child fields. Unsniff lets you construct powerful descriptions for record fields using a display format. The display format is nothing but a string interspersed with $<child-field-number> items. In the example above, we can construct a display format `"$1 = $2"`. This will assign a clear meaning to the record field. The values of record fields are seen when the tree is collapsed in the tree view as shown below.


RelativeDistinguishedName
AttributeTypeAndValue *id-at-commonName (.2.5.4.3) = fc4test.unleashnetworks.com*

Record with display format $1=$2

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

In order to use record fields you must attach a <recorddisplayformat> tag to the XML definition of the record field. The XML fragment used to achieve the above effect is shown below.

```
- <FieldDef name="AttributeTypeAndValue">
    <fieldtype>ASNBERSequence</fieldtype>
    <recorddisplayformat>$1 = $2</recorddisplayformat>
  - <FieldDefs>
      <FieldDef name="type" ref="AttributeType" />
      <FieldDef ref="DirectoryString" />
    </FieldDefs>
  </FieldDef>
```

In C++: You can simply construct an arbitrary string and call CUSNFField::SetDisplayValString() .

Usage Notes:

There are two ways to use records to breakout fields

- You can use a *CUSNFRecordField* and add it to the field stack
  In the example above you would call:

```
FieldStm << FID_MY_RECORD; // For C++
FieldStm << "My Record";   // For XML
```

- You can use the stream record functions and push individual fields

```
FieldStm << STM_BEGIN_RECORD("My Record")
              <<  FID_1
              <<  FID_2
         << STM_END_RECORD;
```

STM_BEGIN_RECORD and STM_END_RECORD are specially defined stream operators that work with records. STM_BEGIN_RECORD starts a record – all fields pushed after that point are automatically made child fields of that record. STM_END_RECORD ends the record.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.8 Using Variables

You can use variables to control properties of other fields. Using variables you can create fields that repeat a certain number of times, specify the size of fields, create conditional fields, or select a field from a set of choices. Variables are a powerful tool.

How Variables Work

While defining fields you can also attach named variables to the value of those fields. When a field is pushed onto the field stack, the Unsniff API will check if there is a variable attached to that field. If there is a variable; then the value of the field is saved to the variable. This variable can be referred to in *later* fields. The lifetime of the variable is limited to only one packet or PDU.

Info

Variables can contain numeric or string values.

Example: In the example below, we save the value of the field "*Management Length"* in a variable called "*MgmtLength"*

C++

```
UserAddFieldDef ( FID_MG_LENGTH,
                  new CUSNFNumericField("Management Length",
                                        "Len",
                                         FW_8BITS,
                                         FS_PLAIN));

UserCreateVariable(FID_MG_LENGTH,"MgmtLength");
```

XML

```
<FieldDef name="Management Length" shortname="Len" >
   <fieldtype>Numeric</fieldtype>
   <sizebits>8</sizebits>
   <styles>label</styles>
   <variable>MgmtLength</variable>
</FieldDef>
```

Unsniff Evaluation Copy

Using Variables

Variables are typically used in *expressions*. These expressions are typically used in autosize fields, autorepeat fields, conditional fields, etc. These are explained in *Sec 6.3.7 : FieldDef* in detail.

- To refer to a variable you have to prefix the name with a $ sign. So in the above example: to access the variable *MgmtLength* – you must use *$MgmtLength*

While pushing fields on to the stack variables are automatically initialized as shown below.

```
FieldStm << FID_MG_LENGTH;
FieldStm << { .. other fields .. } ;   // insert other fields  if you want
UserGetVariableValue( "$MgmtLength" ); // Returns the integer value of Length
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.8.1 System Variables

The Unsniff API automatically gives you access to several system variables. You can use these variables for any purpose. The following table lists all the system variables.

Variable Name	Description
$SYSTEM_NEXT_BYTE	The next byte in the frame.
$SYSTEM_NEXT_WORD	The next 16 bit word in the frame
$SYSTEM_NEXT_DWORD	The next 32 bit word in the frame
$SYSTEM_FRAME_TOTAL_BYTES	The total number of bytes in the frame (ie packet or PDU)
$SYSTEM_FRAME_REMAINING_BYTES	The number of bytes yet to be accounted for in the current frame
$SYSTEM_FRAME_REMAINING_BITS	The number of bits yet to be accounted for in the current frame

Using System Variables
An example of using system variables is shown below. In this example, the field "Media Payload" is a data field that occupies the rest of the frame. Typically you would push this field after all the other fields have been accounted for.

XML

```
<FieldDef name="Media Payload">
   <fieldtype>binary</fieldtype>
   <sizeexpr>8*$SYSTEM_FRAME_REMAINING_BYTES</sizeexpr>
   <styles>compressed-visual,novalue</styles>
   ...
</FieldDef>
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

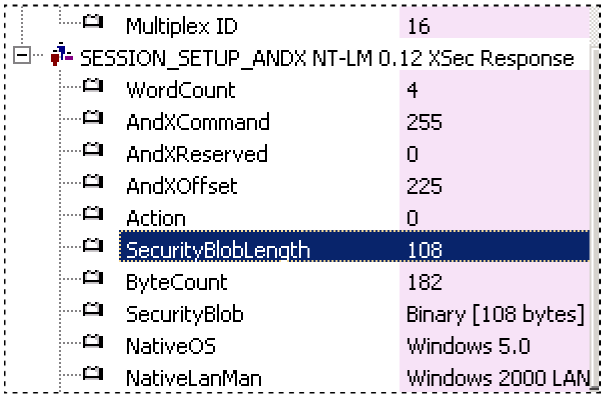
4.5.9 Variable Length Fields

This is one of the most common patterns you will encounter while designing protocol plugins. The size of a field *{Field-B}* is governed by some preceding field *{Field-A}*. The size can usually be represented as a mathematical expression. To model such fields:

- attach a variable to the "controller field" {*Field-A*}
- define the dynamic length field *{Field-B}* with a sizebits value of FW_AUTOSIZE
- set sizeexpr of {*Field-B*} to a mathematical expression using the variable attached to *{Field-A}*

Example:

Consider the following case involving the SMB Protocol. The length of the field *SecurityBlob* is determined by a preceding (not necessarily immediately) field called *SecurityBlobLength*. The value of the field *SecurityBlobLength* specifies the length in bytes of the field *SecurityBlob*.



Multiplex ID	16
SESSION_SETUP_ANDX NT-LM 0.12 XSec Response	
WordCount	4
AndXCommand	255
AndXReserved	0
AndXOffset	225
Action	0
SecurityBlobLength	108
ByteCount	182
SecurityBlob	Binary [108 bytes]
NativeOS	Windows 5.0
NativeLanMan	Windows 2000 LAN

The way to handle this using the Unsniff API is:

- Create a variable "*SecBlobLength*" and attach it to the field *SecurityBlobLength*

- Create a field *SecurityBlob* with a size of FW_AUTOSIZE

- Attach a size expression *("8 * $SecBlobLength")* to the field. Note that we have to multiply the length by 8 to yield the value in bits.

- Now you can just push the fields normally on the stack without worrying about length manipulations
```
FieldStm <<  FID_SEC_BLOB_LEN << FID_BYTE_COUNT
         <<  FID_SEC_BLOB << FID_NATIVE_OS << etc..;
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

The definition of variable length fields is shown below

C++

```
      UserAddFieldDef ( FID_SEC_BLOB_LENGTH,
                        new CUSNFNumericField("SecurityBlobLength",
                                              "SecBlobLen",
                                              FW_16BITS,
                                              FS_HOSTORDER6));

      UserCreateVariable(FID_SEC_BLOB_LENGTH,"SecBlobLength");


      UserAddFieldDef ( FID_SEC_BLOB,
                        new CUSNFNumericField("Security Blob",
                                              FW_AUTOSIZE,
                                              FS_PLAIN));

      UserSetSizeExpr(FID_SEC_BLOB," 8 * $SecBlobLength");
```

XML

```
        <FieldDef name="SecurityBlobLength" shortname="SecBlobLen" >
           <fieldtype>Numeric</fieldtype>
           <sizebits>16</sizebits>
           <styles>hostorder</styles>
           <variable>SecBlobLength</variable>
        </FieldDef>

        <FieldDef name="Security Blob" >
           <fieldtype>binary</fieldtype>
           <sizeexpr>8 * $SecBlobLength</sizeexpr>
        </FieldDef>
```

Tip

As in other cases, if you are writing a C++ plugin you can handle variable length fields without the use of variables and expressions. This will however involve writing a small amount of code[7].

```
const LPBYTE pSecBlobLength = UserGetCurrentFramePtr();
USHORT SecBlobLen = PTRVAL_USHORT(pSecBlobLength);
UserSetFieldSize(FID_SEC_BLOB, TOBITS(SecBlobLen) );
FieldStm << FID_SEC_BLOB_LEN << FID_BYTE_COUNT << FID_SEC_BLOB;
```

Unsniff Evaluation Copy

[6] We use FS_HOSTORDER because the SMB protocol encodes the 16 bit number in little endian format.

[7] Unsniff main design principle is to separate the field definitions from the process of decoding a given packet. This violates that principle. However, there may be times you want to go down to this level for performance reasons or when it is not possible to support your protocol using this technique.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.10 Auto Repeat Fields

Many protocols define a set of *fields or records* that repeat a certain number of times.

There are two variations of the above pattern:

- Number of repeats can be expressed as a function of some preceding field
- Repeat a given field until the end of frame

Repeat a field 'n' times

You may have noticed that the structure shown below is very commonplace.

```
typedef struct Message_T
{
        USHORT          RecordCount;
        struct          {
                UCHAR           MsgType;
                ULONG           TxId;
                . . .
                . . .
        } OneRecord [ RecordCount ];
} Msg_T;
```

To support the above pattern

C++

```
        UserAddFieldDef ( FID_RECORD_COUNT,
                          new CUSNFNumericField("RecordCount",
                                                FW_16BITS,
                                                FS_PLAIN));

        UserCreateVariable(FID_RECORD_COUNT,"RecCnt");


        // Create the OneRecord struction
        CUSNFRecordField * pOneRec = new CUSNFRecordField ( //. . .
                   pOneRec->AddChild(FID_MSG_TYPE, // . .
                   pOneRec->AddChild(FID_TX_ID, // . .
                   // add other children of record
        UserAddFieldDef ( FID_ONE_REC,pOneRec);


        UserSetRepsExpression(FID_ONE_REC, "$RecCnt");
```

XML

```
          <FieldDef name="RecordCount" >
             <fieldtype>Numeric</fieldtype>
             <sizebits>16</sizebits>
             <variable>RecCnt</variable>
          </FieldDef>

          <FieldDef name="OneRecord" >
             <fieldtype>Record</fieldtype>
             <repsexpr>$RecCnt</sizeexpr>
             <FieldDefs>
                <FieldDef name="MsgType" >
                   . . .
                </FieldDef>

                <FieldDef name="TxnId" >
                   . . .
                </FieldDef>
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

```
        </FieldDefs>
    </FieldDef>
```

Usage Notes

It is very easy to use the above technique to decode a packet. You just have to push the fields once onto the field stack. The auto-repeat field is magically repeated the required number of times.

```
FieldStm << FID_RECORD_COUNT

         << FID_ONE_REC   // .. will automatically repeat the
                          // field required number of times

         << // .. other fields
```

Unsniff Error Checking will ensure that junk values or incorrect use of variables do not cause any grief – so you can use this technique without worrying about overflows and other errors.

Tip

Alternate ways of handling this are always available to C++ plugins. This will however involve writing a small amount of code[8].

```
const LPBYTE pPtr = UserGetCurrentFramePtr();
USHORT RecordCount = HPTRVAL_USHORT(pPtr);
FieldStm << FID_RECORD_COUNT;
while (RecordCount--) {
        FieldStm << FID_ONE_RECORD;
}
```

Repeat a till end of frame

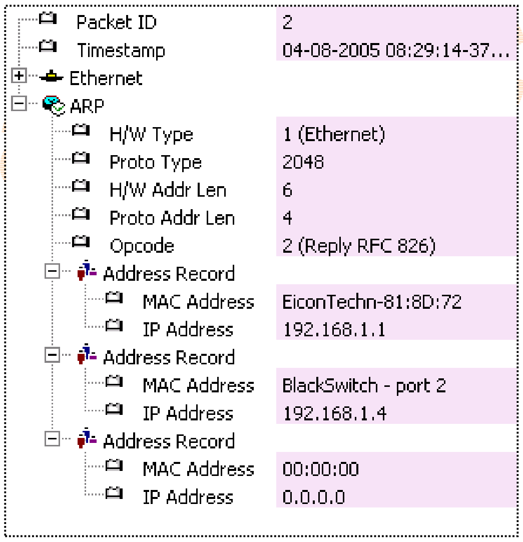
This is another common pattern. The length of the frame itself implicitly defines the number of times a field repeats.

Consider the ARP protocol (shown on right):

You can observe that the Address Record Repeats till the end of frame. The number of times this record repeats cannot be deduced by looking at any single field in the packet.

The pattern used here is:

- Repeat the field *"Address Record"* until the end of frame – in other words. Fill the remainder of the frame with *"Address Records"*



```
Packet ID              2
Timestamp              04-08-2005 08:29:14-37...
Ethernet
ARP
  H/W Type             1 (Ethernet)
  Proto Type           2048
  H/W Addr Len         6
  Proto Addr Len       4
  Opcode               2 (Reply RFC 826)
  Address Record
    MAC Address        EiconTechn-81:8D:72
    IP Address         192.168.1.1
  Address Record
    MAC Address        BlackSwitch - port 2
    IP Address         192.168.1.4
  Address Record
    MAC Address        00:00:00
    IP Address         0.0.0.0
```

Unsniff Ev... Copy

[8] Resort to this technique only if you need to – for performance reasons or for real complex scenarios. This violates the Unsniff principle of having intelligent self sufficient fields.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

The Unsniff API uses the FS_FILL style to accomplish the job. *Any field* which has the FS_FILL style is automatically repeated till the end of frame. For the above ARP example:

C++

```
CUSNFRecordField * pRec;
pRec = new CUSNFRecordField("Address Record",
                            FS_FILL);

    pRec ->AddChild(FID_MAC,   new CUSNFMacAddressField("MAC Address",
                                          FS_LABEL));

    pRec ->AddChild(FID_IP,    new CUSNFIPAddressField("IP Address",
                                          FS_LABEL));

UserAddFieldDef(FID_ADDRESS_RECORD, pRec);
```

XML

```
         <FieldDef name="Address Record" >
            <fieldtype>Record</fieldtype>
            <styles>fill</styles>

            <FieldDefs>
               <FieldDef name="MAC Address" >
                    <fieldtype> macaddress</fieldtype>
               </FieldDef>
               <FieldDef name="IP Address" >
                    <fieldtype> ipaddress</fieldtype>
               </FieldDef>
             </FieldDefs>

         </FieldDef>
```

Usage Notes
The auto-repeat field is magically repeated till the end of frame is reached

```
FieldStm <<  FID_ADDRESS_RECORD ; // will automatically repeat
                                  // till end of the frame
```

Unsniff Evaluation Copy

Tip
Alternate ways of handling this are always available to C++ plugins.

```
while ( ! UserIsBreakoutComplete()) {
        FieldStm << FID_ADDRESS_RECORD;
}
```

Note:
If a *"FS_FILL style field"* is a child field of a record. Then the field is auto-repeated to the length of the record – not to the length of the entire frame.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.11 Choice Fields

A common pattern in protocols is when a field (eg message type) is used to pick a message format from a selection of choices. We have seen how a preceding field can influence the size (*Sec 4.5.8*) and number of repetitions (*Sec 4.5.9)* of a field. Similarly, Choice Fields are used to determine the *Structure* of a message from a set of possible choices.

Consider the following example from the SMB protocol:

The field *LockType* determines whether the record *LOCKING_ANDX_RANGE* has a structure of :

- *LOCKING_ANDX_RANGE_LARGE_FILE* - if LockType = 0x10
- *LOCKING_ANDX_RANGE_SMALL_FILE* - for all other values of LockType

The class that is used to handle Choice fields in the Unsniff API is called *CUSNFChoiceField*. The way to handle this field is:

- Define a CUSNFChoiceField and attach a "choice expression" to it (both string and numeric variables are supported)
- Add all the potential choices as children of the choicefield.
- Identify each child with a choiceval. This field will become active if the choice expression matches this choiceval
- Optionally identify one field as the default choice. This field will become active if none of the choiceval-s specified for the child fields matches the choice expression result.

C++

```
UserAddFieldDef ( FID_LOCK_TYPE,
                        new CUSNFNumericField("LockType",
                                                FW_8BITS,
                                               FS_PLAIN));
UserCreateVariable(FID_LOCK_TYPE,"LockType");

pChoi = new CUSNFChoiceField("LOCKING_ANDX_RANGE");

    pChoi ->AddChild(FID_LOCKING_ANDX_RANGE_SMALL_FILE,
                   //.. add the small file record structure here;

    pChoi ->AddChild(FID_LOCKING_ANDX_RANGE_LARGE_FILE,
                   //.. add the large file record structure here;

UserAddFieldDef(FID_LOCKING_ANDX_RANGE, pChoi);

UserSetChoiceExpression(FID_LOCKING_ANDX_RANGE,"$LockType");
UserSetChoiceVal(FID_LOCKING_ANDX_RANGE_SMALL_FILE,"default");
UserSetChoiceVal(FID_LOCKING_ANDX_RANGE_LARGE_FILE,"0x10");
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

XML

```
        <FieldDef name="LOCKING_ANDX_RANGE" >
           <fieldtype>choice</fieldtype>
           <choiceexpr> $LockType </choiceexpr>

           <FieldDefs>

              <FieldDef name="LOCKING_ANDX_RANGE_SMALL_FILE" >
                   <fieldtype> record </fieldtype>
                   <choiceval> default </choiceval>
                    . . define record here . .
              </FieldDef>

              <FieldDef name="LOCKING_ANDX_RANGE_LARGE_FILE" >
                   <fieldtype> record </fieldtype>
                   <choiceval> 0x10 </choiceval>
                    . . define record here . .
              </FieldDef>

        </FieldDef>
```

Usage Notes
Once defined a choice field is really easy to use. You just push the choice field onto the field stack. Internally the correct choice is selected (or an error is generated if no suitable choice was found and if no default choice was specified)

```
FieldStm <<  FID_LOCKING_ANDX_RANGE ;  // will automatically select
                                       // the LARGE or SMALL version
                                       // based on the LockType
```

Tip
You can also specify string choices. This is useful when you are dealing with protocols that choose between several structures based on an OID.

Unsniff Evaluation Copy

The equivalent C++ way of accomplishing the same thing without choice fields is:

```
BYTE lockType = PTRVAL_UCHAR(UserGetCurrentFramePtr());
. .
switch (lockType)
{
   case 0x10: FieldStm << FID_ANDX_LOCKING_RANGE_LARGE_FILE; break;

   default:   FieldStm << FID_ANDX_LOCKING_RANGE_SMALL_FILE; break;
}
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

String Choices

Choice fields can also be used with string variables. A common usage is when dealing with ASN.1 OIDs. In the snippet below the field *extnValueChoice*, can take on several different structures depending on the ExtensionID.

- If $ExtensionID (a string variable) is ".2.5.29.15", then extnValueChoice is a KeyUsage
- For all other $ExtensionID values, extnValueChoice is an extnValue structure

```
- <FieldDef name="extnValueChoice">
     <fieldtype>Choice</fieldtype>
     <choiceexpr>$ExtensionID</choiceexpr>
  - <FieldDefs>
     - <FieldDef ref="KeyUsage">
         <choicevalstring>.2.5.29.15</choicevalstring>
       </FieldDef>
     - <FieldDef name="extnValue">
           <fieldtype>ASNBER</fieldtype>
           <choicevalstring>default</choicevalstring>
           <styles>compressed-visual</styles>
       </FieldDef>
   </FieldDefs>
```

Unsniff Evaluation Copy

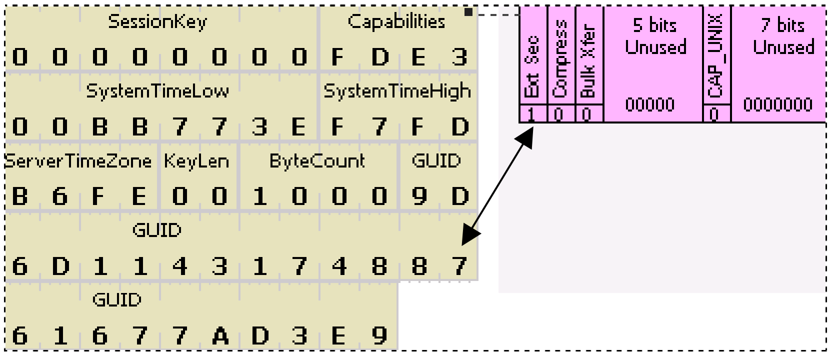
Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.12 Conditional Fields

Sometimes a preceding field determines whether or not a field is even present. A common example is for protocols requiring some kind of security block. A preceding field usually determines if this block is even present. For pure C++ plugins you can simply use an *If statement* The Unsniff API offers the Conditional field for use with XML (and C++ if you wish).

Consider the following example from the SMB protocol:

Only when the capabilities field has the Extended Security bit (Xsec bit) set – is the field GUID present. If the Ext Sec bit is 0, then the GUID field will not be present.



Conditional fields are designed to handle just this pattern. You have seen how the Size, Repetitions, and Structure of a field can be influenced by a preceding field - you can use the Conditional field to control even the presence of a field.

To handle conditional fields :

- Create a variable and attach it to the "controlling" field. You can have more than one controlling field
- Attach an expression to the conditional field using the variable defined above
- The conditional expression must evaluate to a TRUE or FALSE. It must be alogical expression

Unsniff Evaluation Copy

C++

```
// Assume the field "Ext Sec" is already defined as FID_XSEC
UserCreateVariable (FID_XSEC, "Xsec");

UserAddFieldDef ( FID_GUID,
                        new CUSNFGUIDField("GUID", FS_PLAIN);

UserSetCondExpression(FID_GUID," $Xsec==1" );
```

XML

```
<FieldDef name="GUID" >
      <fieldtype>guid</fieldtype>
      <condition> $Xsec == 1 </condition>
      <helptext> A globally unique identifier assigned to the server
      </helptext>
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Usage Notes
You can just push this field onto the stack as if it were present. If the logical expression evaluates to *false* – then the conditional field will just be ignored.

```
                FieldStm <<  FID_SERVER_TIME_ZONE
                         <<  FID_KEY_LEN
                         <<  FID_BYTE_COUNT
                         <<  FID_GUID ;     // This field will just be ignored
                                            // if the expression $Xsec == 1
                                            // evaluates to FALSE
```

The equivalent C++ way of accomplishing the same thing without using conditional fields is:

```
// Check the Ext Sec bit (Bit # 31 of the Capabilities flag)
DWORD Cap = HPTRVAL_DWORD(UserGetCurrentFramePtr());

. .
if (Cap & 0x80000000 ) {
   FieldStm << FID_GUID;
}
```

You can choose to do it either way. For C++ plugins you may want to just use an if statement as shown above instead of using a conditional field if your decode performance is unacceptable.

About Conditional expressions
Conditional expressions are logical expressions. Unsniff has rich support for parsing logical expressions. You can specify any expression using any number of variables.

Some examples:
- $Xsec == 1 (shown above)
- $Flags > 0x08000000
- $Xsec == 1 && $MyType <= 2

4.5.13 External Fields

Unsniff Evaluation Copy

Sometimes we need *"outside help"* to decode a certain block of a bytes. This is pretty common in protocols that carry security blocks inside them. In such cases, we use an *external field*. The length and structure of an external field is defined by a different plugin.

An example is the SPNEGO block within LDAP. In the case of LDAP, SPNEGO is only one of several possible mechanisms. It is not the business of the author of the LDAP protocol to also decode all possible security mechanisms. This is an excellent candidate for the external field.

In the example:
- The Protocol ID for the GSSAPI-SPNEGO protocol is `{B43274B2-BC78-4488-9339-87D5FE259340}".` This is a well known value that is published on the Unsniff website and also available in the Unsniff API.
- The plugin for GSSAPI-SPNEGO defines a top level field called "GSSAPI-SPNEGO". The name of the external field is used to match the correct field in the external plugin.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

In the Unsniff API, the class *CUSNFExternalField* models the external field.

C++

```
CUSNFExternalField * pSpnego =,
                        new CUSNFExternalField("GSSAPI_SPNEGO",
                                                FS_PLAIN);

UserAddFieldDef(FID_SPNEGO, pSpnego);
PSpNego->SetProtIDString("{B43274B2-BC78-4488-9339-87D5FE259340}");
```

XML

```
<FieldDef name="GSSAPI-SPNEGO" >
      <fieldtype> external </fieldtype>
      <protid>{B43274B2-BC78-4488-9339-87D5FE259340}</protid>
</FieldDef>
```

Usage Notes
This is a very powerful mechanism. You can just push the external field as if you had decoded it yourself. Behind the scenes, Unsniff loads the required plugin and asks it to decode the bytes from the current frame position. After the external plugin is done, control again passes to your plugin. You can then proceed to add other fields, blissfully unaware of what just happened. The code snippet to use an external field is shown below:

```
FieldStm <<  FID_SPNEGO;
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14 ASN.1

ASN.1 is by far the most important standardized structure you will see as far as network protocols are concerned. ASN.1 stands for Abstract Syntax Notation One. Many protocols such as SNMP, LDAP, GSSAPI depend on this standard.

- The current release of Unsniff only supports the Basic Encoding Rules of ASN.1

Tip

As with other fields – you can write a plugin for protocols using ASN.1 with C++ or XML. The Unsniff Protocol Plugin XML Specification is very comprehensive. You can translate almost any ASN.1 specification completely to XML. We may even develop a tool for this translation at some time in the future.

The ASN.1 field types supported by Unsniff are:

- CUSNFAsnField
- CUSNFAsnGroupField
- CUSNFAsnHeaderField
- CUSNFAsnChoiceField
- CUSNFAsnSetField
- CUSNFAsnWeakRefField

This table shows the mapping of ASN.1 types to Unsniff types

ASN.1 Type	**Unsniff Class**	**XML Tag <fieldtype>**
All Primitive types	*CUSNFAsnField*	*ASNBER*
SEQUENCE, SEQUENCE OF	*CUSNFAsnGroupField*	*ASNBERSequence*
SET, SET OF	*CUSNFAsnSetField*	*ASNBERSet*
CHOICE	*CUSNFAsnChoiceField*	*ASNBERChoice*
Header only Tag + Length	*CUSNFAsnHeaderField*	*- not available -*
Self Referential ASN types	*CUSNFAsnWeakRefField*	*Automatically done*
BIT STRING	*CUSNFAsnBitField*	*ASNBERBit*

Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14.1 UNIVERSAL Types

These are the basic types defined for ASN.1. Unsniff can handle all these types automatically, you do not have to specify which of the UNIVERSAL types a field represents[9]. Unsniff can automatically deduce it by looking at the ASN Tag present in the packet.

Example :
Consider the *AttributeType* field used in the LDAPv3 protocol. The ASN.1 for LDAPv3 defines *AttributeType* as:

```
AttributeType ::= LDAPString
```

Reading the ASN.1 specification further, we find that *LDAPString* is defined as:

```
LDAPString := OCTET STRING
```

Therefore *AttributeType* is also of type OCTET STRING. Since this is a built in ASN.1 type, Unsniff can handle it using the general purpose *CUSNFAsnField* class.

C++

```
UserAddFieldDef(FID_ATTR_TYPE, new CUSNFAsnField (“Attribute Type”
                                                 “Attr Type”,
                                                  FS_LABEL     );
```

XML

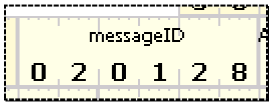
```
<FieldDef name="Attribute Type" shortname=”Attr Type” >
      <fieldtype> ASNBER </fieldtype>
      <styles> label </styles>
</FieldDef>
```

Unsniff Evaluation Copy

Usage Notes:
All ASN.1 fields automatically include the ASN Tag and Length bytes as part of the field itself.

Example: An integer ASN.1 field messageID will be decoded as shown below to include the ASN.1 tag (02) and length (01).

```
FieldStm << FID_ASN_MESSAGE_ID;
```



4.5.14.2 Enumerations

[9] This represents a huge savings for you in terms of complexity of analyzing a ASN.1 document and writing an Unsniff plugin for that protocol.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Enumerationed field types are basic types in ASN.1. You can attach an enumerated list to all ASN BER basic fields. If the actual packet frame contains a numeric or enumerated tag then the enumerated list is used, otherwise a warning is issued and the enumerated list is ignored.

Example: Version is a numeric field with enumerated values of (0,1,2).

```
<FieldDef name="Version">
    <fieldtype>ASNBER</fieldtype>
    <styles>optional</styles>
   <EnumList>
       <Enum name="v1" value="0" />
       <Enum name="v2" value="1" />
       <Enum name="v3" value="2" />
   </EnumList>
</FieldDef>
```

OID Enumerations

This is a powerful feature of the Unsniff API. You can attach enumerated names to OID fields. Since OIDs are also basic types in ASN.1 (UNIVERSAL 6); they use the same <ASNBER> fieldtype in XML and *CUSNFAsnField* in C++. OID Enumerations are very useful in cases where you want to display what each OID means as part of your protocol decode.

Example: Consider this snippet from the X.509 plugin. The AlgorithmOID determins the type of cryptographic hashing and encyprtion used by the certificate.

```
- <FieldDef name="AlgorithmOID">
    <fieldtype>ASNBER</fieldtype>
    <variable>AlgorithmOID</variable>
  - <OIDEnumList>
      <Enum oid=".1.2.840.113549.1.1.1" name="rsaEncryption"
         longname="iso(1) member-body(2) us(840)
         rsadsi(113549) pkcs(1) pkcs-1 (1) rsaEncrpytion (1)"
         />
      <Enum oid=".1.2.840.113549.1.1.2"
         name="md2WithRSAEncryption" longname="iso(1)
         member-body(2) us(840) rsadsi(113549) pkcs(1)
         pkcs-1 (1) md2WithRSAEncryption (2)" />
     - - -
    </OIDEnumList>
  </FieldDef>
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14.3 SEQUENCE and SET fields

ASN.1 structures are defined using the SEQUENCE and SET data types. These in essence define a record field

- SEQUENCE specifies an order in which member fields must appear
- SET member fields can appear in any order

Unsniff handles SEQUENCE types using *CUSNFAsnSequence* (*ASNBERSequence in XML*) and SET types using *CUSNFAsnSetField*.(*ASNBERSet in XML*)

Example:
Consider this fragment from the LDAPv3 specification (this defines the Control field)

```
Control ::= SEQUENCE {
        controlType             LDAPOID,
        criticality             BOOLEAN DEFAULT FALSE,
        controlValue            OCTET STRING OPTIONAL }
```

Observe that all the three member fields are actually built in datatypes (LDAPOID is defined as OBJECT IDENTIFIER). The only remaining issue is the DEFAULT and OPTIONAL modifiers. Both these modifiers can be handled by the optional (FS_OPTIONAL) style.

C++

```
CUSNFAsnSequenceField * pControl = new CUSNFAsnSequenceField(
                                      "Control" );

    pControl->AddChild(FID_CTL_TYPE, new CUSNFAsnField("controlType". .
    pControl->AddChild(FID_CRIT,   new CUSNFAsnField("criticality". .
                                                   FS_OPTIONAL);
    pControl->AddChild(FID_VALUE, new CUSNFAsnField("controlValue",
                                                   FS_OPTIONAL);


UserAddFieldDef(FID_CONTROL, pControl);
```

Unsniff Evaluation Copy

XML

```
<FieldDef name="Control" >
      <fieldtype> ASNBERSequence </fieldtype>
      <FieldDefs>
            <FieldDef name="controlType">
                <fieldtype> ASNBER </fieldtype>
                <helptext> OID of LDAP control type </helptext>
            </FieldDef>

            <FieldDef name="criticality">
                <fieldtype> ASNBER </fieldtype>
                <styles> optional </optional>
            </FieldDef>

            <FieldDef name="controlValue">
                <fieldtype> ASNBER </fieldtype>
                <styles> optional </optional>
            </FieldDef>

</FieldDef>
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14.4 SEQUENCE OF and SET OF

The ASN type SEQUENCE OF defines a autorepeat structure. The outer ASN Length field implicitly defines the number of times the field repeats. Unsniff handles this automatically – you do not need to keep track of the number of repeats. Like always; you can define your fields without worrying about overshooting the frame or encountering junk types. Unsniff does all the error checking for you.

You handle the SEQUENCE OF and SET OF fields using the fill (FS_FILL) styles. Attach this style to the field that will repeat. Unsniff will automatically repeat it the correct number of times.

Example:
Again let us consider the following fragment from the LDAP protocol. The Controls field is defined as a SEQUENCE of Control. Yes, that is the same Control field that we defined in the previous section.

```
Controls ::= SEQUENCE OF Control
```

In the example below: we assume that the field Control has already been defined. Notice the use of the <ref> attribute in the XML. You can find more detail about <ref> attributes in Section 7.

C++

```
CUSNFAsnSequenceField * pControls = new CUSNFAsnSequenceField(
                                      "Controls" );

    pControl->FillChild(FID_CTL, pControl);
```

XML

```
<FieldDef name="Controls" >
      <fieldtype> ASNBERSequence </fieldtype>
      <FieldDefs>
            <FieldDef ref="Control">
                 <styles> fill</optional>
            </FieldDef>
</FieldDef>
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14.5 Tagging

There are two types of tagging from the point of view of the plugin developer.

1. Implicit Tags
2. Explicit Tags

Implicit and Explicit Tagging

The tagging type in effect is usually mentioned at the top of the ASN.1 specification. For example the LDAPv3 specification uses implicit tags.

```
Lightweight-Directory-Access-Protocol-V3 DEFINITIONS
        IMPLICIT TAGS ::=
```

You may also find the tagging type specified inline. For example:

```
InitialContextToken ::= IMPLICIT SEQUENCE {
                  ..

       }
```

Example:

Consider the following fragment from the LDAP protocol. This sequence uses implicit tagging, because the entire LDAP protocol uses it.

```
        MatchingRuleAssertion ::= SEQUENCE {
                matchingRule    [1] MatchingRuleId OPTIONAL,
                type            [2] AttributeDescription OPTIONAL,
                matchValue      [3] AssertionValue,
                dnAttributes    [4] BOOLEAN DEFAULT FALSE }
```

C++

```
CUSNFAsnSequenceField * pMatching = new CUSNFAsnSequenceField(
                                 "MatchingRuleAssertion" );

    pControl->AddImplicitTaggedChild( 1,
                              new CUSNFAsnField("matchingRule". .

    pControl->AddImplicitTaggedChild( 2,
                              new CUSNFAsnField("type". .

    pControl->AddImplicitTaggedChild( 3,
                              new CUSNFAsnField("matchValue". .

    pControl->AddImplicitTaggedChild( 4,
                              new CUSNFAsnField("dnAttributes". .


UserAddFieldDef(FID_MATCHING_RULE_ASSERTION, pMatching);
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

XML

```
<FieldDef name="matchingRuleAssertion" >
      <fieldtype> ASNBERSequence </fieldtype>
      <FieldDefs>

            <FieldDef name="matchingRule">
                <fieldtype> ASNBER </fieldtype>
                <asntagimplicit> 1</asntagimplicit>
                <helptext> The matching rule</helptext>
            </FieldDef>

            <FieldDef name="type">
                <fieldtype> ASNBER </fieldtype>
                <asntagimplicit> 2</asntagimplicit>
                <helptext> The matching rule</helptext>
            </FieldDef>

            <FieldDef name="matchValue">
                <fieldtype> ASNBER </fieldtype>
                <asntagimplicit> 3</asntagimplicit>
                <helptext> The matching rule</helptext>
            </FieldDef>

            <FieldDef name="dnAttributes">
                <fieldtype> ASNBER </fieldtype>
                <asntagimplicit> 4</asntagimplicit>
                <helptext> The matching rule</helptext>
            </FieldDef>


</FieldDef>
```

Note: Explicit Tagging can be handled in exactly the same way as shown above:

- Use *AddExplicitTaggedChild*(<Tag Value> , pField) for C++ field definitions
- Use `<asntagexplicit>` *`value`* `</asntagexplicit>` for XML field definitions

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14.6 User Defined Types

These are user-defined types. Every application can define a set of custom data types that are used within that application. These are assigned a tag such as APPLICATION 10, UNIVERSAL 20 etc.

The purpose of the APPLICATION class is mentioned in the ASN.1 standard [X.409] as shown in the box below.

The standard [X.409] indicated that the APPLICATION class would be used to define a data type that finds wide, scattered use within a particular application and that must be distinguishable (by means of its[abstract syntax]) from all other data types used in the application".

The bottom line idea was to use this class to tag types that would be referenced several times in a specific application. As the tag of class APPLICATION.

Consider the following example from LDAP again for the AddRequest message type.

```
AddRequest ::= [APPLICATION 8] SEQUENCE {
        entry           LDAPDN,
        attributes      AttributeList }
```

Note that the SEQUENCE has a user-defined tag of APPLICATION 8. Using the Unsniff API, we must be able to connect the AddRequest data type to the tag APPLICATION 8.

The Unsniff API supports custom data types in the APPLICATION and UNIVERSAL class via the *AddCustomTag*() method in C++ or the **<asntag>** XML tag.

C++

```
CUSNFAsnSequenceField * pAddRequest = new CUSNFAsnSequenceField(
                                   "AddRequest" );

    pAddRequest->SetCustomTag( ASN1_APPLICATION | 8 );
```

XML

```
<FieldDef name="AddRequest" >
      <fieldtype> ASNBERSequence </fieldtype>
      <asntag> APPLICATION 8 </asntag>
      <FieldDefs>
            <FieldDef ref="entry">
                 . . .
            </FieldDef>
</FieldDef>
```

Unsniff Evaluation Copy

Usage Notes:
Note that in the XML <asntag> element, you can type in the string "APPLICATION 8". You can also use "UNIVERSAL xx" format.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14.7 CHOICE fields

ASN Choice fields are very common. A choice field is used to select a particular field from a set of candidates. The ASN tag of the field identifies the field present.

Consider the following fragment:

```
substrings      SEQUENCE OF CHOICE {
        initial [0] LDAPString,
        any     [1] LDAPString,
        final   [2] LDAPString } }
```

Recall the all tags are IMPLICIT according to the LDAP ASN.1 specification. In the above fragment, substrings is a SEQUENCE of CHOICE field. We have already seen how to support the SEQUENCE OF modifier using FillChild and fill style.

C++

```
CUSNFAsnChoiceField * pChoices = new CUSNFAsnChoiceField(
                                        "choices" );

    pControl-> AddChoiceImplicit ( 0,
                          new CUSNFAsnField("initial". .

    pControl-> AddChoiceImplicit ( 1,
                          new CUSNFAsnField("any". .

    pControl-> AddChoiceImplicit ( 2,
                          new CUSNFAsnField("final". .

CUSNFASNSequenceField * pSubstrings = new CUSNFAsnSequenceField(
                                              "substrings" );

pSubstrings->FillChild(FID_CHOICES, pChoices);
UserAddFieldDef(FID_SUBSTRINGS, pSubstrings);
```

XML

```
<FieldDef name="substrings" >
      <fieldtype> ASNBERSequence </fieldtype>
      <styles> fill </styles>
      <FieldDefs>

            <FieldDef name="choices">
                  <fieldtype> ASNBERChoice </fieldtype>
                  <FieldDefs>
                        <FieldDef name="initial">
                           <fieldtype> ASNBER </fieldtype>
                            <asntagimplicit> 0 </asntagimplicit>
                         </FieldDef>

                        <FieldDef name="any">
                           <fieldtype> ASNBER </fieldtype>
                            <asntagimplicit> 1 </asntagimplicit>
                         </FieldDef>

                        <FieldDef name="final">
                           <fieldtype> ASNBER </fieldtype>
                            <asntagimplicit> 2 </asntagimplicit>
                         </FieldDef>
                  </FieldDefs>
             </FieldDef>
. . .
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14.8 Self Referential ASN fields

Self referential fields are those which have themselves as one of their children. This one is quite nasty, but it occurs in many protocols. The Unsniff API goes to great lengths to make it almost painless for you to support this type of field.

Take this example (*for simplicity let us leave ignore the SET OF modifier*)

```
Filter ::= CHOICE {
        and             [0] ~~SET OF~~ Filter,
        or              [1] ~~SET OF~~ Filter,
        not             [2] Filter,
        . . .
```

You can see that the definition of Filter itself needs Filter to be defined. This is the catch –22 ness of the situation. The C++ version requires the use of a *Weak Reference* field.

C++

```
CUSNFAsnChoiceField * pFilter = new CUSNFAsnChoiceField(
                                      "Filter" );

    pControl-> AddChoiceImplicit ( 0,
                          new CUSNFAsnWeakRefField("and", pFilter) );

    pControl-> AddChoiceImplicit ( 1,
                          new CUSNFAsnWeakRefField("or", pFilter) );

    pControl-> AddChoiceImplicit ( 2,
                          new CUSNFAsnWeakRefField("not", pFilter) );


UserAddFieldDef(FID_FILTER, pFilter);
```

XML

```
<FieldDef name="Filter" >
      <fieldtype> ASNBERChoice </fieldtype>
      <FieldDefs>

         <FieldDef name="and" ref="Filter">
            <asntagimplicit> 0 </asntagimplicit>
         </FieldDef>

         <FieldDef name="or" ref="Filter">
            <asntagimplicit> 1 </asntagimplicit>
         </FieldDef>

         <FieldDef name="not" ref="Filter">
            <asntagimplicit> 2 </asntagimplicit>
         </FieldDef>

. . .
```

Unsniff Evaluation Copy

Note that the XML version requires no work at all !. The **<ref>** attribute can handle circular references as well.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.14.9 BIT STRING fields

A BIT STRING field is a basic field type in ASN.1. It is however different from the other fields because it assigns values to individual subfields. The ASNBERBit XML tag and CUSNFASNBitField C++ class supports the bit fields.

Example: Consider the KeyUsage bit field from the X.509 specification

```
- <FieldDef name="KeyUsage">
    <fieldtype>ASNBERBit</fieldtype>
    <styles>sublayout</styles>
  - <BitList>
      <bit name="digitalSignature" />
      <bit name="nonRepudiation" />
      <bit name="keyEncipherment" />
      <bit name="dataEncipherment" />
      <bit name="keyAgreement" />
      <bit name="keyCertSign" />
      <bit name="cRLSign" />
      <bit name="encipherOnly" />
      <bit name="decipherOnly" />
    </BitList>
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.15 Padding fields

A large number of protocols utilize padding to align fields to a pre-defined boundary. The Unsniff API provides the pad field (*CUSNFPadField*) to address almost all padding situations.

To use a padding field :

- Define a pad field with the appropriate *padding behavior*
- Push the pad field on to the field stack along with the other fields
- Unsniff will automatically add the padding field of the correct size
- If no padding is required for a given packet. The pad field will become a no-op, no pad will be added

Padding behavior

The pad field supports the following behaviors. You must select the most appropriate behavior for your padding. This information can be obtained from the relevant standards document(s) for your protocol.

Behavior	Parameters	Description

PAD_CURRENT	*B = Boundary Size in bits*	Pad to the current 'B' bit boundary.
		
		Example: PAD_CURRENT 32 bit
		(pad to the current 32 bit boundary)
		
		`11 22 00 00   00 00 11 22`
		`     |-----|`
		`        ^`
		`        |`
		`        +--- Pad field (size 2 bytes)`

PAD_NEXT	*B = Boundary Size in bits*	Pad to the next 'B' bit boundary.
		
		Example: PAD_NEXT 32 bit
		(pad to the next 32 bit boundary)
		
		`11 22 00 00   00 00 11 22`
		`     |-------------------|`
		`              ^`
		`              |`
		`              +--- Pad field`
		`                  (size 6 bytes)`

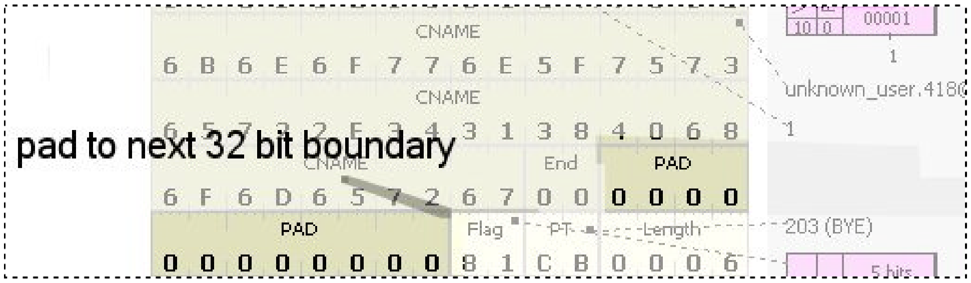
PAD_FILL	*P = Pad Byte*	Pad is indefinite length. Pad will end when the first non-pad byte is encountered or end of frame.
		
		Example: PAD_FILL 0
		(pad with zeros)
		
		`11 22 00 00   00 00 11 22`
		`     |-------------|`
		`              ^`
		`              |`
		`              +--- Pad field`
		`                  (size 4 bytes)`

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Example

RTCP protocol SDES[10] record are used to convey source description information. The end of each record is indicated by a null item type octet. No length octet follows the null item type octet, *but additional null octets MUST be included if needed to pad until the next 32-bit boundary.*



CNAME
6 B 6 E 6 F 7 7 6 E 5 F 7 5 7 3
CNAME
6 5 7 2 2 E 3 4 3 1 3 8 4 0 6 8
pad to next 32 bit boundary
CNAME
End
PAD
6 F 6 D 6 5 7 2 6 7 0 0 0 0 0 0
PAD
Flag
PT
Length
0 0 0 0 0 0 0 0 8 1 C B 0 0 0 6
00001
unknown_user.418
203 (BYE)
5 bits

As shown in the figure the PAD field follows CNAME and END and is 6 bytes long.

To support the above pattern

C++

```
// in ProvideFieldDefs()
// define the pad field

        UserAddFieldDef ( FID_PAD,
                         new CUSNFPadField(CUSNFPadField:PAD_NEXT
                                            32));

// later in BreakoutFields()
// push the pad after cname and end

FieldStm << FID_CNAME << FID_END << FID_PAD;
```

XML

In XML the pad field is defined as part of a record just like any other field. Note the use of pad specific elements (padtype, padalign,padbyte>

```
        <FieldDef name="SDES Chunk" >

                .. ..
                <FieldDef name="Pad" >
                        <fieldtype>pad</fieldtype>
                        <padalign>32</padalign>
                        <padtype> padnext </padtype>
                </FieldDef>

                .. ..
```

Unsniff Evaluation Copy

Tip

The pad field helps you write error free code. It has a lot of built in error checks for buffer overruns. The PAD next and PAD_CURRENT styles can be confusing, you may have to observe some actual traffic to ensure you have the correct type of padding behavior

[10] See RFC 3550 – SDES RTCP Packet

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.16 Using Delay Load

Delay Load is a powerful mechanism designed to improve performance and memory usage. The basic idea is to defer loading of some fields until we actually need them.

Info
Use Delay Loading sparingly. While delay load may improve performance it may hamper readability of your protocol plugin.
Delay Load is not available for pure XML plugins.

To use delay load:

- Group your fields into DelayLoadGroups
- Assign a name to each DelayLoadGroup
- When you need to use a field from a DelayLoadGroup; make a call to *UserLoadDelayLoadGroup("groupname")* – following this call all the fields in that group will become available to you.
- Proceed with pushing fields from the delay load group normally

As an example consider the 802.11 protocol. We may want to define a delay load group for management frames. This will speed up the field definition process because the field definitions for management frames will not be loaded until there is actually a management frame.

C++
Any field definitions between BeginDelayLoad and EndDelayLoad are automatically made part of the delay load group

```
//
// Mgmt Frames (following fields are used by management frames only)
//
UserBeginDelayLoadGroup("Management Frames");
UserAddFieldDef(FID_MG_AUTH_ALGO, new CUSNFNumericField(
                                "Auth Algorithm Number", . . .);
UserAddFieldDef(FID_MG_AUTH_TRANS,new CUSNFNumericField(
                               "Auth Transaction Sequence", . . .);
UserAddFieldDef(FID_MG_BEACON_INTERVAL,new CUSNFNumericField(
                               "Beacon Interval", . . .);

// define all other fields used by management frames here

UserEndDelayLoadGroup();
```

XML

```
<FieldDefs DelayLoadGroup="Management Frames" >
    <FieldDef name="Auth Algorithm Number">
       . . .
    </FieldDef>
    <FieldDef name="Auth Transaction Sequence">
       . . .
    </FieldDef>
    <FieldDef name="Beacon Interval">
       . . .
    </FieldDef>
</FieldDefs>
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Usage Notes
To use fields from a delay load group – you must explicitly load the group. There are two ways of loading a delay load group:

Using the stream operator *STM_USE_DELAY_LOAD_GROUP*

```
FieldStm <<  STM_USE_DELAY_LOAD_GROUP("Management Frames")
         <<  FID_MG_AUTH_ALGO
         <<  FID_MG_AUTH_TRANS
         <<  FID_MG_AUTH_BEACON_INTERVAL;
```

Using *UserLoadDelayLoadGroup*

```
UserLoadDelayLoadGroup("Management Frames");

FieldStm <<  FID_MG_AUTH_ALGO
         <<  FID_MG_AUTH_TRANS
         <<  FID_MG_AUTH_BEACON_INTERVAL;
```

4.5.17 Using Resolvers

A key component of any network analyzer is name resolution. The Unsniff plugin API provides in depth support for name resolution using plugin name resolvers. At the most basic level, a name resolver is something that converts an object (such as addresses, OIDs, or numbers) into human friendly strings.

Name Resolver Plugin

Name resolution is a highly customized task. There can be no one-size-fits-all solution to all types of objects. Therefore Unsniff defines something called a *Name Resolution Target.* This target is something that requires a certain type of name resolution. Each name resolution target has a unique GUID.

Unsniff Evaluation Copy

Example targets:

▪ SNMP OID	{21DDAF85-FCA6-4e42-B593-D221A3633A6E}
▪ IP ADDRESS	{92B15480-5123-4031-97B4-F8A24C3EF5C6}
▪ IP6	{AA7F6546-4123-4deb-A9A8-9029E7873DA7}
▪ MAC (OUI)	{3421E7F6-388F-44d9-849C-DD8E65BFAB10}
▪ OSPF AREA NAME	{A0908595-085E-4cc6-8D19-F6363BB4AB73}

You can also define your own name resolution target (perhaps you want to convert telephone numbers to names; interface numbers to shelf/slot/port, BGP AS numbers to something readable, etc).

To define a custom name resolver, you have to write a different type of plugin. This is called a name resolver plugin. Please refer to the *Chapter 7 : Advanced Plugins* for details.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

In this section we will discuss how to use name resolution in your fields.

Consider an example: We want to resolve an instance of a SNMP OID to a human readable name.

Note that we just use the "Target GUID". We do not care how Unsniff translates the OID to a name. Behind the scenes, Unsniff figures out if there is plugin who claims to be capable of handling the given "Target ID". If there is such a plugin, Unsniff will load it and ask it to perform the task.

C++

OID is defined as an ASN field

```
UserAddFieldDef(FID_OID,new CUSNFAsnField(_T("OID"),FS_LABEL));



// URSV_SNMPOID is the target OID
// defined in UsnfProtocols.c  to be {21DDAF85-FCA6-4e42-B593-D221A3633A6E}

UserAttachResolver(FID_OID,URSV_SNMPOID);
```

XML

```
<FieldDef name="OID">
    <fieldtype> ASNBER </fieldtype>

    <resolveid>
          {21DDAF85-FCA6-4e42-B593-D221A3633A6E}
    </resolveid>

    <styles> label </styles>
</FieldDefs>
```

Unsniff Evaluation Copy

Usage Notes
To use name resolution, simply push the field onto the field stack normally. Name resolution happens automatically.

```
FieldStm <<  FID_OID
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

4.5.18 User Defined Fields

The built in field types you have seen so far can handle the overwhelming majority of protocols. Even really weird fields can be handled with the binary or string field, albeit at a basic level. The Unsniff API however is all about extensibility, you can extend *ANY* of the built in base classes to add your own functionality. You can then register it with the API and use it from an XML plugin.

Select a base class
First of all you have to select a base class that is closest in behavior to your requirements.
If you do not know of any class that is close to your desired behavior follow these suggestions:

1. If your field has a fixed length, derive from CUSNFBinaryField. Override the *GetDisplayValString()* function to craft your own representation of the field

2. If your fields length is dynamic, derive from CUSNFBinaryField directly. Override the *SetValueAutomatic()* as well as the *GetDisplayValString()* function

Implement your class

Create a new class (example *CMyCustomField*) with your desired constructors and methods. A minimum skeleton of a class is shown below. Note that the copy constructor and the MkCopy method are mandatory. In the example we have derived from *CUSNFBinaryField*.

```
// CMyCustomField
//      My custom field for use in this protocol
//
class  CMyCustomField :
       public CUSNFBinaryField
{
public:

       // Full constructor w/o value
       CMyCustomField  (LPCTSTR    pszName,
                        LPCTSTR    pszShortName,
                        DWORD      dwStyle);

       // .. Other constructors if needed


       // Copy constructor (this is a MUST)
       CMyCustomField (const CMyCustomField & CopyFrom);


public:
       virtual LPCTSTR                 GetDisplayValString();
       virtual CUSNFField *            MkCopy(CUSNFField * pOrig);

};
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Use your class

For C++ : Your newly defined class is a first class member. It is just as good as the built-in field types supported by the Unsniff API. You can use your class just like the built in fields

For XML: To use your custom field from XML you have to do a little bit more.

1. You have to register your field type with the Unsniff API using the AddCustomFieldType(Typename, Instance) method.

2. From XML you can refer to custom field types by appending a % symbol in from of the typename

C++

```
.
// Define your custom field

UserAddFieldDef(FID_MYGREAT_FIELD,
                    new CMyCustomField("My Scan Data ",
                                       "MyScan",
                                       FS_LABEL));



// Use your custom field just like any other field

FieldStm << FID_TYPE << FID_MYGREAT_FIELD << FID_LENGTH;
```

XML

```
// In your C++ module

// Define your custom field
UserAddFieldDef(FID_MYGREAT_FIELD,
                    new CMyCustomField("My Scan Data ",
                                       "MyScan",
                                       FS_LABEL));

// Register your type
UserAddCustomFieldDef("MyCustomField, new
                                  CMyCustomField(NULL,NULL,FS_PLAIN));



// In your XML file (note the ' % ' symbol)

<FieldDef name="My Scan Data" shortname ="MyScan" >

    <fieldtype> %MyCustomField </fieldtype>
    <styles> label </styles>
    ..

</FieldDefs>
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5 *Plugins in C++*

This section contains detailed information about how to write protocol plugins in C++. This section is divided into:

- Using the Unsniff API Visual Studio wizards
- Installing plugins
- "Hello World" Step-by-Step
- Handling stream based protocols
- Defining fields
- Breakout a packet into its constituent fields
- Accounting
- Configuration support
- Tips and Tricks

Checklist

Before you jump in and start writing a protocol plugin, here is a quick checklist of things you will need.

- ✓ Install the last version of the Unsniff Plugin Developers API
- ✓ Microsoft Visual Studio 6.0 or Microsoft Visual Studio .NET
- ✓ Your protocol specification (RFC or other specifications document)
- ✓ Some sample packets containing the protocol to test your plugin
- ✓ If you are planning to use XML to define fields, have an XML Editor handy. We recommend the free Microsoft XML Notepad

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.1 The Unsniff Plugin API Visual Studio Wizards

As mentioned in Chapter 2, all Unsniff plugins are COM components. These are written using the Active Template Library (ATL). These COM components are In-process servers, which means that they will be packaged as a DLL. A single DLL can contain any number of plugins.

A typical arrangement for the plugin developer is:

1. Create a single DLL for your entire family of protocols
2. Create separate plugins for each protocol inside this DLL

5.1.1 Wizards Introduction

The most frustrating thing about writing COM objects using ATL is just getting the framework right. Unsniff makes it really easy to get started with the help of two wizards.

1. Plugin Project AppWizard
   Used to create the DLL project, which will house all your plugins

2. Plugin ATL Object Wizard
   Used to create a single protocol plugin

These wizards are automatically installed into your Microsoft Visual Studio environment when you install the plugin API. These wizards will be completely removed from the system when you uninstall the Unsniff Plugin API.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.1.2 Unsniff Project AppWizard

Purpose: This wizard will create a Visual Studio project for you. This project will build a DLL file . All your plugins will be created within this DLL

Step by Step

1. Open Microsoft Visual Studio

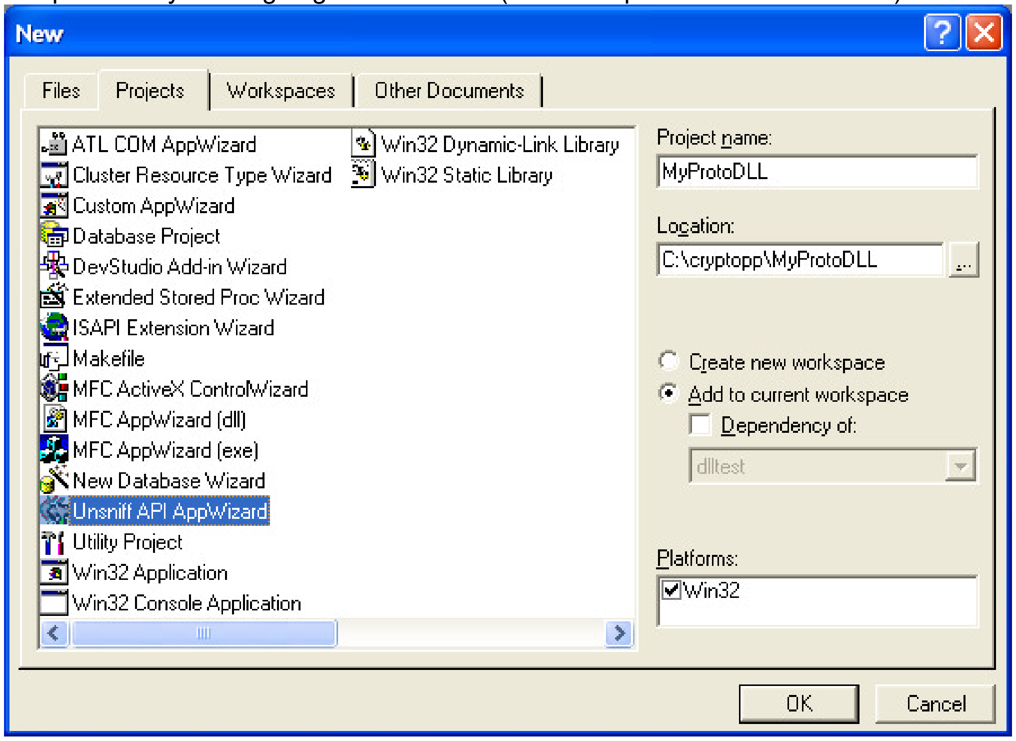
2. Click on "*File*" -> "*New*"

h

3. Switch to the *Projects* Tab. You will now be presented with the screen below

4. Select the "*Unsniff API AppWizard*" project type from the list (as shown below)

5. Type a name for your DLL. We suggest that you use a DLL name that represents the protocols you are going to house in it. (For Example: *AcmeProtocols.dll*)



Project name: MyProtoDLL

Location: C:\cryptopp\MyProtoDLL

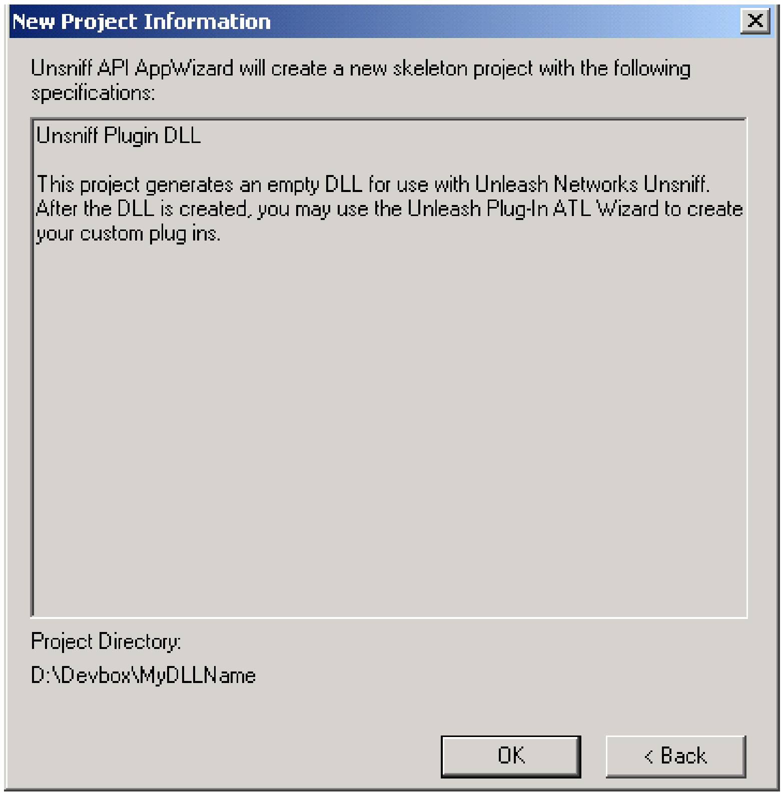
( ) Create new workspace
(●) Add to current workspace
[ ] Dependency of: dlltest

Platforms: [✓] Win32

Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6. Click OK. You will be presented with a confirmation screen (see below)



New Project Information

Unsniff API AppWizard will create a new skeleton project with the following specifications:

Unsniff Plugin DLL

This project generates an empty DLL for use with Unleash Networks Unsniff. After the DLL is created, you may use the Unleash Plug-In ATL Wizard to create your custom plug ins.

Project Directory:
D:\Devbox\MyDLLName

7. Click OK to confirm

   (End of Steps)

You now have a DLL project ready for you. Test your setup by building an empty DLL project.

- Press F7 to build the project

- The project should build without errors (as shown below)

```
msxml.idl
Compiling resources...
Compiling...
StdAfx.cpp
Compiling...
AcmeProtocols.cpp
Linking...
   Creating library Debug/AcmeProtocols.lib and object Debug/AcmeProtocols.exp

AcmeProtocols.dll - 0 error(s), 0 warning(s)
```

- You can use a Debug build during the testing phase. We recommend that you ship your DLL using the Release configuration only

Unsniff Evaluation Copy

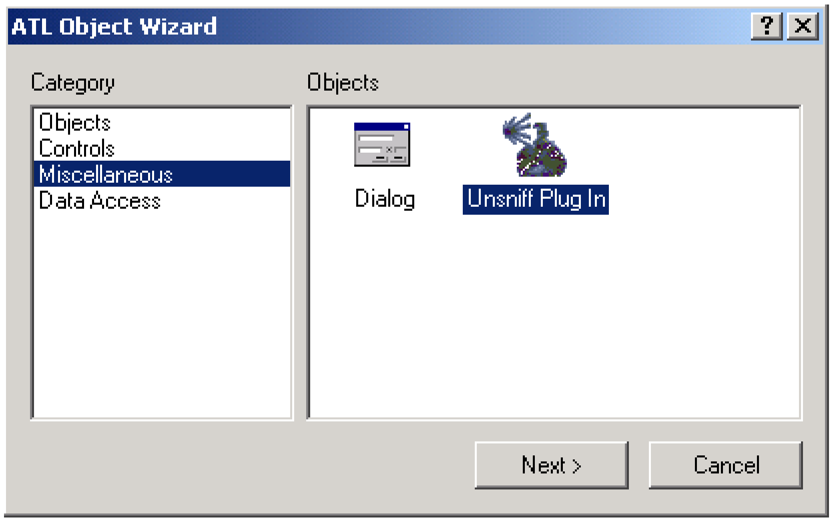
Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.1.3 Unsniff Plugin ATL COM Object Wizard

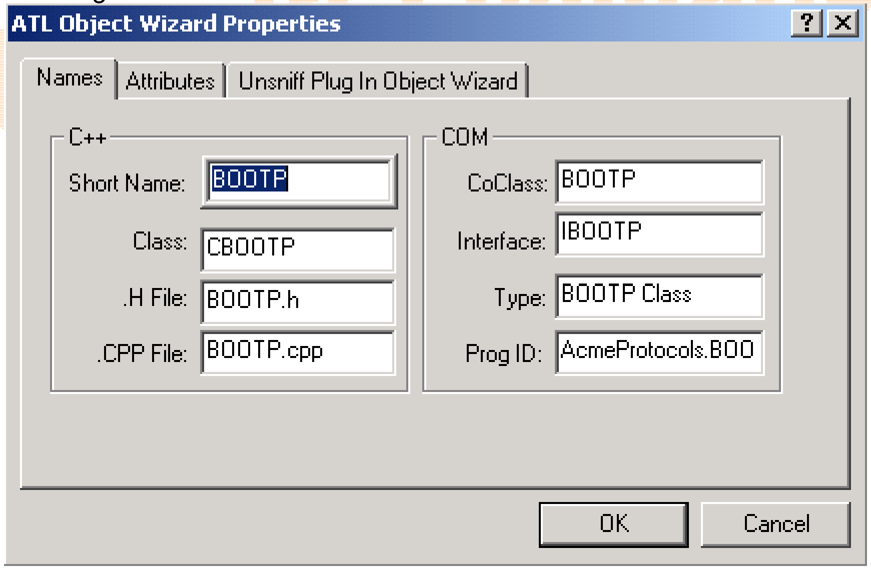
Purpose: To create an Unsniff Plugin object. This object will be responsible for handling your protocol.

Step by Step

1. Open an existing *Unsniff Plugin DLL Project*
2. Click on "*Insert*" -> "*New ATL Object*"
3. From the "*Category*" list; select "*Miscellaneous*" . You will be presented with the following screen.



4. Select the "*Unsniff Plug In*" item and press the "*Next*" button. This will bring up the following window



C++
Short Name: BOOTP
Class: CBOOTP
.H File: BOOTP.h
.CPP File: BOOTP.cpp

COM
CoClass: BOOTP
Interface: IBOOTP
Type: BOOTP Class
Prog ID: AcmeProtocols.BOO

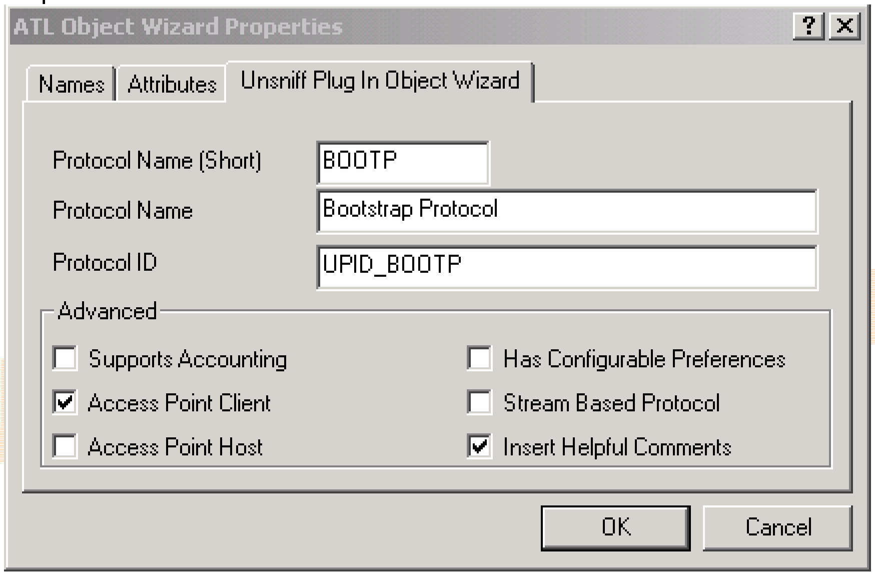
Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Specify the Short Name of the protocol you wish to handle. All other fields are filled in automatically. This table summarizes other entries that you can change.

Item	Description
Short Name	You must specify this. We recommend that you choose a name that is resembles the protocol name. In the example above we chose the short name to be BOOTP because we intend to write a plugin for the BOOTP/DHCP protocol.
These Fields are automatically filled in	
Class	The name of the C++ class
.H file	The header file for the C++ plugin class
.CPP file	The implementation (Cpp) file for the C++ plugin class
Interface	The name of the COM interface you wish to implement. If you are familiar with COM you can play around by adding your own interfaces here
Prog ID	The COM Prog ID under which this object will be registered. You can change this to something like AcmeSoftware.Prot to reflect your companies name.

5. After you are done with Step 4; click on the "Unsniff Plugin Object Wizard" tab. You will be presented with this screen.



Protocol Name (Short): BOOTP
Protocol Name: Bootstrap Protocol
Protocol ID: UPID_BOOTP

Advanced
- [ ] Supports Accounting
- [x] Access Point Client
- [ ] Access Point Host
- [ ] Has Configurable Preferences
- [ ] Stream Based Protocol
- [x] Insert Helpful Comments

Unleash Copy

This screen contains several important fields you are expected to fill up. See the table below for help.

Item	Description
Protocol Name (Short)	A short name for your protocol. We recommend that you restrict this name to less than 8 characters
Protocol Name	The full protocol name
Protocol ID	Each protocol in Unsniff is associated with a GUID. Enter a symbolic name for the GUID associated with this protocol. In the above example UPID_BOOTP is a "well known" GUID for

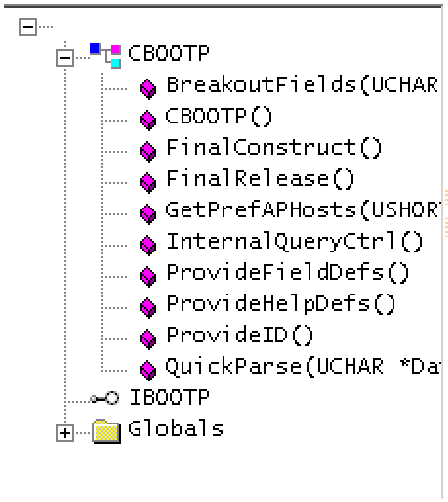
Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

	the BOOTP protocol
Supports Accounting	Does this plugin plan to account for sub protocols or messages. For example : a HTTP plugin may want to account for GET/POST/RESPONSE/ERROR messages
Access Point Client	Can this plugin be accessed via access points? For example : TCP Ports, UDP Ports, Ethernet Ethertypes, etc
Access Point Host	Will this plugin act as an Access Point Host? In other words, are there other protocols riding on top of this one?
Has Configurable Preferences	If this plugin has some custom configuration parameters that can be set/viewed via the "*Customize Plugins"* dialog in Unsniff.
Stream Based Protocol	Is this protocol stream based? Check this if your protocol runs on top of TCP and if it requires reassembly. You must check this if you plan to present your protocol as PDUs in Unsniff.
Insert Helpful Comments	The wizard strews helpful comments throughout the generated code. The generated code actually creates a dummy plugin that decodes two dummy fields. You can use this as a tool for getting off to a quick start.

8. Click OK to confirm

   (End of Steps)

Congratulations!
Your ATL COM plugin is now ready for use. The class view must now have the generated class in it. You are now ready to flesh out this class with details about your protocol.


```
CBOOTP
    BreakoutFields(UCHAR
    CBOOTP()
    FinalConstruct()
    FinalRelease()
    GetPrefAPHosts(USHOR
    InternalQueryCtrl()
    ProvideFieldDefs()
    ProvideHelpDefs()
    ProvideID()
    QuickParse(UCHAR *Da
IBOOTP
Globals
```

Evaluation Copy

The above picture shows a newly insert plugin object. Observe all the methods. We will get to know some of them in more detail in the next chapter.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.2 Installing Plugins

Plugins are really easy to install. All you have to do is to register your DLL. Unsniff has a discovery mechanism[11] which it employs to auto install plugins.

You can install your plugin using two methods.

5.2.1 Installation

Using Unsniff

This is the easiest way to install/uninstall a plugin.

1. Open Unsniff
2. From the main menu bar select "*Plugins*" -> "*Install*"
3. Select the DLL you wish to install

Use Regsvr32

Regsvr32 is a Windows command line tool, which is used to register/unregister COM components.

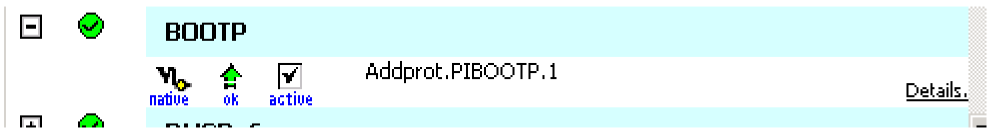
1. Type `regsvr32 <MyDllName.dll>`
2. To Unregister : Type `regsvr32 /u <MyDllName.dll>`

Verify installation

To Check if your plugin has been successfully installed

Unsniff Evaluation Copy

1. Open Unsniff
2. From the main menu bar select "*Plugins*" -> "*Manage Protocols*". This will open the Unsniff Protocol Plugin Manager
3. Locate the protocol, then click the '+' sign to show all registered plugins for that protocol. You should see your plugin here. A correctly installed entry should appear as shown below.



4. Make sure that the '*Ok'* icon is displayed. You can also click on the "*Details*" link to access further detail about the plugin

[11] In case you are curious about the discovery process; Unsniff uses COM Categories to locate all plugins.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.2.2 Activation

If your plugin is the only one registered for your protocol. There is no need for activation. However, there are cases when you may have multiple plugins for a given protocol.

Some scenarios are:

- You have written two or more versions of your plugin with different capabilities
- You have a test version of a plugin
- You have written a plugin for a protocol which is already supported by some other plugin

In those cases, you will need to select one of the many plugins for activation.

1. Open Unsniff
2. From the main menu bar select "*Plugins*" -> "*Manage Protocols*". This will open the Unsniff Protocol Plugin Manager
3. Locate the protocol, then click the '+' sign to show all registered plugins for that protocol. You should see your plugin here.
4. Select the plugin you want to be the *"Active Plugin".* Then click OK

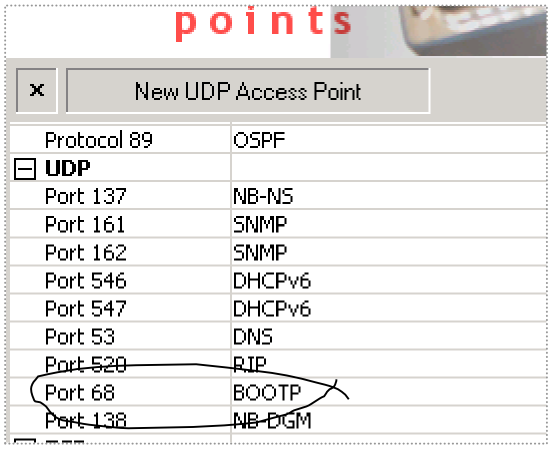
5.2.3 Access Points

One more thing you might want to check is whether your plugin has been wired up correctly to the Access Point framework in Unsniff. You can customize access points for your plugin using the Access Point Manager.

1. Open Unsniff
2. From the main menu bar select "*Plugins*" -> "*Access Points*". This will open the Unsniff Access Points Manager
3. Locate the host protocol(s). If your plugin runs atop UDP as well as TCP, then your host protocols are UDP and TCP
4. Check if your protocol has been wired up to the right access point value. In the example shown on the right, we check if BOOTP has been correctly wired up to UDP Port 68
5. You can customize the access points by adding more if you wish. For example: Observe that DHCPv6 is wired up to two UDP Ports 546 and 547
6. Click OK after verifying that everything is cool

(End of Task)

Unsniff Evaluation Copy



Protocol 89	OSPF
⊟ **UDP**	
Port 137	NB-NS
Port 161	SNMP
Port 162	SNMP
Port 546	DHCPv6
Port 547	DHCPv6
Port 53	DNS
Port 520	RIP
Port 68	BOOTP
Port 138	NB-DGM

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.2.4 Deployment

You have a lot of freedom to decide how to deploy a plugin DLL. As long as the plugin is registered – it really does not matter where it is stored. You are free to install your plugin as you wish.

Some guidelines for deployment:

- Do not install your plugin DLL in System folders (C:\Windows\System32 etc)
- Do not install on a network share for performance reasons
- Try to install all your plugin DLLs in a single folder
- Try to name the folder based on your company or organization name
- While uninstalling ensure that you unregister your plugin

5.2.5 Uninstall

You can remove plugins completely from the system if you wish.

Some guidelines for uninstall:

- Use “Plugins” -> “Uninstall” from the main menu. Select the DLL containing the plugin and press OK
- You can then remove the plugin from the system altogether if you wish.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.3 Hello World

This section is a step-by-step guide to writing a very simple plugin. You will get a chance to use the wizards and write an actual packet decoder.

5.3.1 The HelloWorld protocol

HelloWorld is a fictitious protocol. It is a ridiculously simple protocol – but it is perfect for this exercise.

The Protocol
This protocol defines only two fields *MessageType* and *Checksum.* Every message in this protocol consists of a *Message type* followed by a *Checksum*.

Field Name	Size	Explanation

MessageType	8 bits
	(1 octet)
		This is a message type field. Valid values for this field are:
		
		`0 – "Hello City"`
		`1 – "Hello Country"`
		`2 – "Hello World"`
		`3 – "Hello Martians"`
		`4-255 – "Hello Stranger"`

Checksum	32 bits (4 octets)	A checksum field

Working
This protocol is designed to run on UDP Port 10001

Test Script and Sample Capture
A test script is available in the *Samples/Test* folder. We will use this script to generate these packets. Alternately there is a *tcpdump format* capture file in the *Samples/Test* folder. We can import this file for testing our plugin.

Unsniff Evaluation Copy

Test script : **hw.rb** – a simple Ruby script to generate sample HelloWorld packets. To run this script type , `"hw <hostname or ipaddress>" eg( hw 192.168.1.1)`

Sample Capture: **helloworld.tcpd** – a few sample packets. You can import these packets using *File->Import->From TCPDump* menu.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.3.2 Instructions

The instructions are organized into (1) ***Basic Steps*** (2) ***Implementing Hello World*** (3) ***Testing***

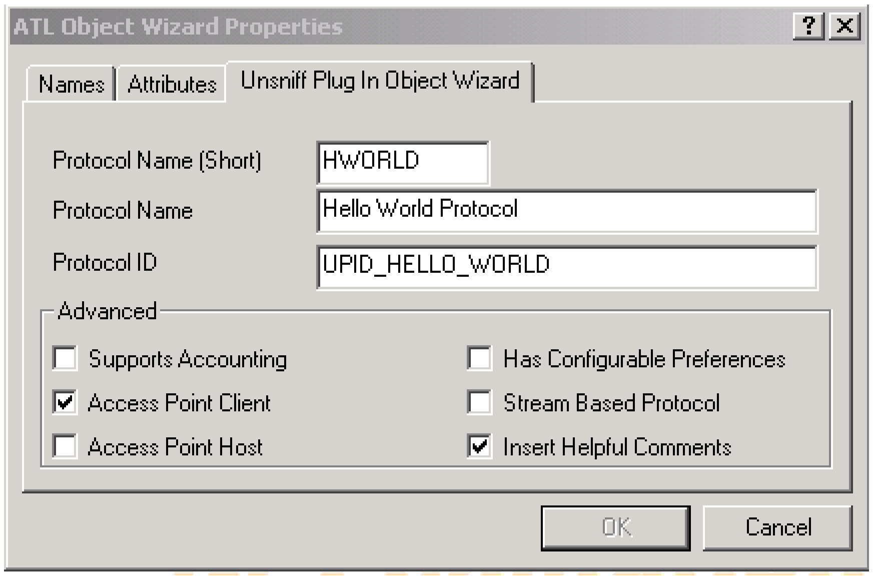
Basic Steps

1. Create a DLL project using the Unsniff Plugin AppWizard using instructions in *Sec 5.1.1* Name this DLL project "*HelloWorld*"

2. Create a plugin COM object using the Unsniff ATL COM Object Wizard using instructions in *Sec 5.1.3.* Use the following options.

   a. Name the object "HelloWorld"

   b. Enter the following details into the Wizard as shown in the screen below.

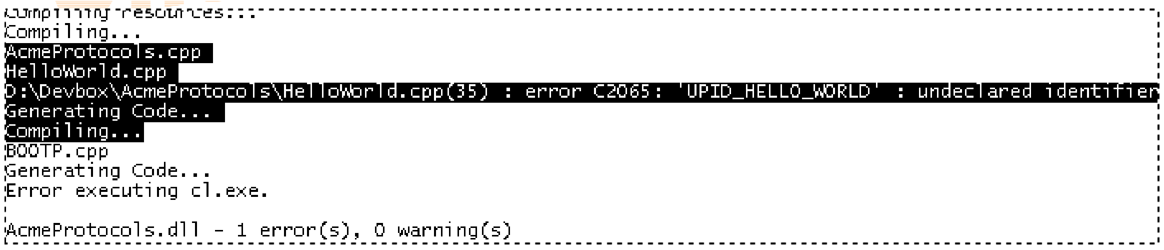


Protocol Name (Short): HWORLD
Protocol Name: Hello World Protocol
Protocol ID: UPID_HELLO_WORLD

Advanced
- [ ] Supports Accounting
- [x] Access Point Client
- [ ] Access Point Host
- [ ] Has Configurable Preferences
- [ ] Stream Based Protocol
- [x] Insert Helpful Comments

Unsniff Evaluation Copy

3. Try building the plugin by pressing F7. You will get this error:



```
Compiling resources...
Compiling...
AcmeProtocols.cpp
HelloWorld.cpp
D:\Devbox\AcmeProtocols\HelloWorld.cpp(35) : error C2065: 'UPID_HELLO_WORLD' : undeclared identifier
Generating Code...
Compiling...
BOOTP.cpp
Generating Code...
Error executing cl.exe.

AcmeProtocols.dll - 1 error(s), 0 warning(s)
```

4. As you know each protocol must be identified with a unique GUID *(See Section 2.1)*. We have given a name for it in the wizard *"UPID_HELLO_WORLD"* – but we have not specified a GUID yet. Do that now using the following steps

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

a. Open GUIDGEN.exe located in the Visual Studio directory

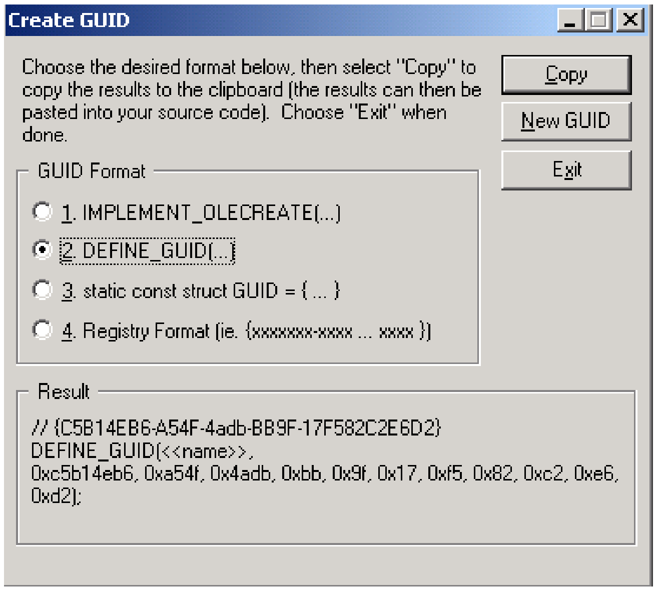
b. Use GUIDGEN.exe located in the Visual Studio directory to generate a GUID in DEFINE_GUID format

c. Copy the generated GUID using the Copy button

d. Open the generated CPP file *HelloWorld.cpp*

e. Paste the clipboard contents to the top of the file

f. Change the string `<<Name>>` to `UPID_HELLO_WORLD`



Choose the desired format below, then select "Copy" to copy the results to the clipboard (the results can then be pasted into your source code). Choose "Exit" when done.

GUID Format

- 1. IMPLEMENT_OLECREATE(...)
- [✓ 2. DEFINE_GUID(...)]
- 3. static const struct GUID = { ... }
- 4. Registry Format (ie. {xxxxxxx-xxxx ... xxxx })

Result

```
// {C5B14EB6-A54F-4adb-BB9F-17F582C2E6D2}
DEFINE_GUID(<<name>>,
0xc5b14eb6, 0xa54f, 0x4adb, 0xbb, 0x9f, 0x17, 0xf5, 0x82, 0xc2, 0xe6,
0xd2);
```

g. The newly added line should now look like this (in HelloWorld.cpp)

```
. .

// {6163A781-4781-4516-BD92-A8EE96E586E7}
DEFINE_GUID(UPID_HELLO_WORLD,
0x6163a781, 0x4781, 0x4516, 0xbd, 0x92, 0xa8, 0xee, 0x96, 0xe5, 0x86,
0xe7);
. .
```

5. Now we have defined the GUID. Time to build again. Now you should be able to build with no errors.

Unsniff Evaluation Copy

Implementing Hello World

6. Take some time to study the *CHelloWorld* class. Open up the *HelloWorld.cpp* file; you can see that the class has the following methods.

Method Name	**Purpose**
FinalConstruct	For Internal Use Only
FinalRelease	For Internal Use Only
InternalQueryCtrl	For Internal Use Only
ProvideID	**Provide Identification information** You can see that this section is already filled up from the information you supplied in the Wizard. The information elements you are expected to supply are: ▪ Name, Short Name, Vendor ▪ Which ICON (resource ID) should be used for this protocol ▪ Which color must be used for this protocol in the "raw" view

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

QuickParse	**Parse the packet in (`Data` of length `DataLength`) and provide the following information** ▪ Number of bytes that you can handle ▪ A description of the packet. This will be shown in the main packet index sheet in Unsniff ▪ What happens next (Disposition Code)? You can specify how the next layer protocol is selected or if this is the last protocol in the chain.
ProvideFieldDefs	**Define all the protocol fields** Observe that a couple of dummy fields have already been defined for you (*Example 1* and *Example 2*). You will be defining fields for the HelloProtocol in this method later
BreakoutFields	**Push the fields onto the field stack in the correct order** Observe that the dummy fields have been pushed onto the stack
GetPrefAPHosts	**Get Preferred Access Point Hosts** Specify how this protocol is wired up to the access point framework in Unsniff. Note that our HelloProtocol prefers to run on UDP port 10001, we will be specifying this information here
ProvideHelpDefs	**Provide Field Level Help definitions** Observe that two dummy field level help has already been provided. We just have to fill up help for our HelloWorld fields here

7. Define Ids for fields. Each field can have a unique identifier. The easiest way to do this is using C++ `enum.` Open up *HelloWorld.h*. You can see that the Wizard has already added a Enum typedef for you. You just have to replace the dummy entries with your protocol entries. In this case we need to define two fields.
(Note that the FieldIDs start at 0)

```
// Field IDs - define a numeric ID for each predefined field
typedef enum
{
        FID_MESSAGE_TYPE=0,
        FID_CHECKSUM,
} FIELD_IDS_T;
```

Unsniff Evaluation Copy

8. Method *ProvideID()* : We can leave it as it is. It already initialized correctly from the wizards.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

9. Method *QuickParse():* We want to customize this method using our knowledge of HelloWorld:

   - A Hello World packet is 5 bytes fixed
   - HelloWorld does not carry any other protocols
   - We want to construct a description of the packet

```
BOOLCHelloWorld::QuickParse(UCHAR * Data, USHORT  DataLength)

  static char buf[256];

  // A description
  typedef struct {
      BYTE   MsgType;
      DWORD  DataLength;
  } HDR_T, * PHDR_T;

  // Check minimum length
  if (DataLength < 5 ) {

      return FALSE;
  }

  // Get a packet description (alternately you can use UserGetEnumString
  //                              API function)
  PHDR_T ph = (PHDR_T) Data;
  const char * pszType="Unknown";
  switch (ph->MsgType)
  {
         case 0  : pszType = "Hello City";break;
         case 1  : pszType = "Hello Country";break;
         case 2  : pszType = "Hello World";break;
         case 3  : pszType = "Hello Martians";break;
         default : pszType = "Hello Stranger";break;
  }

  // Print a description
  //
  DWORD checksum = ntohl(ph->DataLength);
  sprintf(buf,"Hello World Packet, %s, checksum 0x%x", pszType,checksum);


  // Call UserInitQP with all the required information
  //
  UserInitQP(   5,           // 5 bytes
                DISPO_END,   // No more protocols running on HelloWorld
                IID_NULL,    // No next protocol ID
                buf          // Short packet description
            );

  return TRUE;
}
```

Tip

This above example is designed to demonstrate how to get at the raw packet data. There are several helper macros and functions available that can make your code shorter. You can use *UserGetEnumString*() to query a enum field. This will obviate the need for the switch(x) statement. You can also use the *HPTRVAL_xxxx* macros for network-host order conversions.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

10. Method *ProvideFieldDefs():* In this we add the two fields MessageType and Checksum
    - We will use an Enum field (*CUSNFEnumField*) for *MessageType* and *CUSNFNumericField* for *Checksum*
    - We want to label message type and display the checksum in hex
    - We want the MessageType field to be filterable

```
BOOL CHelloWorld::ProvideFieldDefs()
{
   USNF_BEGIN_ENUM_DEF(MessageTypes)
          ENUM_ENTRY(0,  "Hello City")
          ENUM_ENTRY(1,  "Hello Country")
          ENUM_ENTRY(2,  "Hello World")
          ENUM_ENTRY(3,  "Hello Mars"
   USNF_END_ENUM_DEF()

   // Message type field
   UserAddFieldDef(FID_MESSAGE_TYPE,
                  new CUSNFEnumField( _T("Message Type"),/* Name   */
                                 _T("MsgType"),          /* Short name*/
                                 FW_8BITS,               /* Width  */
                                 FS_LABEL|FS_FILTER,     /* Style  */
                                 USE_ENUM(MessageTypes)  /* Enum name*/
                              ));

   // Checksum field
   UserAddFieldDef(FID_CHECKSUM,
                new CUSNFNumericField(   _T("Checksum"),/* Name   */
                                         FW_32BITS,     /* Width  */
                                         FS_HEX         /* Style  */
                  ));


   return TRUE;
}
```

11. Method *BreakoutFIelds():* We just push the two fields onto the stack. *FieldStm* is a stream operator that provides access to the Field Stack.

```
BOOL  CHelloWorld::BreakoutFields(UCHAR * Data, USHORT DataLength)
{
    FieldStm << FID_MESSAGE_TYPE << FID_CHECKSUM;

    return TRUE;
}
```

Unsniff Evaluation Copy

12. Method *GetPrefAPHosts():* We need to specify our preferred access points. We know that HelloWorld is using UDP Port 10001. We just specify that here.

```
STDMETHODIMP CHelloWorld::GetPrefAPHosts(USHORT *pnHosts,ACCPT_T **ppHosts)
{
      USNF_BEGIN_ACCESSPOINT_DEF()
           AP_ENTRY(10001,UPID_UDP)
      USNF_END_ACCESSPOINT_DEF();

      return S_OK;
}
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

13. Method *ProvideHelpDefs():* We can provide field-level help defs for selected fields here. This adds immense value to the packet breakout display. The text you specify will appear in the Balloon Help along with the current value of the field.

```
BOOL CHelloWorld::ProvideHelpDefs()
{
   USNF_BEGIN_HELP_DEF()
      HELP_ENTRY( FID_MESSAGE_TYPE,
                  "HelloWorld Message Type\n"
                  "This determines the type of hello \n"
                  "Eg. 0=HelloCity, 1=HelloCountry, etc\n")

      HELP_ENTRY( FID_CHECKSUM,
                  "Message Checksum\n"
                  "A checksum of the HelloWorld Packet\n"
                  "including the header\n")


   USNF_END_HELP_DEF();
}
```

14. *That's it!* You can press F-7 to build your plugin now. It should build without errors

Testing your plugin

15. Build the plugin without errors (Press F-7)

16. Install your plugin (HelloWorld.dll) using instructions in Sec 5.2

17. Test your plugin by importing the *HelloWorld.pcap* sample capture file in the *samples/captures* directory:

    a. Import the capture file
    b. Observe how your protocol is displayed

Congratulations !

Now we can move on to more complex plugins

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.4 Handling Stream Based Protocols

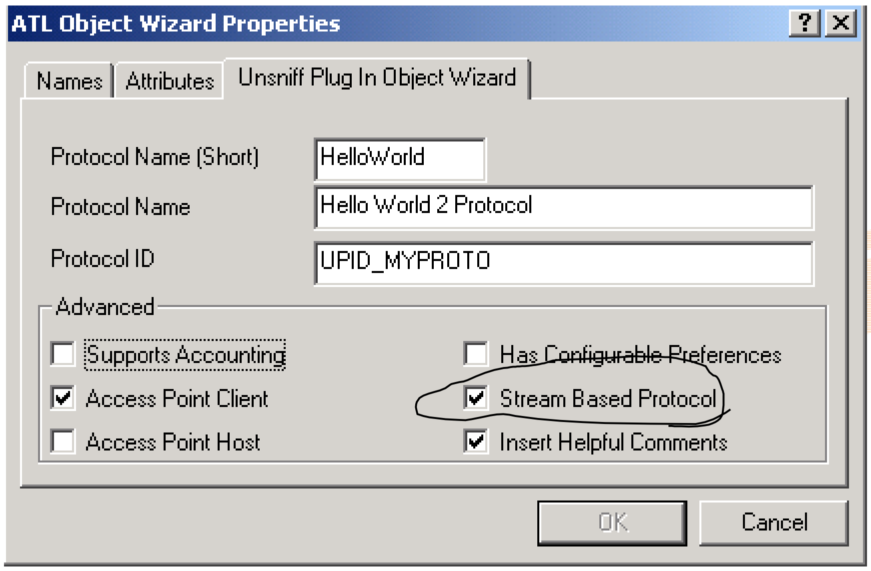
Stream based protocols are those that run on top of a layer such as TCP. These protocols are message based and do not care about packet boundaries. These protocols transmit data is PDUs. Unsniff provides the best support for creating and displaying PDUs.

Info
As of this release the only stream layer supported by Unsniff is TCP

Using Streams
Streams provide a reliable bi-directional data transmission pipeline between two endpoints. The Unsniff API framework handles all retransmissions, duplicate packets, missing packets, and reassembly. All you have to do is use the *IUSNFStream* interface and read from it as if it were a regular socket and construct PDUs.

5.4.1 Adding support for stream based protocols

The easiest way to add support for streams is at the time of using the Unsniff ATL COM Object Wizard. See *Section 5.2*



Protocol Name (Short)	HelloWorld
Protocol Name	Hello World 2 Protocol
Protocol ID	UPID_MYPROTO

Advanced

- [ ] Supports Accounting
- [x] Access Point Client
- [ ] Access Point Host
- [ ] Has Configurable Preferences
- [✓ Stream Based Protocol]
- [x] Insert Helpful Comments

ion Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

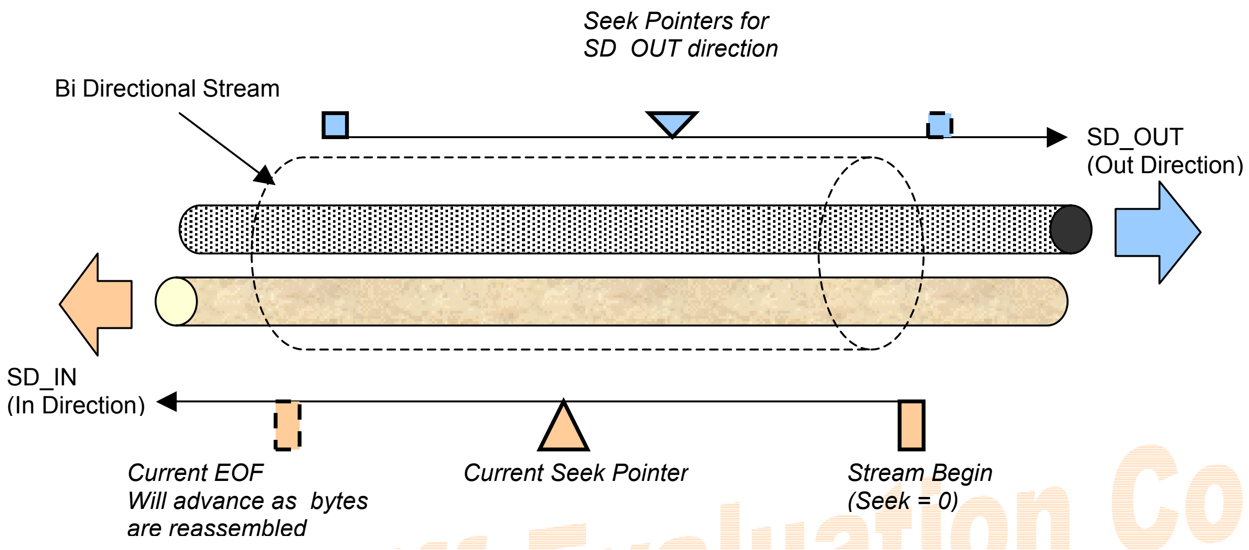
5.4.2 The IUSNFStream interface

This interface is used to wire up your plugin to the Unsniff stream handler mechanism. You can read from this interface as if it were a regular file. As bytes are reassembled by Unsniff you will be notified – you can then read from this stream and construct PDUs.

Concepts

The figure below shows a bi-directional stream. You can read from the stream just as you would from a file. The important thing is that IN and OUT directions are distinct. Each direction has its own seek pointer, EOF (End of File marker), and contents.

All stream operations are available via the *IUSNFStream* interface



Unsniff Evaluation Copy

Interface Methods

Each method requires you to specify a direction. Use the enum values:

- SD_IN : Stream Direction In (For TCP this is the direction of the SYN+ACK packet)
- SD_OUT : Stream Direction Out (For TCP this the direction of the initial SYN)

Method	Parameters	Purpose
GetSeekPos	In - Direction Out - Seek Position	Retrieve the current seek position of the stream. Both Peek and Read will start from this position
SetSeekPos	In - Direction In - Seek Position	Explicitly set the seek position

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

IncSeekPos	In – Direction In – Delta	Increments the seek position relative to the current position
IsEOF	In – Direction Out – True/False	Are there no more bytes to be read in the given direction *at the moment.*
Read	In – Direction In – Number of Bytes Out – Bytes	Read the specified number of bytes starting from the seek position. This command advances the seek position if the Read is successful.
Peek	In – Direction In – Number of Bytes Out – Bytes	Same as Read; but this command does not update the seek position.
GetSize	In – Direction Out – Size	Total number of bytes in the stream for the given direction
SetCookie[12]	In – DWORD cookie	You can attach a user-defined cookie to this stream. Use this if you want to distinguish between streams.
GetCookie	Out – DWORD cookie	The cookie that is currently associated with this stream
GetBytesRemaining	In – Direction Out – Bytes	Number of bytes available in this direction *at the moment*. This is nothing but : *EOF – Current Seek Position*
GetStreamSessTuple	Out – Stream Sess Tuple	The actual address information associated with this stream. Use this if you want to get at the Ipv4/Ipv6/ TCP ports
SeekPattern	In – Direction In – Pattern Out – Seek Position	Search for a pattern. Use this to synchronize a stream. For example: the BGP plugin will search for 16 bytes of 'FF'. That indicates the start of the BGP header
GetStreamID	Out – ID	An Integer ID for the stream assigned by Unsniff
GetStartTimestamp	Out – Timestamp	When did this stream start. For TCP streams, this is when the first SYN segment was seen.[13]
GetSrcAddress	Out – source address	Source address of the stream
GetDestAddress	Out – destination address	Destination address of the stream

Unsniff Evaluation Copy

[12] You can use cookies to save some user defined data for each stream
[13] If the SYN packet is not seen at all; when you are barging in on an existing TCP stream. Then the timestamp will be that of the first packet seen on that stream (could be either direction)

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.4.3 Writing Stream based Plugins

Study the class generated by the Unsniff Plugin ATL COM Object Wizard; when the Stream Based Protocol option is checked.

You will find that in addition to the usual methods (*See Section 5.3.1*). You now have the following additional methods.

Method Name	**Purpose**
IsNotifyProgress	**What kind of stream notification do you want ?** Progress: Notify as data is collected End Only: Notify only when stream is opened or closed Use *Progress* for long running streams (such as a BGP connection) Use *End Only* for short request-response protocols such as HTTP
GetNotifyChunkSize	**Notify Chunk Size** If you want to be notified on progress – what is the minimum amount of data that must collect before you want to be notified.
StreamStart	**Called by the framework when a new stream has started** The corresponding interface IUSNFStream is passed to you. You can use this method to take action if you want.
StreamClose	**Called by the framework when a new stream has ended** The corresponding interface IUSNFStream is passed to you. You can use this method to take action if you want.
StreamIncomplete	**The stream did not end properly** The corresponding interface IUSNFStream is passed to you. You can use this method to take action if you want. For TCP: This happens when the capture is stopped before Unsniff sees the FIN or RST sequence. This can also happen if there is an error in the FIN 4- way handshake.
StreamNotify	**Notification of activity on the stream** The given stream has some activity on it. You are supposed to read both directions of the stream and construct PDUs or UserObjects[14] from it.

Unsniff Evaluation Copy

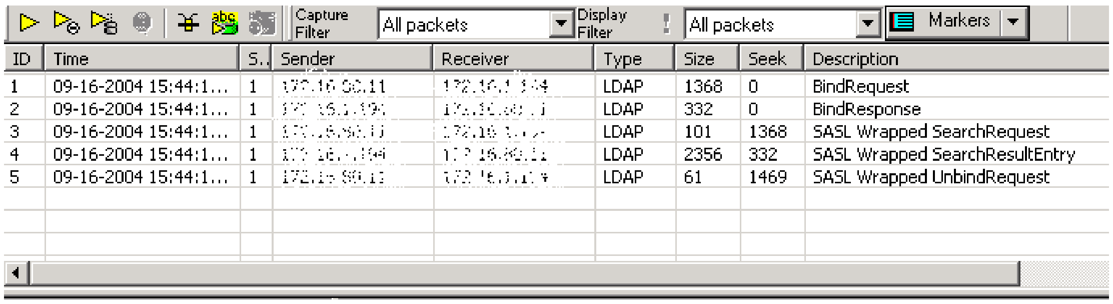
[14] Not discussed in this section

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.4.4 Stream Example (LDAP)

LDAP is a stream based protocol. LDAP messages can be much larger than Ethernet frames (1500 bytes) or many LDAP messages can fit inside an Ethernet frame. The example below works with streams. We will listen to streams and generate LDAP PDUs.

These PDUs will appear in the PDU Sheet of Unsniff as shown below. Notice that some packet sizes are much larger than the Ethernet frame size.



ID	Time	S..	Sender	Receiver	Type	Size	Seek	Description
1	09-16-2004 15:44:1...	1	[illegible]	[illegible]	LDAP	1368	0	BindRequest
2	09-16-2004 15:44:1...	1	[illegible]	[illegible]	LDAP	332	0	BindResponse
3	09-16-2004 15:44:1...	1	[illegible]	[illegible]	LDAP	101	1368	SASL Wrapped SearchRequest
4	09-16-2004 15:44:1...	1	[illegible]	[illegible]	LDAP	2356	332	SASL Wrapped SearchResultEntry
5	09-16-2004 15:44:1...	1	[illegible]	[illegible]	LDAP	61	1469	SASL Wrapped UnbindRequest

LDAP Pseudo-Code: Only *StreamNotify* has been customized. All other Stream functions shown in Table xx, have been left alone – unchanged from the Unsniff ATL COM Object Wizard generated code.

StreamNotify (Direction, Stream)

1. *While we will have enough to work in the given Direction – Repeat Steps 2-5*
2. *Peek at a small number of header bytes from which we hope to deduce the total message length. For LDAP 10 bytes are enough to parse the ASN.1 length, which indicates the total length of the message*
3. *Calculate the total length of the message from the "peeked" header*
4. *Check if you have the total number of bytes in the stream (from total length)*
5. *If the entire message has been read, then its is party time. We can go ahead and create the PDU. This PDU will then show up in the PDU Sheet*

Unsniff Evaluation Copy

Code Snippet

```
STDMETHODIMP CPILdapDemo::StreamNotify(/*[in]*/ IUSNFStream * pSTM)
{
      HRESULT hr;

      //
      // Process IN direction and OUT direction separately
      //
      hr=StreamNotify(SD_IN,pSTM);
      if (FAILED(hr))return hr;

      hr=StreamNotify(SD_OUT,pSTM);
      return hr;
}
// Helper function (not generated by wizard)
HRESULT CPILdapDemo::StreamNotify(STREAM_DIR_T eDir, IUSNFStream * pSTM)
{
      HRESULT       hr;
      BYTE          buf[LDAP_MIN_SIZE]; // enough to parse the packet length
      ULONG         nRead;

      // Handle all request / response pairs
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

```
        bool done=false;
        while (!done)
        {
                // End of file is reached, we are done
                VARIANT_BOOL vbEOF;
                pSTM->IsEOF(eDir, &vbEOF);
                if (vbEOF==VARIANT_TRUE) {
                        ATLTRACE("LDAP:NSTM:At EOF\n");
                        done = true;
                        continue;
                }

                // *Peek* LDAP Header
                hr=pSTM->Peek(eDir,LDAP_MIN_SIZE,buf,&nRead);
                if (hr==S_FALSE) {
                        ATLTRACE("LDAP:NSTM:Cant peek at LDAP Header\n");
                        done=true;
                        continue;
                }
                ULONG ASNType=0L;
                ULONG PDULength=0L;
                if (*buf==0x30) {
                        const BYTE * pptr = ASNBERUtils::ParseType(buf,&ASNType);
                        pptr=ASNBERUtils::ParseLength(pptr,&PDULength);
                        PDULength += (pptr-buf);
                }


                // Do we have complete message in stream
                ULONG uBytes=0L;
                hr=pSTM->GetBytesRemaining(eDir,&uBytes);
                if (uBytes<PDULength )
                {
                        // Not enough bytes have been read to create a PDU
                        done=true;
                        continue;
                }

                ULONG nRead=0L;
                LPCTSTR lpszDesc = "LDAP PDU (Big Encap)";
                pSTM->Peek(eDir,WORKBUF_SIZE,WorkBuf,&nRead);
                lpszDesc=GetPacketDescription(WorkBuf);

                //
                // Ok! Now are all set to create a LDAP PDU
                //
                IUSNFPDU      * pPDU=NULL;
                hr=m_pContainer->CreateNewPDU(&pPDU);
                if (FAILED(hr)||pPDU==NULL)
                {
                        ATLTRACE("LDAP, Cannot Create PDU , Out of Memory\n");
                        return E_OUTOFMEMORY;
                }
                pPDU->SetProtGUID(UPID_LDAP);
                pPDU->SetDescription(CComBSTR(lpszDesc),UserGetDBCookie());
                pPDU->SetStreamDataFromSeek(pSTM,eDir,PDULength);
                pSTM->IncSeekPos(eDir,PDULength);
        }

        m_pContainer->UpdateDisplay();
        return S_OK;
}
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.5 Defining Fields

Section 4, contains an in-depth look at the concept of fields in Unsniff. This section is designed to address issues specific to C++ plugins.

5.5.1 Alternate methods

Strictly speaking you do not need to define any fields. You can add fields directly during the breakout process by dynamically creating them and pushing them on to the field stack.

Consider the HelloWorld protocol

The code snippet below pushed the two predefined fields FID_MESSAGE_TYPE and FID_CHECKSUM on to the fiels stack.

```
        FieldStm << FID_MESSAGE_TYPE << FID_CHECKSUM;
```

This could also have been written like below without using any field definitions:

```
        FieldStm  << new CUSNFEnumField("Message Type", .. )

                  << new CUSNFNumericField("Checksum", FW_32BITS,. . ) ;
```

This technique has major disadvantages.

1. The performance will be slower
2. One of Unsniffs main design principles is to separate field definitions (which is largely a documentation task) from the task of analyzing a given packet. This technique subverts that principle. The result is confusing code that is difficult to write and maintain
3. You cannot use some key techniques like variables, auto-repeats, conditional fields, and autosizing.

There is however times when you may want to use this technique.

1. You do not know the field names until you see the packet itself.
   For example, if the protocol consists of a sequence of {Field-name, Field-value} pairs separated by a ':'. You may have to resort to this technique like shown below:

```
   LPCTSTR lpszFieldName  = GetFieldName(Data,':');    // parse name
   LPCTSTR lpszFieldValue = GetFieldValue(Data,'\n');  // parse value

   FieldStm  << new CUSNFStringField(lpszFieldName,
                                     TOBITS(_tcslen(lpszFieldName)+1),
                                     FS_PLAIN );

   FieldStm  << new CUSNFStringField(lpszFieldValue,
                                    TOBITS(_tcslen(lpszFieldValue)+1),
                                     FS_PLAIN );
```

2. This could be used to represent an unsupported part of the protocol. *In the example below we do not know how to parse 40 bytes of EXOTIC_MESSAGE*

```
  If (MsgType == MSG_EXOTIC_MESSAGE)  {

      FieldStm  << new CUSNFBinaryField("Unknown block of 40 bytes",
                                        TOBITS(40),
                                        FS_PLAIN );
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.5.2 Using FieldStm

FieldStm provides the main interface to the field stack. Every C++ plugin automatically has access to the FieldStm object. The following operators are defined for FieldStm:

Operator	Explanation
`FieldStm << FIELD_ID;`	Push a field identified by the integer `FIELD_ID` onto the field stack.
`FieldStm << "Field Name";`	Push a field named "`Field Name`" onto the field stack. Use this method for fields defined by XML. You are expected to ensure the uniqueness of the name while defining the XML field definitions. You may have to use the `<id>` attribute.
`FieldStm << new` *`CUSNFField`*`( . .);`	Push a dynamically created field onto the field stack. You do not have to worry about deleting the field.
`FieldStm << STM_BEGIN_RECORD("record 1");`	Begin a new record named "`record 1`". All subsequent fields pushed will be added to this record. This will end when a `STM_END_RECORD` is seen.
`FieldStm << STM_END_RECORD;`	End the previous `STM_BEGIN_RECORD`
`FieldStm << STM_COMMENT("My comment");`	A comment field. You can use comment fields to add your own elements to the details tree.
`DWORD dwType; FieldStm << STM_SAVE_NUMERIC(FID_TYPE, dwType)`	Push the numeric field `FID_TYPE` and save its value into the variable `dwType`. This only works for numeric fields
`FieldStm << STM_NULL;`	No op.
`FieldStm << STM_USE_DELAYLOADGROUP("Management Frames");`	We want to use the fields defined in the delay load group "`Management Frames`". *See section 4.x for information on delay loading.*

Unsniff Evaluation Copy

Usage Notes:

Typically, you will want to chain the fields together to create a highly readable and easy to maintain breakout.

In the example below we add the structure for a 802.11 Beacon.

```
FieldStm << STM_BEGIN_RECORD("Beacon")
                   << FID_MG_TIMESTAMP
                   << FID_MG_BEACON_INTERVAL
                   << FID_MG_CAPABILITY
                   << FID_MG_SSID;
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.5.3 UserAddFieldDef method

This is the method used to add a field definition. The prototype for this method is:

BOOL UserAddFieldDef (DWORD FieldId, CUSNFField * pField);

Parameters

Name	Description
DWORD FieldID	A 32 Bit field ID Each field defined must have a unique ID. The easiest way to do this is to use an enum. When you use the Unsniff ATL COM Object wizard, a sample enum block is generated for you in the *.h file.
*CUSNFField * pField*	A newly constructed field object. All field objects must be derived from the CUSNFField object.

Return values:

- TRUE if success
- FALSE if failure

If there was a failure, you can see the reason for failure in the Unsniff Log Window

Usage Notes:

Creating fields

You must explicitly create the field that is passed to UserAddFieldDef in the pField parameter. Do not use fields created on the stack or as member variables.

Unsniff Evaluation Copy

Destroying the field

You do not have to worry about destroying the *CUSNFField* derived object that you passed to *UserAddFieldDef*. The Unsniff API Framework will take control of managing the lifetime of the field. It will destroy the field when it is no longer needed.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.6 Accounting

Unsniff extends powerful accounting capabilities to your plugin. Your protocol can account for number of message subtypes within your protocol. You get to define what types you want to account and perform the actual accounting itself. The results of your accounting are shown in the "*Statistics"* tab in the Unsniff application.

Info
The accounting feature is not available to pure XML plugins

5.6.1 Add Accounting support

The easiest way to support accounting in your plugin is to check the "Supports Accounting" checkbox in the Unsniff ATL COM Object Wizard . This will automatically add the skeleton code needed to support accounting.

The methods related to accounting are:

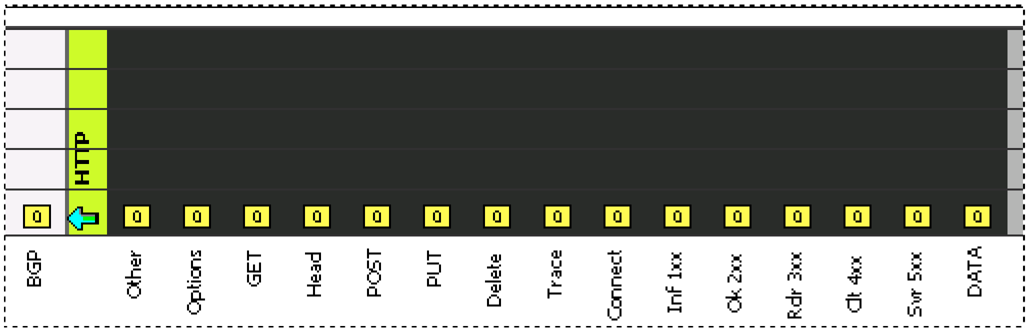
Method	Parameters	Purpose
ProvideAcctDefs	None	Provide a list of items you wish to account for. This is typically a set of sub message types. Each accounting item is identified by a unique ID and a short name.
QuickParse	In - Data In - DataLength	This function is called for each packet. It is your job to quickly scan the packet and update the correct accounting items based on the data contained in the packet.
UserUpdateAcct	In – Accounting Item ID	Indicate to the API that we want to update the accounting item with the ID. The Unsniff API automatically increments the packet count and byte count for this item.
USNF_ACCT_ENTRY Macro	Item Id Prot GUID Name	Use this macro to define a single accounting item. The Prot GUID parameter is used to attach a protocol to this accounting item. Use prot GUID of GUID_NULL if there is no chance of expanding this accounting item further.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.6.2 An Example

Consider the HTTP protocol. We want to account for each of the sub types. We want the statistics for HTTP protocol to appear as shown below. The statistics is available in the Statistics Sheet in Unsniff. You can expand each protocol to access sub type accounting by clicking on the "→" on the graph.



Assign IDs

First we have to assign a unique integer ID for each accounting item. This easiest way to do this is to create an enum in your class header file. If you used the wizard, this block of code is already created for you – you just have to replace the contents with your accounting items.

```
// Accounting items
typedef enum
{
        ACCT_OTHER=0,
        ACCT_OPTIONS,
        ACCT_GET,
        ACCT_HEAD,

        // . . . define other  items here

        ACCT_SERVER_ERROR,

        ACCT_DATA,
} ACCOUNTING_ITEMS_T;
```

Unsniff Evaluation Copy

Provide Acct Defs

We have to flesh out the *ProvideAcctDefs()* method. In this method you must define each accounting item name, its ID (see above step), and a protocol GUID. You can use a GUID_NULL if the accounting item is not attached to any specific top-level protocol.

```
BOOL    CPIHTTP::ProvideAcctDefs()
{
        // Add accounting items below, see example
        USNF_BEGIN_ACCT_DEF()
                USNF_ACCT_ENTRY(ACCT_OTHER,   GUID_NULL,   "Other");
                USNF_ACCT_ENTRY(ACCT_GET,     GUID_NULL,   "GET");

                 // . . . define other message types

                USNF_ACCT_ENTRY(ACCT_POST,    GUID_NULL,   "POST");
                USNF_ACCT_ENTRY(ACCT_PUT,     GUID_NULL,   "PUT");
                USNF_ACCT_ENTRY(ACCT_SUCCESS,GUID_NULL,   "Ok 2xx");
        USNF_END_ACCT_DEF()
}
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Perform Accounting
In the QuickParse method, look at the packet data and update accounting information for your items.

```
BOOL   CPIHTTP::QuickParse(UCHAR * Data, USHORT  DataLength)
{

       if ( PacketType (Data) is "GET" )
              UserUpdateAcct(ACCT_GET);


       if ( PacketType (Data) is "POST" )
              UserUpdateAcct(ACCT_POST);

       // . . . define other message types
```

5.6.3 Add accounting manually

The Unsniff ATL COM Object Wizard is the best way to add accounting support to a new plugin. If you have already created a plugin without accounting support and later wish to add accounting support to it; you will have to do it manually.

In your header file *"MyPlugin.h*":

- Include the file `#include "USNFAcctImpl.h"` and add the following line to the inheritance chain of your class `public CUSNFAcctImpl<IUSNFAccounting>`

- Add the IUSNFAccounting interface to the COM MAP as shown below:

```
BEGIN_COM_MAP(CMyPlugin)

       . . . other interfaces

       COM_INTERFACE_ENTRY(IUSNFAccounting)

       . . . other interfaces

END_COM_MAP()
```

- Add the following function prototype for the ProvideAcctDefs() method.

```
public:
       Virtual BOOL ProvideAcctDefs();
```

In your implementation file *"MyPlugin.cpp*":

- Implement the *ProvideAcctDefs*() method

<End of task>

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

5.7 Configuration Parameters

Ok so your have written a plugin, now you want to allow the user to customize your plugin. This is done via configuration parameters. You can specify your parameters inside your plugin – these parameters are then managed via the *Plugs->Configuration* menu bar.

Info

The configuration parameters feature is not available to pure XML plugins.

5.7.1 Add Configuration Support

The easiest way to add configuration support is to check the "Has Configurable Preferences" in the Unsniff Plugin ATL COM Object Wizard. This will automatically add the skeleton and even a couple of sample configuration parameters. You just have to replace the sample configuration parameters with your own. Each configuration parameter consists of:

- A Key String used to uniquely identify the parameter. The key must be unique within your plugin
- A Data type. This helps Unsniff manage the parameter by using the correct GUI controls.

The methods related to configuration support are:

Method	Parameters	Purpose
ProvideConfigDefs	None	Provide a list of configuration parameters. This list will be managed by Unsniff.
OnConfigChange	None	The configuration might have changed. You should update your state with the new configuration by issuing *UserQueryConfigXXXX*() calls.
UserQueryConfigXXXX()	In – Key Out – Value	Different variants of this method exist. Select the method that is appropriate for the data type of the configuration entry.

Unsniff Evaluation Copy

5.7.2 Unsniff Plugin Configuration

The configuration parameters you define will be managed by Unsniff. You can bring up the dialog by selecting "Plugins → Configure" menu item. Your plugin will have its parameters grouped together as a separate node.

- Specify the value for the configuration parameter from the right column
- Click on any parameter to access help for that parameter

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

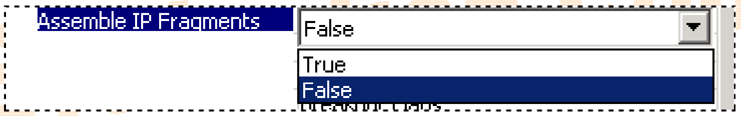
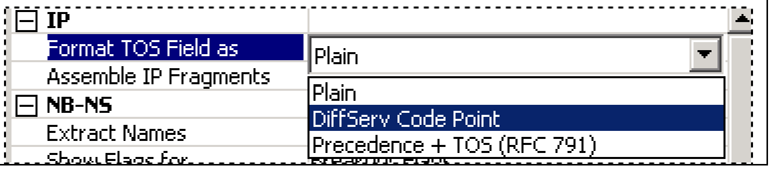
5.7.3 The USNF_xxxx_CONFIG_ENTRY macro

Types of Configuration Parameters:
You define configuration parameters using the *USNF_xxxx_CONFIG_ENTRY* macro. The xxxx must be substituted for the correct data type. You must define each configuration entry using the correct macro for that type. The macro has the following prototype.

USNF_xxxx_CONFIG_ENTRY(***Key****,* ***Name****,* ***Help String****,* ***Default Value****)*

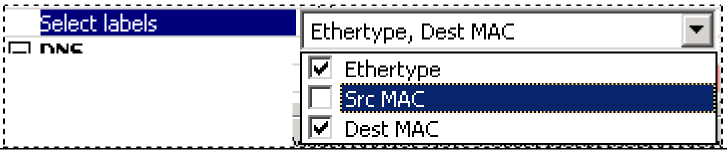
Type	Explanation
Key	A string that uniquely identifies the parameters. This key must be unique within your plugin
Name	The name of the configuration parameter. This appears in the Unsniff "Customize Plugins" dialog as shown here.
Help Text	A string explaining in detail the purpose of the configuration parameter. This enables you to define clean and self-documenting parameters. The help text is shown below
Default Value	The initial or default value of the parameter. This varies by the data type of the parameter.

Data types
The following data types are defined.

Type	Explanation
String USNF_STRING_CONFIG_ENTRY	The parameter is a string.
Numeric USNF_NUMERIC_CONFIG_ENTRY	The parameter is a 32-bit number.
Boolean USNF_BOOL_CONFIG_ENTRY	A Boolean (True/False) parameter  Assemble IP Fragments: False (True / False)
Enum USNF_ENUM_CONFIG_ENTRY	The user can choose a single item from a list of options. An example is shown below. The use can select one of three options to format the TOS field.  IP — Format TOS Field as: Plain (Plain / DiffServ Code Point / Precedence + TOS (RFC 791)); Assemble IP Fragments; NB-NS — Extract Names

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Enum Multiselect USNF_ENUM_MULTISELECT_CONFIG_ENTRY	The configuration parameter consists of several items that the user has specified. In the example below, the user can choose any combination of the three options.  Select labels DNS Ethertype, Dest MAC ☑ Ethertype ☐ Src MAC ☑ Dest MAC
Color USNF_COLOR_CONFIG_ENTRY	The parameter is a color.

Query the current value of the configuration parameter:

The UserQueryConfigXXXX() method is used to query the current value of the parameter.

```
BOOL    UserQueryConfigBool(LPCTSTR lpszKey, bool * pOut);
BOOL    UserQueryConfigNumeric(LPCTSTR lpszKey, DWORD * pOut);
BOOL    UserQueryConfigColor(LPCTSTR lpszKey, COLORREF * pOut);
BOOL    UserQueryConfigString(LPCTSTR lpszKey, LPCTSTR * pOut);
BOOL    UserQueryConfigChoice(LPCTSTR lpszKey, USHORT  * pOut);
BOOL    UserQueryConfigMultichoice(LPCTSTR lpszKey, int nArraySize, USHORT * pArray)
```

All the methods are pretty simple. They query a parameter using a key and return the value into a pointer.

The UserQueryConfigMultiChoice() method uses an extra parameter. The pArray is an array of size nArraySize. Upon return from the function the array contains a map of what items were set. Array[0] = 1 if the first item was selected, Array [1] = 0 means the second item was not selected.

5.7.4 An Example

A simple example is in order. We want the user to specify two parameters:

1. Whether the plugin should extract any hostnames by listening to packets (Default TRUE)

2. Allow user to select any combination of Flags to display in detail (there are 3 choices shown to the user Plain Flags, NB Flags, NS Flags) (Default : Show Plain and NB Flags)

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Provide Configuration Parameter Definitions

Fill out the ProvideConfigDefs() method generated by the wizard. Note that the entire block is defined within the BEGIN_CONFIG_DEF and END_CONFIG_DEF section.

```
BOOL   CMyPlugin::ProvideConfigDefs()
{
       // String tables
       USNF_BEGIN_STRING_TABLE(FlagOptions)
              USNF_STRING("Breakout Flags")
              USNF_STRING("Breakout NB Flags")
              USNF_STRING("Breakout NS Flags")
       USNF_END_STRING_TABLE()

       // Config Defs
       USNF_BEGIN_CONFIG_DEF()
              USNF_BOOL_CONFIG_ENTRY("p.extractnames",
                  "Extract Names", "Listen to NetBIOS Name Service exchanges
                                   (Response, Registration). Add all hostnames
                                   in these messages into the Unsniff Name
                                   Cache",
                                    true)


              USNF_ENUM_MULTISELECT_CONFIG_ENTRY("p.breakouts",
                  "Show Flags for", "A separate layout will appear for each
                                     Flags field selected",
                                     Options1,
                                     "0,1")
       USNF_END_CONFIG_DEF();
}
```

Handle Configuration Changes

In this method we save the user specified configuration parameters into member variables. We can then adapt the plugin behavior as per the users wishes. We also demonstrate how to use the multiselect parameter.

```
BOOL   CMyPlugin::OnConfigChange()
{
       UserQueryConfigBool("p.extractnames",&copt.fExtractNames);

      USHORT SelectedFlags[3];

      UserQueryConfigMultiChoice("p.breakouts",3,SelectedFlags);
      For (int I=0;I<3;I++) {
            if (SelectedFlags[I] ) {
                Cout << "Flagss " << I << "Was selected by user" << Endl;
            }

      }

       return TRUE;
}
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6 XML Plugins

One of the most powerful features of the Unsniff Network Analyzer is the ability to leverage the benefits of XML to quickly create protocol plugins.

There are two ways in which you can use XML:

1. Use XML to create the complete protocol plugin. This includes defining fields, identifying the plugin, creating a packet description, working with access points, and handling stream based protocols.

2. Use XML to define fields only. A C++ plugin is still responsible for pushing the fields on to the field stack for a given packet and for all other functions such as streams, access points. This option is a subset of option 1.

This section is organized into:

- Using an XML Editor
- Installing XML plugins
- The Unsniff Protocol Plugin XML Specification
- “Hello World” Step-by-Step
- Plugin using both XML and C++

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.1 Using an XML Editor

XML Documents are notoriously hard to read with a plain text editor. This is especially true if you want to load large documents for editing. We recommend that you use your favorite XML editor if you plan on working on a decent size XML plugin.

We recommend the free XML Notepad from Microsoft. It is a very basic XML editor but it is stable and above all free. If you have access to a full featured XML editor such as XML Spy, it is even better !!

You can download *XML Notepad* from: http://msdn.microsoft.com/xmlnotepad

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

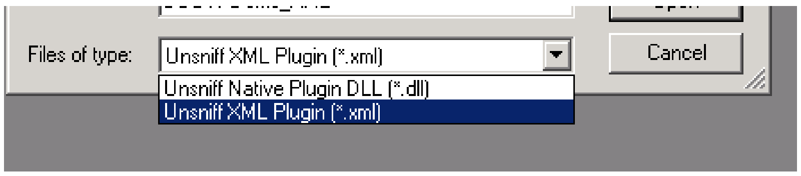
6.2 Installing XML Plugins

An XML Plugin is nothing more than a simple XML document. An XML document can contain at most one plugin. XML Plugins are installed using the Unsniff Application.

To install a XML plugin automatically simply copy it to the InstallPath\xmlplugs folder. These plugins are discovered automatically and loaded.

To manually install a XML plugin follow these steps:
Assume that you have written an XML plugin for the BOOTP protocol in an XML file called "BOOTPDemo_XML.xml"

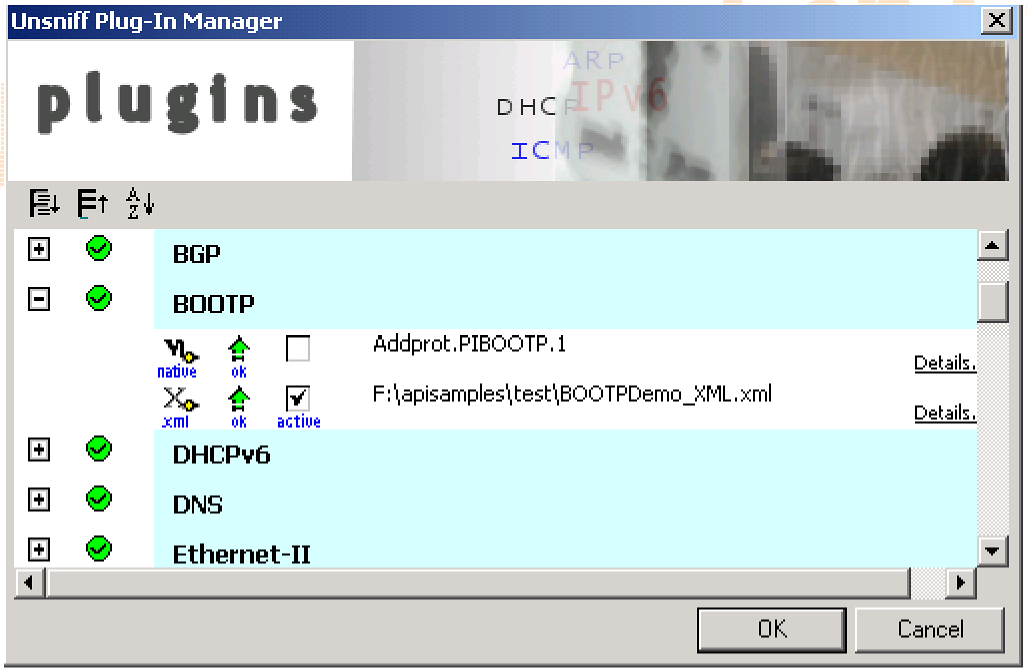
- Select Plugins → Install from the main application toolbar in Unsniff
- In the "*File Open*" Dialog, select "Unsniff XML Plugins (*.xml)" from the drop down list, this will allow you to select XML files.



Files of type: Unsniff XML Plugin (*.xml)

Unsniff Native Plugin DLL (*.dll)
Unsniff XML Plugin (*.xml)

- The select plugin will be installed by Unsniff. If there is an error, please view the log window using *View → Log Window.*
- To check if the plugin has been installed successfully. Click on *Plugins → Manage Protocols.* Scroll down to the protocol name and expand the row. You should see your XML plugin. Note that an "X" icon distinguishes XML Plugins, native (C++) plugins are shown with a 'N' icon. In the example below, our plugin has been installed and also activated. The native plugin is now inactive, all BOOTP packets will be decoded using our XML plugin.



Unsniff Plug-In Manager

BGP
BOOTP
native ok Addprot.PIBOOTP.1 Details..
xml ok active F:\apisamples\test\BOOTPDemo_XML.xml Details..
DHCPv6
DNS
Ethernet-II

Version 1.3 Feb 3, 2006

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3 Unsniff Protocol Plugin XML Specification

6.3.1 Top-Level Structure

The top-level structure of the XML protocol plugin document is shown below.

```
<?xml version="1.0" encoding="UTF-8" ?>

<USNFProtocol>

   . . . common items
   . . . such as vendor name, conformance, color,
   . . . root field

     <FieldDefs>
          . . . field definitions (mandatory)
     </FieldDefs>

     <DescriptionString>
          . . . how to construct a packet description
     </DescriptionString>

     <AccessPoints>
          . . . list of access points to attach this protocol
     </AccessPoints>

</USNFProtocol>
```

The entire XML document is specified in a block with a <*USNFProtocol*> tag. Apart from <USNFProtocol>, the only mandatory tag is the top level <FieldDefs> tag. The Description String and AccessPoints tags are optional.

Top Level Tags:

Tag	Usage
<USNFProtocol>	The base tag. This tag defines properties of the protocol. Some child element may be left out – if you are only defining fields in this XML document.
<FieldDefs>	Define all your fields within this tag. At least one FieldDefs tag must be present in the XML document. Additional FieldDefs may be present if you are defining a Delay Load Group
<DescriptionString>	This block defines how to format a packet description for a given packet[15]. This block is optional.
<AccessPoints>	This block defines the access point (e.g., host protocol, port) information for this protocol. Alternately you can configure access points manually using the Access Point Manager within Unsniff. This block is optional

Unsniff Evaluation Copy

[15] This is the same description that you constructed in the *QuickParse()* method in a C++ plugin.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.2 USNFProtocol

This is the top most tag in the XML document. This tag is used to:

- Serve as a container for all other tags
- Define common elements such as name, shortname, color, icon for the protocol

Attributes

Attribute name	Whether Mandatory	Description
<id>	Mandatory	A unique name for the protocol
<name>	Optional	Full name of the protocol. If not specified <id> is used.
<shortname>	Optional	Short name of the protocol. If not specified <id> is used
<protid>	Mandatory	The protocol GUID in registry format. Use a predefined protocol ID from the USNFProtocols.c file or generate your own protocol ID for custom protocols. See Section 4.fd.f.xd

Elements

Attribute name	Whether Mandatory	Description
<vendor>	Optional	Your company or organization name.
<conformance>	Optional	Description of what RFCs, standards documents, or specifications were used for this protocol.
<color>	Optional	What color is used to represent this protocol in the raw bytes view. Use #RRGGBB hex format
<icon>	Optional	An icon file (*.ICO) which is used to denote this protocol
<version>	Optional	Version number of this plugin XML. The format is <Major>.<Minor>
<rootfield>	Mandatory	The name of the *entry point field* of this protocol.
<FieldDefs>	Mandatory	This element contains a block of field definitions
<DescriptionString>	Optional	This element specifies how to format a descriptive name for a given packet
<AccessPoints>	Optional	Contains access point information
<Defaults>	Optional	Specify default style for fields. If no style is specified in the <FieldDef> tag, then Unsniff will use the default style.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Example:
The following excerpt is taken from the BOOTP protocol in the samples directory.

Info
Note that we are using the protocol id for BOOTP from the USNFProtocols.h file (UPID_BOOTP). This is because BOOTP happens to be a "well known" protocol. If we had used a new GUID then this protocol would have been treated as a whole new protocol (even though it has the same name BOOTP)

```
<UsnfProtocol id="BOOTP"
              shortname="BOOTP"
              name="Bootstrap protocol"
              protid="{CF2428E1-4843-48CF-B7DD-CCC9E5AE4BC1}">

   <vendor>Unsniff Plugin API Demo</vendor>
   <conformance>RFC 2131, RFC 2132</conformance>
   <color>#F0F000</color>
   <icon>mybootp.ico</icon>
   <version>1.0</version>
   <rootfield>BOOTPMessage</rootfield>

   <FieldDefs>
      - - - -
   </FieldDefs>

   <DescriptionString>
      - - - -
   </DescriptionString>

   <AccessPoints>
      - - - -
   </AccessPoints>

</UsnfProtocol>
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.3 DescriptionString

The purpose of this tag is to:

- Allow Unsniff to construct a meaningful text description of a packet. This is called the packet description.

The goal is to quickly construct a packet description without requiring a full packet breakout. This description appears in the packets sheet and is accessible via the Scripting interface. If this tag is not present then the Protocol name is used as the description.

Attributes
This tag does not use any attributes.

Elements

Element name	Whether Mandatory	Description
<Format>	Mandatory	A string containing optional special wildcards for parameters. If specified, these wildcards will be substituted with the actual parameter values.
<Params>	Optional	If the <Format> tag uses parameter wildcards. The <Params> section must actually define those wildcards.

6.3.3.1 Params

The Params tag defines a set of parameters for use with the <DescriptionString> tag. This tag is valid only as a child of the DescriptionString tag.

Attributes:

This tag does not use any attributes.

Unsniff Evaluation Copy

Elements

Element name	Whether Mandatory	Description
<Param>	Mandatory	A parameter for use with wildcard substitution. *Parameter number k* will be used to substitute wildcard *$k*

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.3.2 Param

The Param tag defines a single parameter. A parameter is nothing but a predefined field. The corresponding field definition <FieldDef> must already be present in the XML document. Parameters can also be fillers, which are dummy fields used to account for unimportant fields.

Attributes:

Attribute name	Whether Mandatory	Description
<ref>	Optional (Either ref or filler must be present)	A reference to a pre-defined field. *The pre-defined field cannot be in a DelayLoadGroup.*
<filler>	Optional (Either ref or filler must be present)	A count of number of *BITS* to fill. Use fillers liberally to account for unimportant fields for the purpose of constructing a packet description

Elements
This tag does not define any elements

Example Description String:

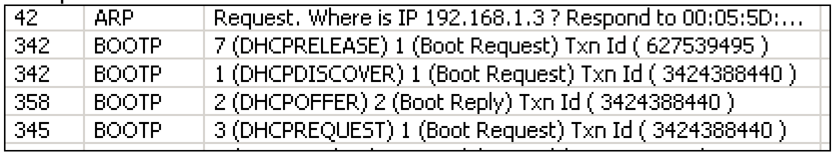
When presented with a BOOTP packet we want to construct a description in the following format **"DHCPDISCOVER Request Txn Id (99380)"**

In the following fragment $1 = Field "*Option Type*" $2 = *Junk (Filler 24 bits)* $3 = Field "*Transaction ID*" and so forth.

```
- <DescriptionString>
    <format>$5 $1 Txn Id ( $3 )</format>
  - <Params>
      <Param ref="Option Type" />
      <Param filler="24" />
      <Param ref="Transaction ID" />
      <Param filler="1872" />
      <Param ref="DHCP Message" />
    </Params>
  </DescriptionString>
```

Unsniff Evaluation Copy

Woot! Your XML plugin can construct a flexible packet description. Admittedly this is not as powerful as a C++ plugin; but it sufficient for many protocols.
Sample shown below



42	ARP	Request. Where is IP 192.168.1.3 ? Respond to 00:05:5D:...
342	BOOTP	7 (DHCPRELEASE) 1 (Boot Request) Txn Id (627539495)
342	BOOTP	1 (DHCPDISCOVER) 1 (Boot Request) Txn Id (3424388440)
358	BOOTP	2 (DHCPOFFER) 2 (Boot Reply) Txn Id (3424388440)
345	BOOTP	3 (DHCPREQUEST) 1 (Boot Request) Txn Id (3424388440)

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.4 AccessPoints

The purpose of this tag is to:

- Wire up your protocol into the access point framework in Unsniff.

If the AccessPoints tag is not present. You will have to manually wire up your protocol using the Access Point Manager in Unsniff.

Attributes
This tag does not use any attributes.

Elements

Element name	Whether Mandatory	Description
<AccessPoint>	Mandatory	Define a single access point. You can have as many of these tags as you want.

6.3.4.1 AccessPoint

The AccessPoint tag defines a single access point. This tag is valid only as child element of the *<AccessPoints>* tag.

Attributes:

Attribute name	Whether Mandatory	Description
<hostid>	Mandatory	Protocol GUID of the host protocol in registry format. Consult the USNFProtocols.c file for a list of standard protocol GUIDs. **Example**: If you are writing a plugin for a protocol on top of UDP. Specify the protocol GUID of UDP here. `{14D7AB53-CC51-47e9-8814-9C06AAE60189}`
<apvalue>	Mandatory	Access point value. This depends on the host protocol. For TCP and UDP this represents the Port number. For Ethernet this value is the EtherType, etc.

Unsniff Evaluation Copy

Elements
This tag does not define any elements

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Example:

Consider the following example of the SNMP protocol. This protocol will ride by default on UDP ports 161 and 162 and TCP ports 161 and port 162

```
- <AccessPoints>
    <AccessPoint hostid="{14D7AB53-CC51-47e9-8814-
      9C06AAE60189}" apvalue="161" />
    <AccessPoint hostid="{14D7AB53-CC51-47e9-8814-
      9C06AAE60189}" apvalue="162" />

    <AccessPoint hostid="{77E462AB-2E42-42ec-9A58-
      C1A6821D6B31} " apvalue="161" />
    <AccessPoint hostid="{77E462AB-2E42-42ec-9A58-
      C1A6821D6B31} " apvalue="162" />

  </AccessPoints>
```

UDP 161,162

TCP 161,162

6.3.5 Defaults

The Defaults section is used to assign global default values for elements. At present the only default element supported is the <styles> element

Attributes:
This tag does not use any attributes

Elements:

Element name	Whether Mandatory	Description
<styles>	Mandatory	Specify default styles for fields. This style is assumed to be present even if not explicitly specified in the <FieldDef> element.

Unsniff Evaluation Copy

Usage Notes:

Use the Defaults section if a majority of fields in your protocol have certain properties that would be a pain to specify on a field-by-field basis. An example usage scenario.

- If all strings in your protocol are Unicode. You can specify <styles> Unicode </styles> in the defaults section. This way you can avoid repeating the same style for all fields. If a certain field is not Unicode, you can negate the style using <style > ~unicode </field> in the <FieldDef> element.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.6 FieldDefs

The purpose of this tag is to:

- Serve as a container individual fields using the <FieldDef> tag *(note that this singular)*
- Support Delay Loading

Attributes

Attribute name	Whether Mandatory	Description
<delayloadgroup>	Optional	Used for Delay Loading. You can use this tag to specify that all the fields defined in this block belong to a delay load group by the name present in the tag. This attribute can only appear when the parent of the corresponding <FieldDefs> tag is the <USNFProtocols> tag. In other words, only top-level FieldDefs can have use the <delayloadgroup> attribute.

Elements

Element name	Whether Mandatory	Description
<FieldDef>	Mandatory	Defines an individual field.

Examples

```
<FieldDefs>
     <FieldDef>
    -    Field 1 definition
    -    here
     </FieldDef>

     <FieldDef>
    -    Nested FieldDef
         <FieldDefs>
             - Nested fields
         </FieldDefs>
     </FieldDef>

</FieldDefs>
```

FieldDefs in action (including nested fields)

```
<UsnfProtocol>

<FieldDefs>
  - - all main fields go here
</FieldDefs>

<FieldDefs delayloadgroup =
            "Mgmt Frames">
     - - all fields in delay load
     - - group go here
</FieldDefs>
```

DelayLoadGroup for "Mgmt Frames"

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.7 FieldDef

This is the key tag that appears the maximum number of times in most Unsniff XML documents. A good understanding of using this tag will get you 3/4ths of the way to understanding this specification.

The purpose of this tag is to:

- Define a protocol field

Attributes:

Attribute name	Whether Mandatory	Description
<name>	Mandatory for non-ref fields	The full field name.
<shortname>	Optional	A short form of the full field name. This will be used by Unsniff if there are any space constraints while displaying the full name. Try to keep the short name < 10 characters long
<id>	Optional	All fields are expected to have a unique field name. However in many cases it is not possible to adhere to this rule. In those cases, use the id tag to specify a unique id for the field
<ref>	Optional	Unsniff allows you to *rubber-stamp* the field definitions of an already defined field. Use this tag to copy all properties of this field from an existing field.

Elements

Unsniff Evaluation Copy

Element name	Whether Mandatory	Description
<fieldtype>	Mandatory for non-ref fields	The field type can be one of the Unsniff built in field types or a user-defined field type.
<sizebits>	Optional	The size of the field in bits. The minimum fieldsize is 4 bits for a standalone field and 1 bit for a member of a flags field.
<styles>	Optional	Field styles control how fields are interpreted and presented to the user.
<sizeexpr>	Optional	An expression using variables that evaluates to the size of the field in bits.
<reps>	Optional	An expression using variables that yields the number of repetitions of the field
<choiceexpr>	Mandatory for Choice fields	The expression is evaluated to yield a number or a string. Based on this result – one of the candidate members of the choice field is

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

		activated.
<choiceval>	Mandatory for members of Choice fields	This is a number attached to a candidate member of a Choice field. The <choiceval> is used to match and activate this field among all the other candidates
<choicevalstring>	Mandatory for members of Choice fields using a string choice (eg. OID)	This is a string attached to a candidate member of a Choice field.
<condition>	Optional	This is a logical expression that evaluates to true/false. If the expression evaluates to true – then this field is active, if false – this field is treated as a NO-OP.
<variable>	Optional	Attach a variable to this field. The variable name is a string containing no spaces or special characters. The first letter of the variable name must be alphanumeric. Do not prefix the variable name with a $ sign.
<protid>	Mandatory for external fields	For external fields, this protid indicates which protocol will help us decode this field.
<helptext>	Optional	The online field level bubble help text. This can span several lines. You can also include some HTML tags in the text if you wish
<asntag>	Optional	The user defined ASN tag attached to the field. Examples are: UNIVERSAL: 4, APPLICATION 10, etc.
<asntagexplicit>	Optional	The explicit ASN Tag attached to this field.
<asntagimplicit>	Optional	The implicit ASN Tag attached to this field
<BitList>	Only applicable to fields of type ASNBERBit	Specifies a bit list. This is a sequence of <bit> tags which specify the position of each bit in the ASN.1 Bitstring
<EnumList>	Mandatory for enum fields	Contains a list of name , value pairs which constitute this enum. You can also define a long name for your enums.
<OIDEnumList>	Optional only for ASNBER fields	Contains a translation of OIDs to human readable strings. Use for ASN.1 based plugins only.
<FieldDefs>	Optional	Child fields of this field. You must use this tag to define child fields of:

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

		▪ Records ▪ Flags ▪ Choices ▪ ASN Seq/Set/Choice

<recorddisplayformat>	Only valid for record fields or ASNBERSequence fields	Format a description of the record field based on the values of child fields.
		
		See Sec. 4.5.7 for details of formatting record fields.
		
		For example:
		
		`" Type = $1, length = $2, value = $3"`
		
		`where $1,$2,$3 are the first second and third child fields of the record.`

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.7.1 The ID attribute

The ID attribute assigns a unique ID to the field. Use this attribute if the name cannot be made unique. Take the example of the SMB protocol: there is a field called "Flags" which is defined in many commands including the header and inside the TREE_CONNECT command. Using the ID attribute we can assign a unique ID to each flags field without changing the name of the field.

```
<FieldDef name="Flags" id="Main.Flags">
  <fieldtype>Flags</fieldtype>
  <styles>filter,sublayout</styles>
  <helptext>Flags and Flags2 contain bits which, . . .

 - - - - -

<FieldDef name="Flags" id="TreeConnect.Flags">
        <fieldtype>Numeric</fieldtype>
        <sizebits>16</sizebits>
        <helptext>Addl info (bit0 set = disconnect Tid)</helptext>
</FieldDef>
```

Having defined the unique IDs, it is easy to push fields on to the field stack.

```
FieldStm << "Header.Flags";       // the flags in the main header

FieldStm << "TreeConnect.Flags"; // in the tree connect command
```

6.3.7.2 Ref fields

Use the ref attribute if you want to copy the properties of an already defined field. This idiom Is very useful if you have a lot of identical fields that just differ by their name . Using the ref attribute helps you cut down on unnecessary field definitions.

Assume that we have a complex bit field named Capabilities. This field appears in many records. Instead of defining the field again in each record, you can define it once. Then you can just create a <ref > to represent this field inside all the records that use it.

```
<FieldDef name="Capabilities">
    <fieldtype>Flags</fieldtype>
    <styles>sublayout</styles>
    <sizebits>32</sizebits>
  <FieldDefs>
    - <FieldDef name="CAP_EXTENDED_SECURITY"
    - - - Complex field definitions of the Capabilities flag
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

You can just create a ref to the "*Capabilities*" field in the *SessionSetup* record as shown below:

```
- <FieldDef name="SESSION_SETUP_ANDX NT-LM 0.12 Req">
    <fieldtype>Record</fieldtype>
  - <FieldDefs>
    + <FieldDef name="WordCount" shortname="Wcnt">
    + <FieldDef name="AndXCommand" + <FieldDef
      name="AndXReserved"
   - - - add other fields of the record here
     <FieldDef ref="Capabilities/>"
```

Since the capabilities field is used in many records, we just saved ourselves a lot of duplicate and difficult to maintain elements.

6.3.7.3 FieldType

The field type element specifies the type of field. The field type is case insensitive – so lower case, upper case, and mixed case are accepted by Unsniff.

Valid field types are:

Element name	Explanation
"ASNBER" *"ASNBERChoice"* *"ASNBERGroup"* *"ASNBERHeader"* *"ASNBERSequence"* *"ASNBERSet"* *"ASNBERBit"* *"Binary"* *"Choice"* *"Enum"* *"External"* *"Flags"* *"GUID"* *"IPAddress"* *"IPv6Address"* *"MACAddress"* *"Numeric"* *"Numeric64"* *"Record"* *"String"* *"Pad"*	Denotes the built-in field type for this field
Any name starting with a % sign.	A user defined field type.

See *Chapter 4 : Fields* for more information about Fields.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.7.4 Styles

The <styles> element is used to attach one or more styles to the field.

- You can combine styles by separating them with commas
- Styles are case insensitive
- You can clear a style using the unary '~' symbol[16]
- The styles specified here are combined with the styles in the top level <Defaults> section of the document.

The following styles are valid:

Style name	Explanation
"plain" *"filter"* *"filterdisplaystring"* *"label"* *"sublayout"* *"novisual"* *"nodetail"* *"novalue"* *"protocolitem"* *"hostorder"* *"unicode"* *"showbitflags"* *"align"* *"reverse"* *"fill"* *"hex"* *"hostorder"* *"conditional"* *"optional"* *"hidden"* *"compressed-visual"* *"signed"*	The style to be attached to the field. If a specified style is not applicable to a particular field type, it is silently ignored.

Unsniff Evaluation Copy

Usage Notes:

Some examples:
- The LANMAN field is a UNICODE string aligned natively. The native alignment of a UNICODE string is understood by Unsniff to be 16 bits. Extra padding is automatically added to align it if necessary. We also set the label and Unicode style here.

```
- <FieldDef name="NativeLanMan">
    <fieldtype>String</fieldtype>
```

[16] You can also use the '!' symbol to specifically negate a style

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

```
        <styles>unicode,label,align</styles>
        <helptext>Servers native LAN Manager type in
          unicode</helptext>
```

- Here we set the styles for a numeric field. We will clear the *hostorder* style, which has been set in the top-level Defaults section.

```
  - <FieldDef name="TxnId">
        <fieldtype>Numeric</fieldtype>
        <styles>label, ~hostorder</styles>
```

6.3.7.5 Expressions

As you have seen in Section 4.xx.x. variables are used to save the value of a field that has already been pushed onto the field stack. The Unsniff API allows you to use variables in the form of expressions. There are two types of expressions:

- Integer Mathematical Expression (used by <sizeexpr> , <reps> , <choiceexpr> tags
- Logical expressions (used by <condition> tag)

Math Expressions

Consists of a string of variables and operators. The operators allowed are: **" + - * / ( ) "**
The precedence rules for these expressions are the same as any programming language. You can use any number of variables to form these expressions.

Examples:

- The size of field *SecKey* is *4 x TotalDataLength – 2 x OptionalWordCount*

```
<FieldDef name="SecKey">
    <fieldtype>binary</fieldtype>
    <sizeexpr> 4 * $TotalDataLength – 2 * $OptWordCount </sizeexpr>
```

- The record *Address Record* will repeat *TotalDataLength – HeaderLength / 16* times.

```
<FieldDef name="Address Record">
    <fieldtype>record</fieldtype>
    <reps> ($TotalDataLength – $HeaderLength) /16 </reps>
    <FieldDefs>
        <FieldDef>
        // .. members of Address Record
```

Logical Expressions

Variables and Logical Operators are used to create this expression.
The operators allowed are: **" && || == != ~ ( ) > < >= <= ".** The precedence rules for these expressions are the same as any programming language. You can use any number of variables to form these expressions.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Examples:

- The 128 bit field *AuthBlock* is *present only when $AuthType is 1 or 3*

```
<FieldDef name="AuthBlock">
    <fieldtype>binary</fieldtype>
    <sizebits>128</sizebits>
    <condition>
        $AuthType == 1 || $AuthType == 3
    </condition>
```

6.3.7.6 EnumList

This tag is valid only when the fieldtype is *"enum"* or *"ASNBER"*. For all other field types, the EnumList tag is ignored and a warning is generated.

The EnumList tag is used to:

- Define a block of enumerated entries and attach it to a field

Attributes:
This tag does not define any attributes

Elements

Element name	Whether Mandatory	Description
<enum>	Mandatory	Define a single enum entry. The <EnumList> tag is actually just a collection of <enum> tags

Unsniff Evaluation Copy

The <Enum> tag
The Enum tag is used to define a single enumerated entry. The Enum tag is valid only within an EnumList tag.

Attributes:

Attribute name	Whether Mandatory	Description
<name>	Mandatory	The text name of the enumerated entry.
<longname>	Optional	An optional descriptive name of the enumerated entry
<value>	Mandatory	The integer corresponding to this entry. All <enum> tags within an <EnumList> must have distinct values. You can specify the value in hex or decimal
<oid>	Optional only when child of OIDEnumLIst	A OID in dotted decimal format. Eg. "1.3.6.1.2.1.1.1"

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Elements
This tag does not define any elements

Example <*EnumList*>

In the example, we define an enumerated field called NtStatus. You can see that the value can be in hex or decimal and that the longname attribute is optional.

```
- <FieldDef name="NtStatus">
    <fieldtype>Enum</fieldtype>
    <sizebits>32</sizebits>
    <styles>protocolitem,label</styles>

  - <EnumList>
      <enum value="0x00000000"
            name="SUCCESS"
            longname="STATUS_SUCCESS"/>

      <enum value="1"
            name="STATUS_WAIT_1" />

        - - - // define all other enum values here
```

6.3.7.7 OIDEnumList

This tag is valid only when the fieldtype is *"ASNBER"*. For all other field types, the OIDEnumList tag is ignored and a warning is generated.

The OIDEnumList tag is used to:

- Map OID (Object Identifiers) to human readable names

Attributes:
This tag does not define any attributes

Elements

Element name	Whether Mandatory	Description
<enum>	Mandatory	Define a single enum entry. The <EnumList> tag is actually just a collection of <enum> tags

Unsniff Evaluation Copy

The <Enum> tag
The Enum tag is used to define a single enumerated entry. The oid attribute of the <enum> tag is used when it is a child element of OIDEnumList See previous section on <EnumList> for details about the enum tag.

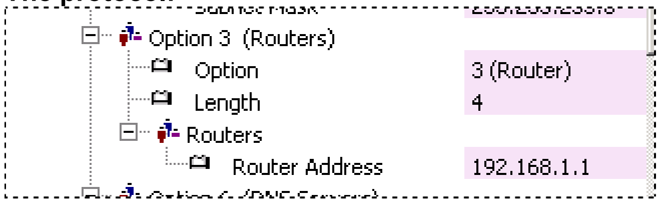
Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

6.3.7.8 Sub Fields using FieldDefs

Using FieldDefs tag as an element of FieldDef can create nested fields. This technique is used to create records, flags, choice, ASN structures, etc.

All the rules of Section 6.3.6 (FieldDefs tag) apply to its usage as a child of a FieldDef. Consider the example shown below. The Field Option 3 contains an auto-repeat sub field called *routers.*

The protocol:



```
Option 3 (Routers)
    Option              3 (Router)
    Length              4
    Routers
        Router Address  192.168.1.1
```

XML fragment showing nested fields:

```
- <FieldDef name="Option 3 (Routers)" id="Option_3">
    <fieldtype>record</fieldtype>
  - <FieldDefs>
      <FieldDef ref="Option" />
      <FieldDef ref="Length" />
    - <FieldDef name="Routers">
        <fieldtype>record</fieldtype>
      - <FieldDefs>
        - <FieldDef ref="Router Address">
            <reps>$OptionLength / 4</reps>
            <helptext>- - - </helptext>
          </FieldDef>
        </FieldDefs> // end of Routers record
      </FieldDef>   // end of Routers field
    </FieldDefs>  // end of Option 3 (Routers) record
  </FieldDef>   // end of Option 3 (Routers) field
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

7 Advanced Plugins

In addition to protocol plugins, the Unsniff Network Analyzer also supports other types of plugins. These plugins are written in C++ and require a decent amount of COM knowledge to get started. They make Unsniff even more extensible and customizable. You can write entire applications on top of the Unsniff platform.

7.1 Types

Type	Purpose
Eavesdroppers	Eavesdrop plugins can subscribe to a single or all protocols. The Unsniff framework will callback each registered eavesdrop plugin as packets are processed. This is the way you can hook into the Unsniff packet processing engine.
Name Resolvers	It is excruciatingly painful for a human network analysis expert to spend time looking at strings that make sense only to machines. You can write name resolvers for any entity that you think will be useful to you. For example: A name resolver for "Station ID" might convert a string like "*NJ7784.883:99102*" to something like "*NJ Walmart – WAN–9 Port 102*" . This conversion can be done by looking up a database or file or any other scheme. It is up to you. Other examples are: • MAC address to OUI + Address • SNMP OID to name • 802.11 BSSID to test network name
Custom User Objects	User Objects are high level entities that are of interest to the network analysis expert. Unsniff allows you to define your own user object types in addition to the standard user object types (such as image, HTML, file, audio). You can then attach a custom renderer to display this object. User Object will prove to be invaluable in real life network analysis. Some potential types: • "***Config File***" : Boot files via TFTP to remote systems. DSL or Cable providers can check the file(s) that were actually sent. This is conclusive proof. • "***Streaming Audio***" : You can extract the audio stream and play it back to monitor it for quality and content • "***Chat Session***" : Security personnel and law enforcement may want to extract chat sessions as user objects • "***Trades***" : Developers and testers of a stock broker trading software may want to extract the actual data sent to exchanges for debugging purposes

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Custom User Object Renderers	You can control how a particular user object is rendered in the *User Objects Sheet* in Unsniff. You may wish to show the chat session in a HTML window or plain text. You may wish to format the text in a particular way for display. Using custom user object renderers. • You can **render** your user object as you wish. You will be given a HDC to paint on • You can **play** your user object if it supports that kind of activity • You can **save** you user object in a particular format If you do not provide a renderer, Unsniff will still be able to display the user object in the sheet and save it.
User Interface Plugins	You can integrate your plugin completely into the Unsniff user interface. You can add menu items, toolbars, buttons, and dialogs into Unsniff. Using a simple callback model you will be notified when a toolbar button or menu item is selected. You can then display your user interface items.
Custom Sheets	Custom Sheets are full blown ActiveX controls that appear as a separate sheet in the Unsniff capture window. Custom Sheets are the "top of the heap" among all types of plugins. You can use custom sheets to develop your own full featured applications on top of the Unsniff framework. Some examples[17]: • **VOIP call monitor**. Show a panel with currently active calls, finished calls, calls experiencing high jitter. This panel has a sub panel with an embedded media player / signal level meter. • **Wireless Manager** : Show a panel with a snazzy graphical display of the channels. Show signal levels on each channel. Show active networks in a separate channel. • **Statistics** : Show station wise traffic. You can duplicate the matrix displays found on some old protocol analyzers or develop your own 3D visualizations of traffic A Custom Sheet can also be an Eavesdrop plugin. This will enable your cool network application to seamlessly plugin into (1) the user interface and (2) the packet processing engine

Unsniff Evaluation Copy

[17] None of these sheets actually exist ! These are just ideas for network analysis experts who can also code. You can use your imagination, let us know via our message boards of any cool sheets you are developing.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

7.2 Development Environment

All the advanced plugins are COM components that implement one (or more) specific interfaces. We strongly recommend that you write plugins using **Microsoft Visual C++ and Active Template Library (ATL)**. You could also write some plugins in VisualBasic, C# , VB.NET or any other language supporting COM early binding. At this point we cannot support plugins written in other languages. All plugins must be housed in a DLL project.

Info
Unsniff does not provide the Unsniff ATL COM Plugin Object Wizard for advanced plugins. You have to use the standard ATL COM Object Wizard. You can still use the Unsniff Plugin APP Wizard to create your DLL Project.

Tip
The best way to get started is to copy a similar sample plugin from the Unsniff API "samples" directory. Note that you still have to use the ATL COM Object wizard to generate the correct GUIDs and registry scripts for your plugin. You can then use the sample plugin to fill in the relevant sections.

7.2.1 View installed plugins

- All advanced plugins need to be registered with Windows. You can use the "Plugins"→"Install" menu item from the main application menu to install them
- You can view installed plugins using the "Plugins" → "Plugin List" from the main application toolbar. For more information about the Plugin List dialog, press F1 while the dialog is open.

Unsniff Evaluation Copy

7.2.2 COM Tips

- Use FinalConstruct and FinalRelease to perform your intiaiization (see the samples)
- Use ATLTRACE statements to insert debugging text in your code
- Use CComBSTR to convert normal strings to the BSTR format required by COM
- Use W2A to convert BSTR to normal strings

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

7.3 Eavesdroppers

Samples located in folder "samples/plugins/eaves_demo "

Eavesdrop plugins implement the *IUSNFEavesdrop* Interface.

The IDL definition of this interface

```
/*
 * IUSNFEavesdrop - Eavesdropper
 *    Generic eavesdropping interface
 *    Allows a plugin to listen to interesting events happening
 *    inside the packet processing engine of Unsniff.
 */
  [
    object,
    uuid(8D063005-6959-4b4e-996F-F3A854E8E929),
    dual,
    helpstring("IUSNFEavesdrop Interface"),
    pointer_default(unique)
  ]
  interface IUSNFEavesdrop: IDispatch
  {
    [id(1), ..] HRESULT  AttachSite([in] IUSNFContainer * pCTR);
    [id(2), ..] HRESULT  DetachSite();
    [id(3), ..] HRESULT  GetEavesdropTypes([out, retval] DWORD *pVal);
    [id(4), ..] HRESULT  GetEavesdropProtocolID([out] GUID * pProtID);
    [id(5), ..] HRESULT  NewLayerPacket([in] IUSNFProtData * pLayerData);
  };
```

Methods

Name	Parameters	Purpose
AttachSite	IUSNFContainer	This plugin has been attached to the Unsniff application. • save this interface pointer if you want to avail of services from the container later • perform any initialization
DetachSite	None	You plugin is no longer attached to the application
GetEavesdropTypes	DWORD	What types of eavesdropper are you ? Currently this value returned is ignored. You can just return S_OK from this method
GetEavesdropProtocolID	GUID	Return the GUID of the protocol you are interested in. • return GUID_NULL if you want to eavesdrop all packets at the lowest layer
NewLayerPacket	IUSNFProtData	This is the main callback method from the application. A new packet was received, the interface pointer (IUSNFProtData) contains the data at the layer you requested

See Appendix X for details about other interfaces

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

7.4 Name Resolvers

Samples located in folder "samples/plugins/resolver_demo "

Resolver Target

Name resolvers are used to convert a random collection of bytes to a meaningful human readable string. Name resolution is triggered when :

- A resolver is attached to a field using *UserAttachResolver(..ResolverTargetGUID)*
- A resolver is attached via the <resolveid> XML tag

The resolver target is the entity that is being resolved to a human readable string. Each resolver target must have a unique GUID. This is known as the "Resolver Target GUID".You must first determine the Target GUID for your resolver. Some resolver targets are pre-defined in Unsniff (see the file USNFProtocols.c for a list). If you cannot find them there you need to define your own resolver target.

Built in Resolver Targets from *USNFProtocols.c*

```
// Predefined RESOLVER TARGET IDs
// GUIDs (e.g.. SNMP OID = {21DDAF85-FCA6-4e42-B593-D221A3633A6E} )
DEFINE_GUID(URSV_IPADDRESS,…);      // IP Address
DEFINE_GUID(URSV_IP6ADDRESS,…);     // Ipv6 Address
DEFINE_GUID(URSV_MACADDRESS,…);     // MAC Address
DEFINE_GUID(URSV_OSPFAREA_NAME,…); // OSPF Area Name
DEFINE_GUID(URSV_SNMPOID,…);        // SNMP OID
```

Name resolvers plugins implement the *IUSNFCustomResolver* interface. If you are defining your own resolver target then you must also implement the *IUSNFCustomResolverTarget* interface

The IDL definitions

IUSNFCustomResolver

```
/*
 * IUSNFCustomResolver - Custom Resolver
 * Methods to handle name resolution of user defined entities
 */
[
      object,
      uuid(4C446404-A08C-4e02-A04B-BF08EE018879),
      dual,
      helpstring("IUSNFCustomResolver Interface"),
      pointer_default(unique)
]
interface IUSNFCustomResolver: IDispatch
{
  [id(1),..]          HRESULT  AttachSite([in] IUSNFContainer * pCTR);
  [propget, id(2),..] HRESULT  TargetID([out, retval] GUID *pVal);
  [id(3),..]          HRESULT  DoResolve([in] short IdSize,
                                         [in, size_is(IdSize)] BYTE * IdBytes,
                                         [out,retval] BSTR * pResolvedName);
  [id(4),..]          HRESULT  DoResolveString([in] BSTR ToBeResolved,
                                            [out,retval] BSTR * pResolvedName);
};
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Name	Parameters	Purpose
AttachSite	IUSNFContainer	This plugin has been attached to the Unsniff application. • save this interface pointer if you want to avail of services from the container later • perform any initialization
TargetID *get_TargetID*	GUID	What is your resolver target ID ? You can specify a built in target ID a custom resolver target ID.
DoResolve	Size (long) Bytes (BYTE*) BSTR	A binary array of length IdSize is given to you in IdBytes. You are expected to interpret this binary array and return a string representing the human readable resolved name. Example: If you are resolving IP Addresses the IdBytes array wiil contain 4 bytes representing the address and IdSize = 4
DoResolveString	BSTR (in) BSTR	This is same in functionality as DoResolve but gives you a text version of the raw name. You can interpret it and return a human readable name. Example: If you are resolving IP Addresses the ToBeResolved string wiill contain the string ip address (eg "192.168.87.22")

You do not have to support both the DoResolve and DoResolveString methods. Unsniff will automatically call DoResolveString if the DoResolve method returns E_NOTIMPL or another error.

IUSNFCustomResolverTarget

```
/*
 *  IUSNFCustomResolverTarget - A custom resolver target
 *      For example - A Site ID, Station ID, BSS Name, etc
 */
interface IUSNFCustomResolverTarget: IUnknown
{
  [helpstring("..")]  HRESULT GetTargetIID([out] GUID * pIid);
  [helpstring("..")]  HRESULT GetTargetName([out] BSTR * pVal);
  [helpstring("..")]  HRESULT GetTargetDescription([out] BSTR * pVal);
};
```

Unsniff Evaluation Copy

Methods

Name	Parameters	Purpose
GetTargetIID	GUID	A unique GUID for this resolver target. You can generate a unique GUID using the GUIDGEN tool.
GetTargetName	BSTR	A short name for the resolver target. Example : "Station ID"
GetTargetDescription	BSTR	A description of the resolver target.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

7.5 Custom User Object

A custom user object plugins implements the *IUSNFCustomUserObject* Interface. Each type of user object must have a unique GUID.

The IDL definition of this interface

```
/*
 *    IUSNFCustomUserObject - A custom user object
 *
 */
[
  object,
  uuid(95B58998-D224-4fdf-85C6-5E7BB631DED2),
  helpstring("IUSNFCustomUserObject Interface"),
  pointer_default(unique),
  local
]
interface IUSNFCustomUserObject : IUnknown
{
  [helpstring("..")] HRESULT GetUserObjectIID([out] GUID * pIid);
  [helpstring("..")] HRESULT GetShortName([out] BSTR * pVal);
  [helpstring("..")] HRESULT GetLongName([out] BSTR * pVal);
  [helpstring("..")] HRESULT GetDescription([out] BSTR * pVal);
};
```

Methods

Name	Parameters	Purpose
GetUserObjectIID	GUID	The unique GUID assigned to this type of user object.
GetShortName	BSTR	The user object name short
GetLongName	BSTR	A longer name for the user object type
GetDescription	BSTR	A detailed description of the user object type.

A custom user object plugins implements the *IUSNFCustomUserObject* Interface. Each type of user object must have a unique GUID.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

7.6 Custom User Object Renderer

As you saw in the previous section, you can define your own types of user objects (such as orders, transactions, video, etc). That provides you basic features like displaying and saving the user object to a file. If you want to do more with user objects, you have to create your own "*user object renderers* ". A renderer gives you complete control of how your user object is represented in Unsniff to the user. A custom user object renderer must implement the *IUSNFCustomUserObjectRenderer* interface.

The IDL definition of this interface

```
/*
 *  IUSNFUserObjectRenderer - A plugin that can render a user object
 *   Examples of user objects are : ImageViewer, Chat Viewer, Webbrowser etc.
 */
[
      object,
      uuid(D510F5B5-CCEF-4770-A3C0-12513EAAC2AD),
      helpstring("IUSNFUserObjectRenderer Interface"),
      pointer_default(unique),
      local
]
interface IUSNFUserObjectRenderer : IUnknown
{
      [..] HRESULT CanHandle([in] REFIID Oiid, [out] VARIANT_BOOL * pVal);
      [..] HRESULT Render([in] IUSNFUserObject * pVal,
                          [in] RECT * pRectClient,
                          [out] RECT * pRectImage,
                          [in] HDC hDC);
      [..] HRESULT GetObjectID([out] GUID * pVal);
      [..] HRESULT Play([in] IUSNFUserObject * pVal);
      [..] HRESULT Transform([in] IUSNFUserObject * pVal,
                             [out] IStream ** ppTransformedStream);
      [..] HRESULT SaveToFile([in] BSTR FileName,
                              [in] IUSNFUserObject * pSaveThis);
};
```

Methods

Name	Parameters	Purpose
CanHandle	GUID	Can you render this type of user object. The GUID identifies the user object type. Return VARIANT_TRUE or VARIANT_FALSE depending on your capabilities.
Render	UserObject RectClient RectImage [out] HDC	Render the user object passed to you. Use this method if it is possible to paint your user object to a DC. After you are done – return the dimensions of what you drew in the RectImage parameter. The HDC denotes the graphics context on which you must render. *Return E_NOTIMPL if this method is not applicable*
GetObjectID	GUID	What type of user object can you render ? Return the GUID of the user object type here.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Play	UserObject	Playback the user object. Implement this method if it makes sense to the type of user object you want to render. For example : It would make sense to play a "sound" but not to "play" a image. *Return E_NOTIMPL if this method is not applicable*
Transform	UserObject Istream	Apply a custom transformation on the user object. For example: you canuse this method to convert 'audio' userobjects to use linear PCM encoding. *Return E_NOTIMPL if you do not have a transformation*
SaveToFile	UserObject FileName	This method gives you a chance to save the user object is a custom format to a file. If you do not implement this method Unsniff will fill in a default SaveToFile. The default SaveToFile will save the raw contents of the user object to a file. You can use this method to save it your way. For example: You can save raw streaming audio to a WAV file or save a binary order to an XML file. *Return E_NOTIMPL if you are quite happy with the default SaveToFile mechanism*

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

7.7 User Interface Plugins

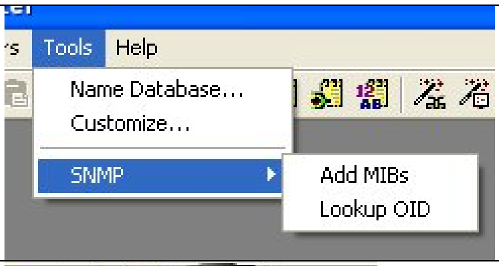
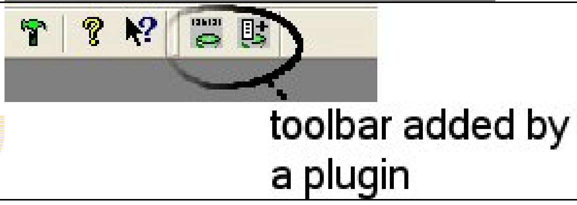
Samples located in folder "samples/plugins/ui_demo "

At some point your plugin is going to require some user interface elements. You may want to incorporate a dialog , a menu item or toolbar to allow the user to interact directly with your plugin. Unsniff allows a third party developer to directly integrate their application with the Unsniff user interface. You can merge your menu items and toolbars into Unsniff.

To write user interface plugins, you need to understand the concept of UI Host Site and the UI Customizer.

- The UI Host Site represents the Unsniff application. This application exposes a COM interface *IUSNFUIHostSite*. This interface has methods that let anyone add user interface elements.
- The UI Customizer represents your UI plugin. You will have to implement the *IUSNFUICustomizer* interface. You will be given a chance to add your user interface elements to the host site. The host site will then call you back whenever those user interface elements are activated.

Examples:

A plugin has added two menu items to the Unsniff menu	 Tools Help Name Database... Customize... SNMP Add MIBs Lookup OID
A plugin has added a toolbar to the Unsniff main toolbar	

Unsniff Ev... ...py

IUSNFUIHostSite IDL Definitions

Info

You do not have to implement the *IUSNFUIHostSite* interface. This interface is implemented by the Unsniff main application. You must call methods on this interface in order to add your user interface elements to Unsniff

```
/*
 *  IUSNFUIHostSite - hosting site (i.e) main application
 */
  [
    object,
    uuid(4F15361F-3131-45f0-9DDE-41B361C24200),
    helpstring("IUSNFUIHostSite  Interface"),
    pointer_default(unique),
    dual
  ]
```

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

```
interface IUSNFUIHostSite : IDispatch
{
  [id(1),  ..]  HRESULT  AddAppMenuItem([in] BSTR bsLocationString,
                               [in] BSTR bsFlyBy,
                               [in] BSTR bsTooltip,
                               [in] int MyResId);
  [id(2),  ..]  HRESULT  AddAppSeparator([in] BSTR bsLocationString);
  [id(5),  ..]  HRESULT  AddToolbar([in] UICONTEXT_T Ctx,
                           [in] HBITMAP hBitmap,
                           [in] int nButtonCount,
                           [out] ULONG * pToolbarID);
  [id(6),  ..]  HRESULT  AddToolbarButton([in] ULONG ToolbarID,
                                 [in] int Pos,
                                 [in] BSTR bsTooltip,
                                 [in] BSTR bsFlyBy,
                                 [in] int MyResId);
  [id(7),  ..]  HRESULT  ConfigFolderPath([out, retval] BSTR *pVal);
  [id(8),  ..]  HRESULT  PrintLogMessage([in] API_LOG_MESSAGE_LEVEL_T eLevel,
                                [in] BSTR msg);
  [id(9),  ..]  HRESULT  ActivateUI([in] BSTR bsVal);
  [id(10), ..]  HRESULT  PrintOutputMessage([in] BSTR bsVal);
  [id(11),  propget, ..]HRESULT InstallFolderPath([out, retval] BSTR *pVal);
  [id(12), ..]  HRESULT  GetActiveContainer([out,retval]IUSNFContainer **p);
};
```

Methods

Name	Parameters	Purpose
AddAppMenuItem	LocationString Flyby Tooltip Resource ID	Add a menu item to the location indicated by the bsLocationString parameter. **Location Strings** are simply a concatenation of where you want to place your menu item. Location strings start with an '&' character. *Example*: To place a menu item (Lookup OID) under Tools -> SNMP the location string is : *"&Tools\\SNMP\\Lookup OID"*[18]. If some items in the location string do not exist, Unsniff will automatically create sub-menu items for the missing items. Flyby: Flyby text Tooltip: Tooltip text Resource ID: You must assign a resource ID to each item. When this item is selected you will be notified with this resource ID.
AddAppSeparator	LocationString	Add a separator at the Location String.
AddToolbar	UIContext Bitmap Num Buttons Toolbar ID [out]	Add a toolbar to Unsniff. *UIContext*: Specify UICTX_APPLICATION *Bitmap*: A strip of 16x16 images. *Num Buttons* : Maximum number of buttons on the toolbar *Handle(ToolbarID)*: The UI Host site will create the toolbar and return a handle. Save this handle, you must use this handle to issue AddToolbarButton commands.

Unsniff Evaluation Copy

[18] Note the double backslashes "\\". The first backslash is just an escape character for the next one. Location strings are simply menu items separated backslashes.

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

AddToolbarButton	Toolbar ID Position Tooltip Flyby Resource ID	Add a toolbar button (command) to a toolbar identified by ToolbarID. *Toolbar ID*: The handle returned by a previous AddToolbar call *Position* : Zero based position of this button in the toolbar bitmap *Tooltip* : Tooltip text for this button *Flyby* : Flyby text for this button *Resource ID*: A user specified command id you attach to this button. When the user presses this toolbar button, the UI Host will call the OnAction method with this Resource ID.
ConfigFolderPath (get_)	String	Get the configuraton folder path where you can store you custom specific data. Your plugin can store its data anywhere on the system you like.
PrintLogMessage	LEVEL Message	Print a message to the Log Window. You can alert the user about error conditions via the log window. This way your plugin can integrate its error reporting into Unsniff. *Level* : The error level `L_CRITICAL, L_MAJOR, L_MINOR, L_INFO` *Message*: You error message in a BSTR
PrintOutputMessage	Message	Print a message to the Output Window. Use this if you plugin performs some batch tasks like compiling, scanning files. *Message*: Your output message in a BSTR
ActivateUI	UI ElementName	Activate the specified Unsniff UI component if possible.This makes the selected UI element visible. *UI Element Name*: Currently "Log Window" or "Output Window".
GetActiveContainer	IUSNFContainer	The currently active capture window. Use this to get a handle into the contents of the current capture window. **Warning** Do not cache the return value of *GetActiveContainer*. Since Unsniff can handle multiple documents at the same time, the current active container may changed or closed at any time by the user. Instead call *GetActiveContainer* each time you want to access the current capture file.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

IUSNFUICustomizer IDL Definitions

```
/*
 *  IUSNFUICustomizer - a plugin needs to implement this interface
 *    if it wants to add UI Elements (menus, toolbars, windows) to the main
 *    container
 */
  [
    object,
    uuid(3146BBA7-EBB3-484b-877D-00D7894BA1BB),
    helpstring("IUSNFUICustomizer Interface"),
    pointer_default(unique),
    dual
  ]
  interface IUSNFUICustomizer : IDispatch
  {
    [id(1), ..]  HRESULT AttachSite( [in] IUSNFUIHostSite * pHost);
    [id(2), ..]  HRESULT OnAction(   [in] IUSNFUIHostSite * pHost,
                                     [in] int  MyResId);
  };
```

Name	Parameters	Purpose
AttachSite	HostSite	Your plugin is now being attached to a UI Host Site. You must call methods on the supplied *IUSNFUIHostSite* interface to add menu items, toolbar buttons, etc (see the previous section on *IUSNFUIHostSite*)
OnAction	HostSite MyResID	A toolbar button or menu item with the given MyResID was pressed. You can now take action.

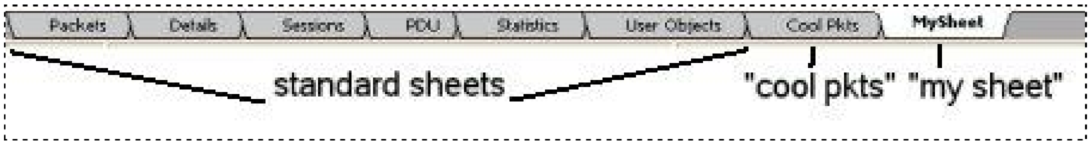
Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

7.8 Custom Sheets

Samples located in folder "samples/plugins/ui_demo "

As far as Unsniff plugins are concerned, Custom Sheets represent the top of the mountain. Custom Sheets allow you to extend the Unsniff interface by integrating your own windows as sheets in the capture window. A screenshot of two custom sheets are shown below. Notice how they integrate into the Unsniff capture window. You can also see that the first six standard sheets that are built into Unsniff.



Custom Sheets

- Custom Sheets are ActiveX controls. We recommend using ATL and WTL to write your custom sheets. They are lightweight and extremely powerful libraries without compromising on performance.

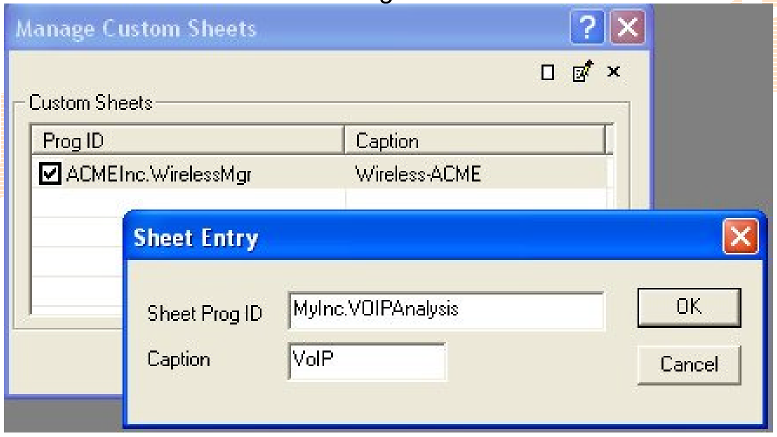
- Implement your custom sheet as an Eavesdropper (implement the *IUSNFEavesdrop* interface). This way you can get in on the action happening in Unsniff as each packet is processed. You can update your sheet when interesting things happen.

Installing Custom Sheets

Unlike other types of plugins custom sheets must be installed manually. Each custom sheet is an ActiveX control. You must enter the ProgID of the control and a caption for the sheet in Unsniff.

- Select "Plugins" → "Sheets" from the main Unsniff menu

- The Sheets dialog is displayed. Select the "New" icon from the top-right corner. You will be shown the New Sheets Dialog



Prog ID	Caption
☑ ACMEInc.WirelessMgr	Wireless-ACME

Sheet Prog ID: MyInc.VOIPAnalysis
Caption: VoIP

- Enter the ProgID of your Sheet and a short caption. The caption will be shown in the sheets tab. Try to keep the caption to less than 12 characters.

- Press OK

<End of Task>

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Evaluation Copy

Appendix A – API Error Codes

Whenever Unsniff detects an abnormal situation such as security violation, incorrect API call, undefined or errored fields, it will generate an error message. This message can be seen in the Unsniff Log Window provided the correct API Trace Level is set (Sec 2.x). When an API Call returns FALSE, you can get further information about the error code by called *UserGetLastError()*.

Critical Errors:
A critical error is one that prevents your entire plugin from working. This is an indication of a bug in your plugin or in the Unsniff application itself.

To check if an error code is a critical error code use the *IS_CRITICAL_ERROR(x)* macro.

Code	Id	Description
UAPIERR_API_NOT_INIT	0x80040204	You have not yet called *UserInitAPI*(). You should not encounter this error if you use the wizard.
UAPIERR_FRAME_PREMATURE_END	0x80040205	The frame was cut short unexpectedly. This indicates a serious system error.
UAPIERR_INVALID_MODULE	0x80040206	The plugin DLL module is invalid. This indicates a linkage problem with the DLL. You may have to recompile your plugin
UAPIERR_OVERSHOT_FRAME	0x80040207	You tried to refer to memory beyond or below the frame. The frame refers to the raw packet data for your layer.
UAPIERR_NO_ID	0x80040208	This plugin has no identifcation information. For C++ plugins you must call *UserInitID()* , for XML plugins you must initialize the plugin attributes.

Unsniff Evaluation Copy

Major Errors:
A major error is one that affects the current packet or PDU – resulting in a partial or no decode at all. Other portions of your plugin may continue to work satisfactorily.

To check if an error code is a major error code use the *IS_MAJOR_ERROR(x)* macro.

Code	Id	Description
UAPIERR_FIELD_INVALID_SET	0x80040301	You are trying to set an incompatible value to a field.
UAPIERR_FIELD_TOO_BIG	0x80040302	This field is too big. The maximum size of a field is 64k bytes
UAPIERR_FIELD_UNDEFINED	0x80040303	The field identified by the integer FieldID or the string field name is not defined. You must first define the field in XML or via *UserAddFieldDef*().
UAPIERR_FIELD_MISALIGNED	0x80040304	This is a sign of a buggy plugin. Usually protocols make sure that fields are aligned on their natural

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

		boundaries. This error is generated when you try to push a field onto the field stack when the frame is at the wrong alignment. Example : when you try to push a 32 bit integer field when the frame pointer is at an odd offset
UAPIERR_NO_GUARD	0x80040305	System Error.
UAPIERR_BAD_ID	0x80040306	The ID supplied to the field is bad.
UAPIERR_FIELD_TOO_SMALL	0x80040307	The field is too small. The minimum size of a top level field is 4 bits (a nibble). You can have 1-bit fields only as members of a bit-field.
UAPIERR_DUPLICATE_FIELD	0x80040308	This field is a duplicate. This happens when you try to define a field using the same *Field Name* or *ID*.
UAPIERR_DUPLICATE_ENUM	0x80040309	The enumeration has a duplicate value.
UAPIERR_UNSORTED_ENUM	0x80040310	The enumeration list is not sorted. This is a performance warning. Please sort your enums using a tool like Excel or Word.
UAPIERR_INVALID_STYLE	0x80040311	This style specified is invalid.
UAPIERR_FIELD_NOT_FOUND	0x80040312	The field was not found in the list of pre-defined fields or in the list of fields currently pushed onto the field stack.
UAPIERR_FIELD_NOT_DEFINED	0x80040313	This field has not been defined. Please define using XML <FieldDef> or using the C++ *UserAddFieldDef*() method
UAPIERR_FIELD_NOT_ALIGNED	0x80040314	Same as FIELD_MISALIGNED
UAPIERR_RECORD_TOO_DEEP	0x80040315	The record nesting is too deep. Unsniff allows a nesting level of five levels. Try to redefine your fields using less nesting.
UAPIERR_NO_RECORD	0x80040316	There is no such record
UAPIERR_NO_END_RECORD	0x80040317	No matching end record was found. This happens when you begin a record and forget to end it before returning from *BreakoutFields()*
UAPIERR_INVALID_FLAGS	0x80040318	The flags field is invalid. See error message for more detail.
UAPIERR_INVALID_PARAMETER	0x80040319	The parameter was invalid See error message for more detail
UAPIERR_BAD_TYPE	0x80040320	The field type is unexpected or bad.
UAPIERR_INVALID_RESOURCE	0x80040321	The resource specified (icon or field image) is bad
UAPIERR_MAJOR_ERROR	0x80040322	System error.
UAPIERR_INVALID_LAYER	0x80040323	System error.
UAPIERR_PARSE_ERROR	0x80040324	There was an error in *UserInitQP*(). There may be other error messages that appear along with this that offer more detail.
UAPIERR_WRONG_PROCESS	0x80040325	You are calling a method in an incorrect context. For example: you are expected to push fields onto the

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

		field stack only in the BreakoutFields() method. If you try to push fields anywhere else you will get this error. This check is applied for all API methods to ensure that your plugin is well written.
UAPIERR_WRONG_FIELD_TYPE	0x80040326	Your API call failed because you applied it to a wrong field type.
UAPIERR_UNKNOWN_CONFIG_TYPE	0x80040327	The configuration parameter is unknown.
UAPIERR_UNKNOWN_STREAM_OPER	0x80040328	The stream operation is unknown.
UAPIERR_ASN_VALUE_NOT_SET	0x80040329	The ASN field is not yet on the field stack. As a result its value is not set.
UAPIERR_INVALID_XML_URI	0x80040330	The XML Uri is invalid (cannot be found)
UAPIERR_XML_SCHEMA_ERROR	0x80040331	The XML protocol plugin file violates the Unsniff XML Schema. Other messages are generated along with this that identify the exact places where the schema is violated.
UAPIERR_BREAKOUT_ERROR	0x80040332	The field breakout process had an error. Other messages are generated that identify the exact sources of error.
UAPIERR_XML_DELAYLOAD_ERROR	0x80040333	The delay load process failed for the XML document. As a result all fields in the requested delay load <FieldDefs> are not available.
UAPIERR_VALUE_NOT_SET	0x80040334	The API call failed because the value of the field is not yet set. This could be because the field is not yet on the Field Stack.
UAPIERR_INVALID_VARIABLE	0x80040335	The variable name is invalid. Only alpha numeric strings beginning with an alphabet are allowed
UAPIERR_VARIABLE_NOTFOUND	0x80040336	The variable cannot be found because either: ▪ It was not defined ▪ The field attached to the variable has not yet been pushed on to the field stack
UAPIERR_INVALID_FIELD	0x80040337	The field will become invalid due to the current action. See the error message for the field name and how exactly it became invalid.
UAPIERR_ASN_TAG_ERROR	0x80040338	There was a tagging error in your ASN.1 field. A common mistake is to specify explicit or implicit ASN.1 tags only for some fields for a ASN SEQUENCE or SET. ▪ You must specify tags for all fields of a SEQUENCE / SET ▪ You cannot mix and match implicit and explicit tags
UAPIERR_INF_LOOP_DETECTED	0x80040339	The Unsniff API detected an infinite

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

		loop in your breakout logic. This ability of Unsniff prevents many crashes or hung application situations. This error message gives you a graceful chance to fix your bug.
UAPIERR_XML_PLUGIN_ERROR	0x80040340	The XML Plugin could not be installed into Unsniff due to an error. This error message is accompanied by other *system* error messages that specify the exact problem.

Minor Errors:
A minor error is one that does not affect the current packet – but it may result in an inefficient decode or sometimes incorrect decode. Other portions of your plugin may continue to work satisfactorily.

To check if an error code is a minor error code use the *IS_MINOR_ERROR(x)* macro.

Code	Id	Description
UAPIERR_PERF_WARNING	0x80040401	Performance warning
UAPIERR_FRAME_NOT_FULL	0x80040402	You did not account for the entire frame. This happens when your Field Stack does account for every byte in the frame
UAPIERR_TRUNCATED	0x80040403	A string was truncated because it was too big.
UAPIERR_NOT_IMPL	0x80040404	The functionality requested has not yet been implemented
UAPIERR_EMPTY_BREAKOUT	0x80040405	The field stack is empty. You did not account for any fields in the frame.
UAPIERR_EMPTY_CONFIG	0x80040406	The configuration block is empty.
UAPIERR_DUPLICATE_CONFIG	0x80040407	The configuration item is a duplicate. Only one will be honored.
UAPIERR_UNKNOWN_CONFIG	0x80040408	The configuration item is unknown. Please specify the item in the *ProvideConfigDefs*() method
UAPIERR_UNKNOWN_KEY	0x80040409	The accounting key in unknown, define the key in *ProvideAcctDefs*() method
UAPIERR_ASN_LENGTH_ERROR	0x80040410	The ASN.1 field length had an error. See the error message for details.
UAPIERR_ALIGNMENT	0x80040411	The API call failed to an alignment error.
UAPIERR_BUFFER	0x80040412	The string operation failed because the buffer supplied was too small
UAPIERR_XML_SCHEMA_WARNING	0x80040413	The XML schema has some incorrectly defined elements. This is just a warning. You may want to fix this warning to produce a clean XML document.
UAPIERR_UNSUPPORTED	0x80040414	The operation on the field is unsupported
UAPIERR_DUPLICATE_CHOICE	0x80040415	There is an ambiguous choice field.

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

		This case is similar to having a C++ switch statement with duplicate case values.
UAPIERR_SIZE_MISMATCH	0x80040416	There was a size mismatch between two fields
UAPIERR_INVALID_OPERATION	0x80040417	The operation is invalid in the current state of the API

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

Appendix B – XML Schema

This section contains the schema of the Unsniff XML Specification.

Info
This XML Schema must be used for basic validation only. It does not address complex elements like field styles or required attributes for some field types. We recommend that you use this schema for basic validation and use Unsniff for advanced validation. To use Unsniff, set the log warning level to INFO and observe the log window. All schema errors and warnings will appear in the log window.

```
<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="UsnfProtocol" type="UsnfProtocolType" />
  <xsd:complexType name="UsnfProtocolType">
    <xsd:sequence>
      <xsd:element name="vendor" type="xsd:string" />
      <xsd:element name="conformance" type="xsd:string" />
      <xsd:element name="color" type="xsd:string" />
      <xsd:element name="icon" type="xsd:string" />
      <xsd:element name="version" type="xsd:decimal" />
      <xsd:element name="rootfield" type="xsd:string" />
      <xsd:element name="FieldDefs" type="FieldDefsType" />
      <xsd:element name="DescriptionString" type="DescriptionStringType" />
      <xsd:element name="AccessPoints" type="AccessPointsType" />
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" />
    <xsd:attribute name="shortname" type="xsd:string" />
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:attribute name="protid" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="AccessPointsType">
    <xsd:sequence>
      <xsd:element name="AccessPoint" type="AccessPointType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="AccessPointType">
    <xsd:attribute name="hostid" type="xsd:string" />
    <xsd:attribute name="apvalue" type="xsd:int" />
  </xsd:complexType>
  <xsd:complexType name="DescriptionStringType">
    <xsd:sequence>
      <xsd:element name="format" type="xsd:string" />
      <xsd:element name="Params" type="ParamsType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ParamsType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="Param" type="ParamType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ParamType">
    <xsd:attribute name="ref" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="FieldDefsType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="FieldDef" type="FieldDefType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="FieldDefType">
    <xsd:sequence>
      <xsd:element name="fieldtype" type="xsd:string" />
      <xsd:element name="sizebits" type="xsd:int" />
```

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved

```
      <xsd:element name="styles" type="xsd:string" />
      <xsd:element name="helptext" type="xsd:string" />
      <xsd:element name="variable" type="xsd:string" />
      <xsd:element name="EnumList" type="EnumListType" />
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" />
  </xsd:complexType>
  <xsd:complexType name="EnumListType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="enum" type="enumType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="enumType">
    <xsd:attribute name="value" type="xsd:int" />
    <xsd:attribute name="name" type="xsd:string" />
  </xsd:complexType>
</xsd:schema>
```

<<END>>

Unsniff Evaluation Copy

Copyright © Unleash Networks P. Ltd, 2003-05, All rights reserved