CANAPE Workshop

BSides Munich
2018

Hi everybody!
I am Hans-Martin

I work as a Senior Security Analyst with a focus on offensive security and vulnerability research.

If you dig deep in the internet you can find videos of me, headspinning ☺

About this workshop and slides

I learned about CANAPE during a penetration test of a client-/server application that used a proprietary protocol. CANAPE was the perfect tool for that task, however I had several issues mainly due the minimal/missing documentation.

This workshop is designed as a „hands on“ course for an audience that has a basic understanding about binary protocols, but never worked with CANAPE before. It is based on James Forshaws workshop at 44con workshop from 2014.

I didn’t attend that workshop, nor would I say that I know all aspects/tweaks in CANAPE but I hope I can save you some time with this introduction. Feel free to ask questions, if I can’t answer them now I will get back to you by mail.

These slides are not primarily designed as a presentation, but as a reference for later use, so that everyone could use them to get started with CANAPE.

Saved states

We will do different small workshop projects that show you different CANAPE features on a small demo application (Super Funky Chat).

During the workshop preparation, I created a backup of the CANAPE project that represents the state after completion of the subproject. You can use these files if you got lost or something else broke to continue. Just to make sure that you don’t get lost ☺

Todays agenda

Cause everyone needs a plan…

1. Introduction
2. Proxying and capturing traffic
3. Net Graphs
4. Packet logging and tools
5. Parser development
6. Working with layers
7. State Modeling
8. Traffic replay/manipulation
9. Creating network clients

1.

Introduction

Say „hello“ to CANAPE
(and Super Funky Chat)

What is CANAPE?

CANAPE is a capture and manipulation tool for arbitrary network protocols. It was developed by James Forshaw during his time at Context IS.

Simplified, CANAPE can be described as “Burp for binary protocols” ☺

When it comes to HTTP traffic, most security researchers use a man in the middle proxy like Burp or OWASP Zap to intercept and manipulate http requests/responses.

If you have to deal with a binary protocol, the situation is different. There is no “one for all” tool. Researchers uses different tools or small scripts to capture, manipulate or fuzz network traffic.

CANAPE takes the well known web application testing paradigm and applies it to arbitrary protocols. It provides a GUI that allows an easy manipulation of network packets, which speeds up the entire process.

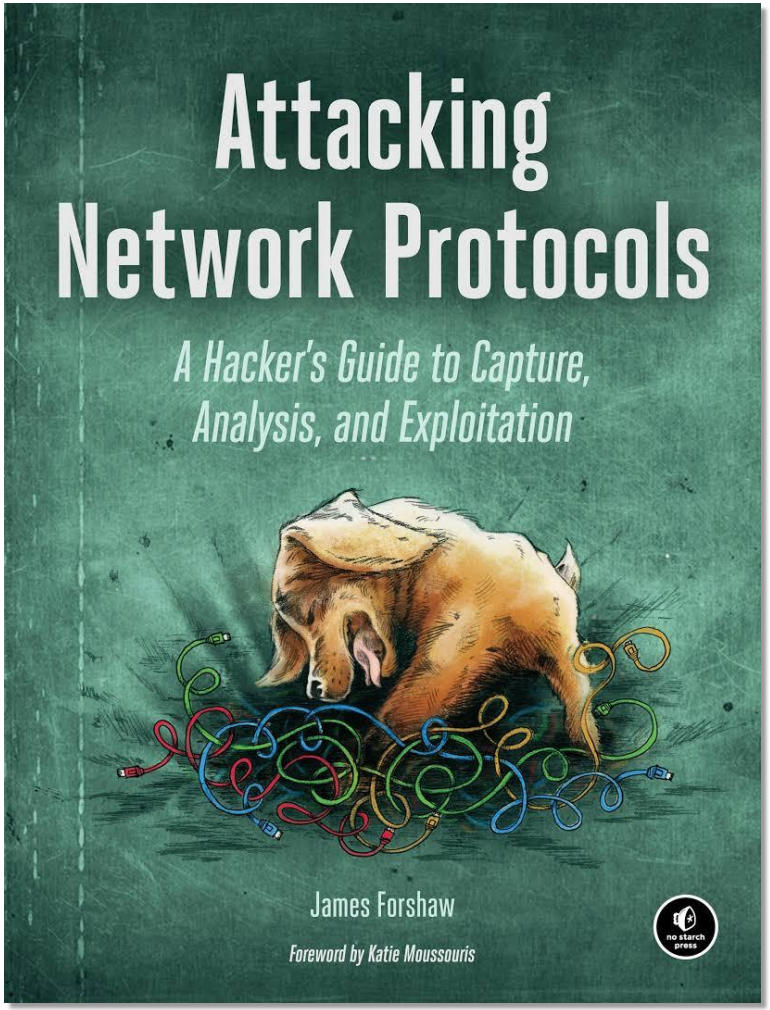
CANAPE is written in C#. While it provides MONO support, the main target platform is Microsoft Windows. CANAPE is OpenSource, you can get the code at GitHub.

James Forshaw also developed a network library for .NET core (CANAPE.Core) but this does not include a GUI.

Attacking Network Protocols

James Forshaw also wrote a book about this topic. It is not a CANAPE user guide, but provides a great introduction to “attacking network protocols” ☺

James helped me tons during the preparation of this workshop! Thanks for that!

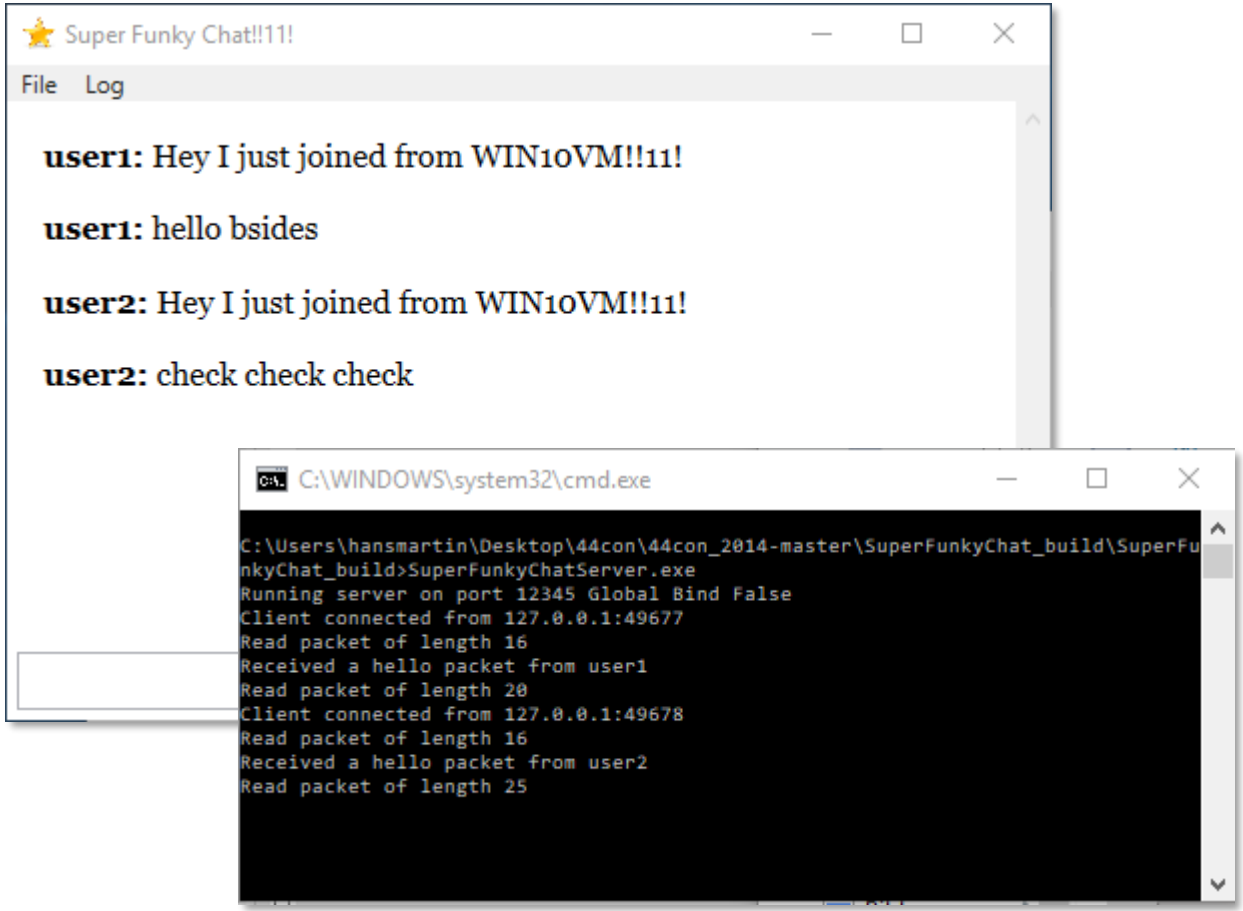


Super Funky Chat

Super Funky Chat is a small example application that was developed by James Forshaw for his 44con workshop.

It is a simple IM application that uses a binary protocol and has some awesome security features, making it a perfect target for the workshop.

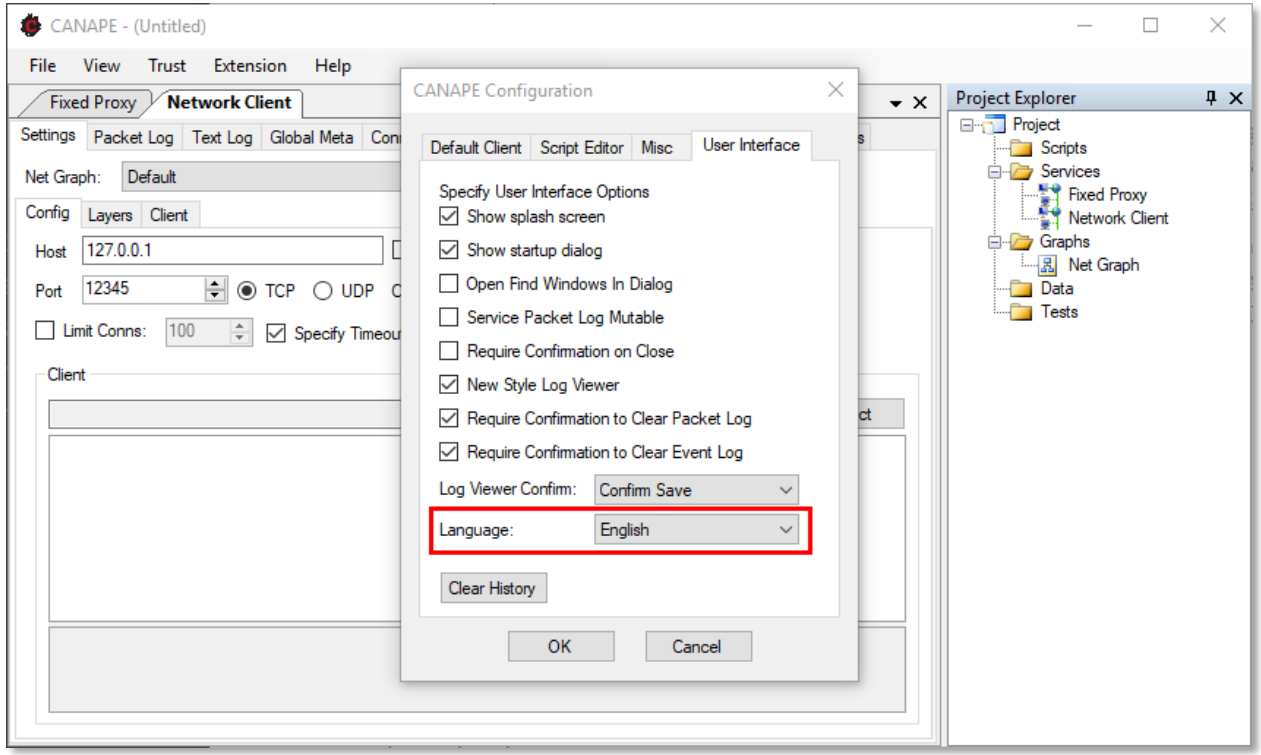
There is also a .NET Core version.



Adjust language

CANAPE supports multiple languages (German/ English), however the German translation is incomplete and sometimes misleading.

By default CANAPE selects the language based on your OS settings, however you can change that in the CANAPE configuration.



CANAPE components

A CANAPE project consists of multiple components that can be accessed via the project explorer on the right side of the screen.

Services

Network services allow you to capture, receive or send traffic in CANAPE

A project can have multiple services, for example if the target uses different protocols, ports.

Graphs

CANAPE uses graphs to model the data flow and network state of a protocol.

Depending on the protocol, graphs can become quite complex and have multiple layers.

Scripts

Dynamic scripts (Python, C#) that help you to parse or modify the traffic. Protocol parsers are also located here

These scripts/parsers are later called by other components, for example graph nodes.

Data

The Data section contains static data that can be used by other components, for example network clients.

You can store arbitrary data here, but we use it mainly use it for network packet captures

Tests

CANAPE allows you to create simple test cases for your scripts and parsers. This speeds up parser development.


Workshop 1:
Let‘s have a first look at CANAPE
and SuperFunkyChat

2.

POXYING AND CAPTURING TRAFFIC

Getting data into CANAPE

Service components

You basically start your CANAPE project with one or more services.

The service provides a template for:

- Data flow
- State model
- Dynamic content

When a new connection is made, a new instance is instantiated based on that template.

You define network services to get network data into CANAPE or to communicate with external target services. By default, CANAPE provides the following service types, but you can add services for other networks like NamedPipes or COM ports.

Proxies

Proxy services intercept the client/server communcation and are the easiest way to get network packets into CANAPE.

CANAPE provides a variaty of proxy services, from simple packet forwarders to HTTP or SOCKSv4/v5 proxies.

Please note that CANAPE only works on the appli-cation level (TCP/UDP) and doesn‘t perform attacks like ARP spoofing or DNS poisioning to intercept traffic.

Servers

You can develop a server endpoint inside CANAPE. The server implementation can be used to evaluate the security of network clients.

CANAPE provides several server templates, for example a simple echo server or a HTTP endpoint.

.

Clients

Clients are used to test network services.

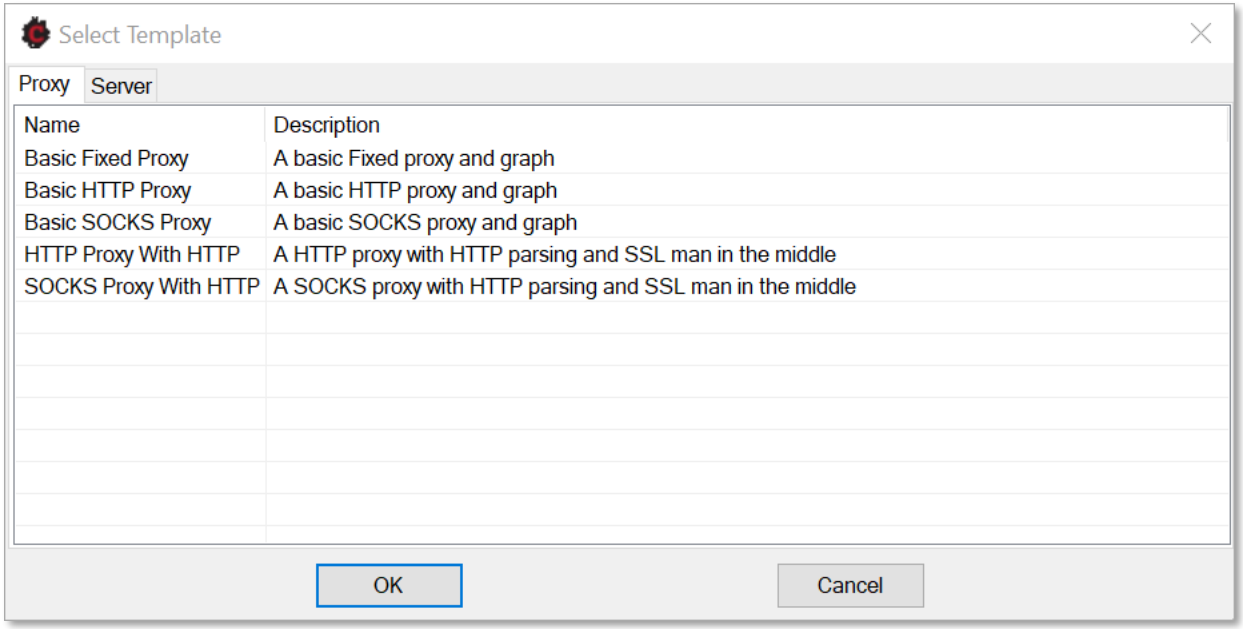
You can generate basic clients from previously captured or imported network traffic and manipulate the data via Python/C# scripts.

CANAPE templates

CANAPE provides templates for common proxies or server services.

A fixed proxy is ideal for applications that are not “proxy aware”.

CANAPE also supports HTTP/SOCKS proxies. You can also forward outgoing data to a proxy service.

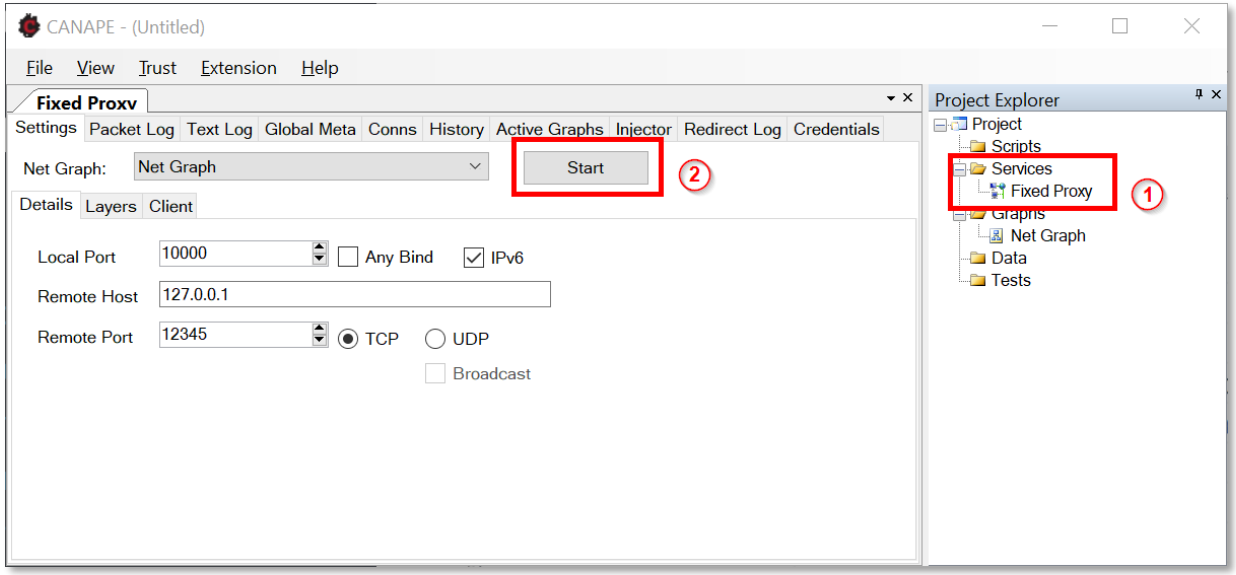


Capture network traffic

Define a network service (right click on the services tap) and click on the start button to start the service.

You will also need to define a Net Graph, but we will deal with that later.

Please don't forget that you need to modify the client environment to redirect network traffic through the CANAPE services.



Workshop 2: Capturing network traffic

3.

NET GRAPHS

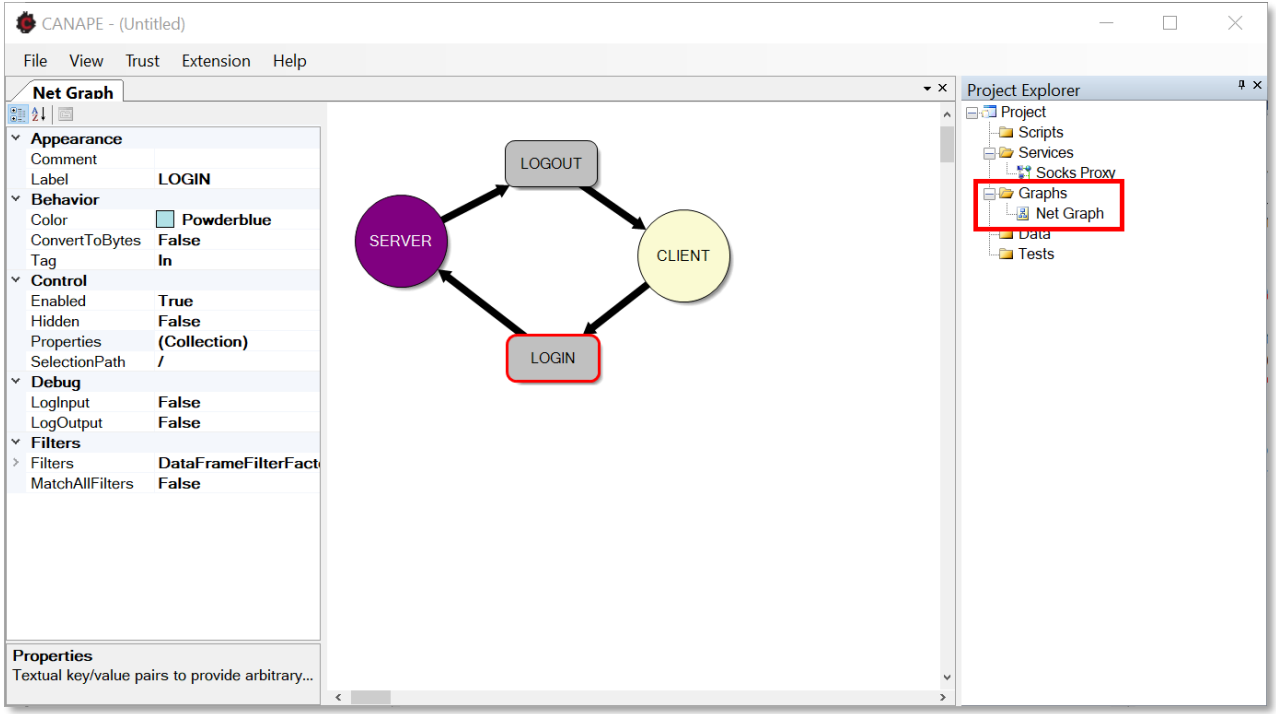
„Drawing“ data flows

Net Graphs

CANAPE uses Net Graphs to describe the data as a directed graph.

You can access Net Graphs in the “Graphs” section of the CANAPE project explorer.

Net Graphs can become quite complex as they might have multiple nodes/layers, depending on the target protocol.

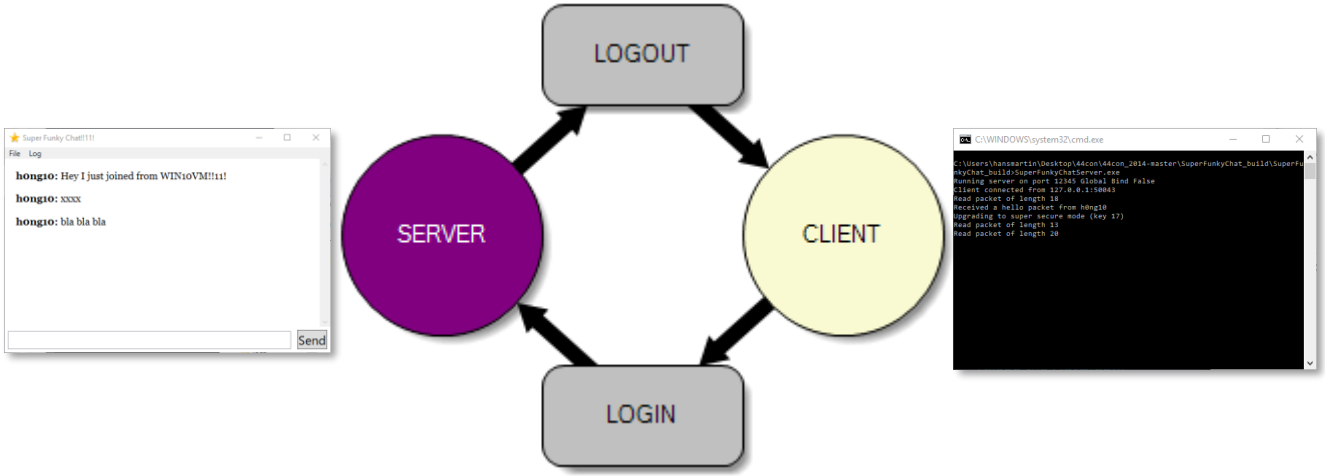


Net Graphs in Service Contexts

The node names “SERVER” and “CLIENT” might be misleading, but that depends on the viewpoint.

It helps to say:

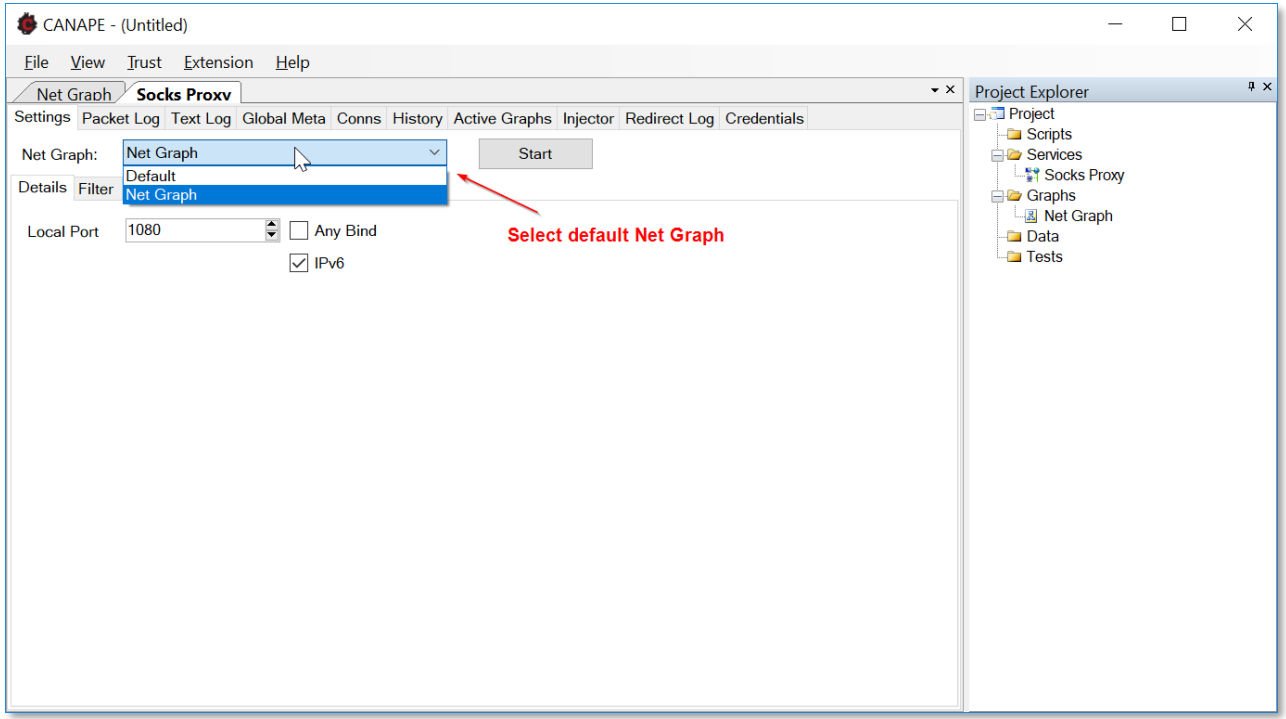
“The client data enters CANAPE at the proxy-SERVER and leaves it through the customized CLIENT.”



Specifying Net Graphs

Every network service is connected to a Net Graph. You can select the Net Graph to use in the service settings.

Hint:
If you change something in the NetGraph, for example adding a new node, you must restart the service to reload the NetGraph.



Graph Editor commands

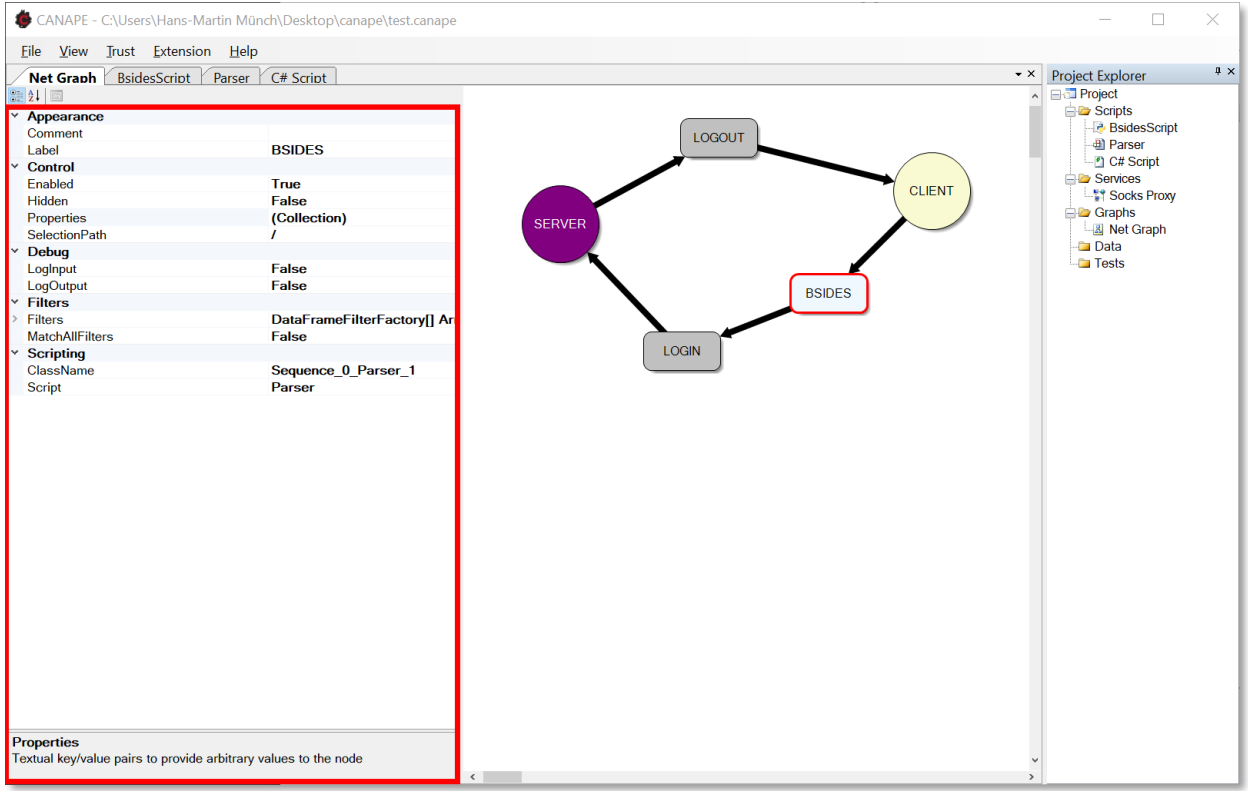
The graph editor allows you to easily modify Net Graphs via drag and drop. You only need to remember the following commands.

To...	Use the...
...select and drag nodes	left mouse button.
..add a new node	right mouse button. You can select the node type from the drop down menu
..draw a line between nodes	**left mouse button + left CTRL key** while pointing on a existing node

Node properties

The settings for each node can be edited in the properties editor on the left.

Most properties are quite self explaining. The description at the bottom often helps ☺



Standard Nodes

When a network packet traverses the Net Graph, it passes one or more nodes.

By connecting multiple nodes, you can inspect or manipulate a packet without writing a single line of code.

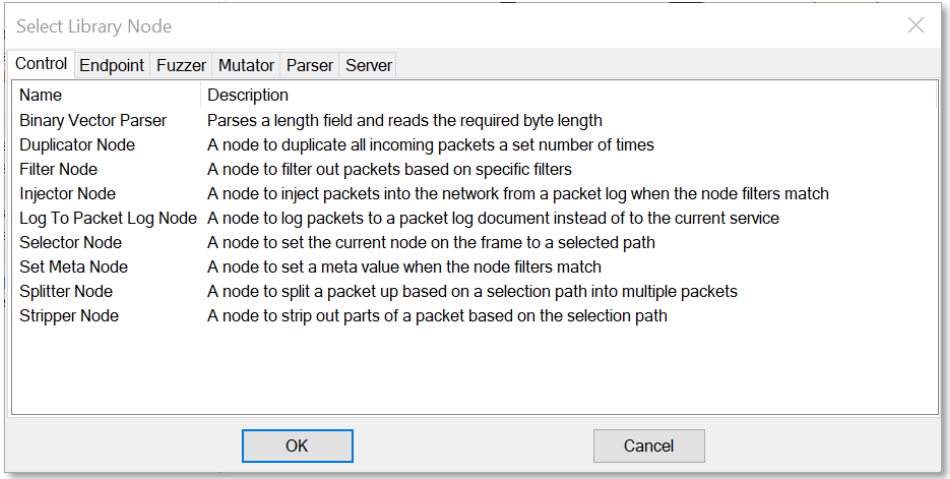
CANAPE provides multiple standard nodes for common tasks like logging, replacing data etc.

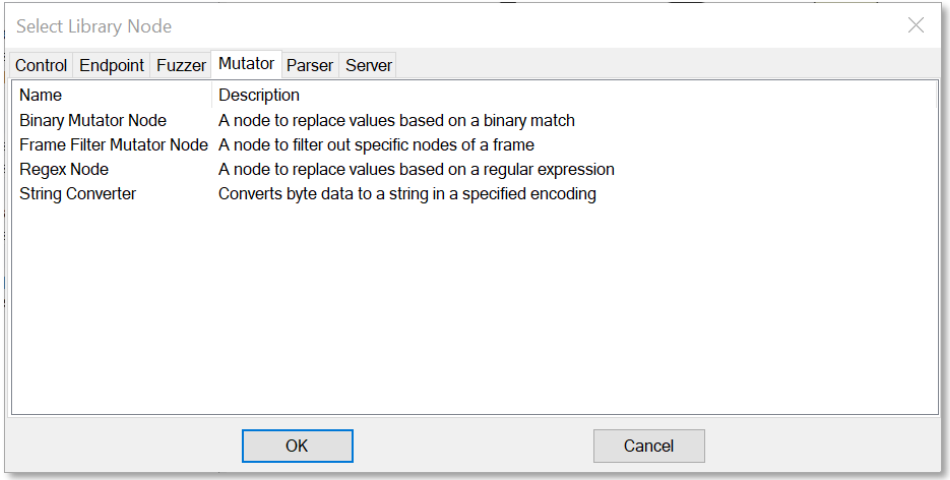
Node Name	Description
Log Node	Logs packets as they traverse the node
Descicion Node	Perform an IF condition on packets.
Switch Node	Select an output based on state
Edit Node	Displays a dialog to allow manual packet editing
Dynamic Node	Run a Python/C# script with the network packet as input
Layer Section	Graph container (Inception!!!) and layer processing

Library node

The library node is a special node that provides access to several other useful nodes.

For example, you can use a “regex node” to search and replace values in the current packet without writing a single line of code.



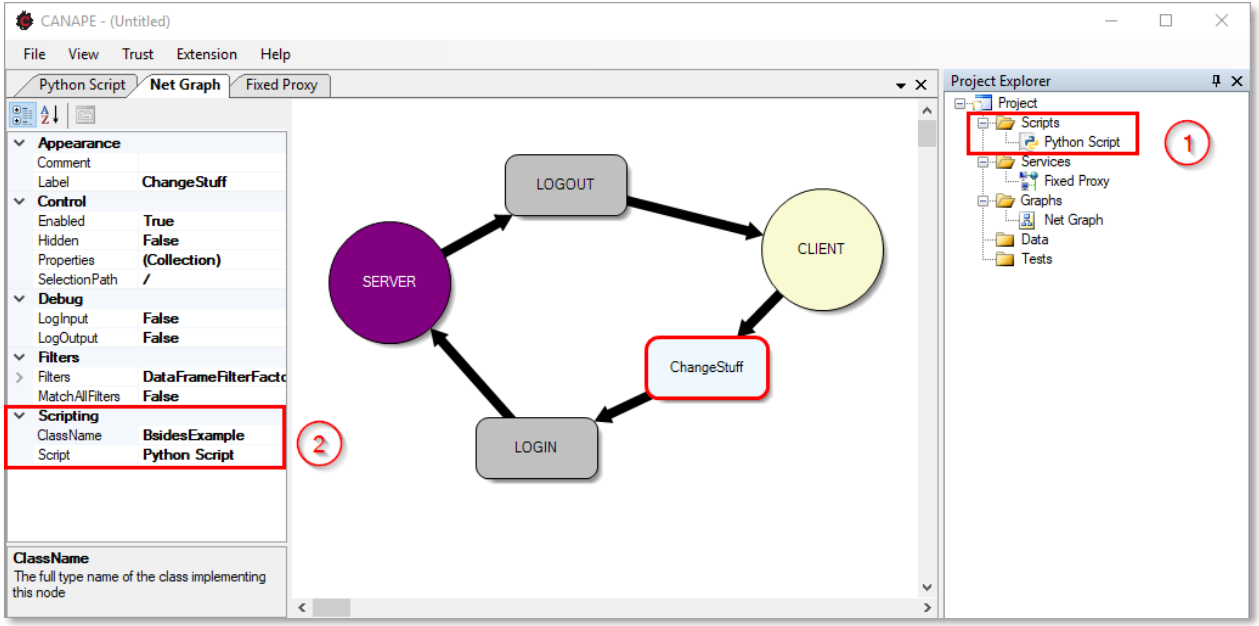


Dynamic node

A “Dynamic node” allows you to run your own scripts with the network packet as input. This functionality is also used in the background by network parsers (more on that later).

You define your scripts in the script folder (1) and then select the script/class in the node properties (2).

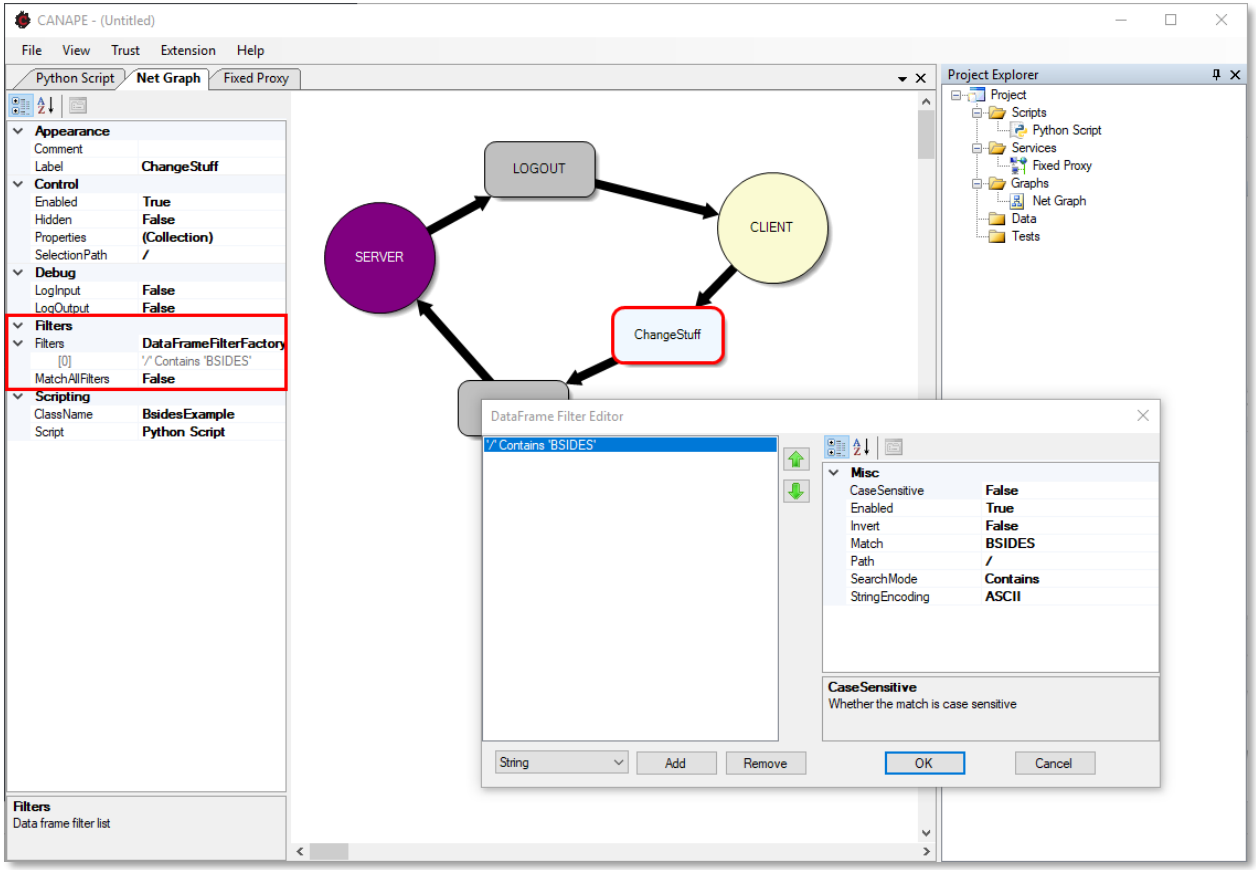
We will use this function later to parse network packets.



Node filters

Every node has a “filters” property that allow you to define match rules for what packets the node will process.

You can define multiple filters, each match will trigger the processing on the node. To force a match on all filters, set the “MatchAllFilters” property to “True”



Workshop 3: Playing with Net Graphs

4.

PACKET LOGGING/ TOOLS

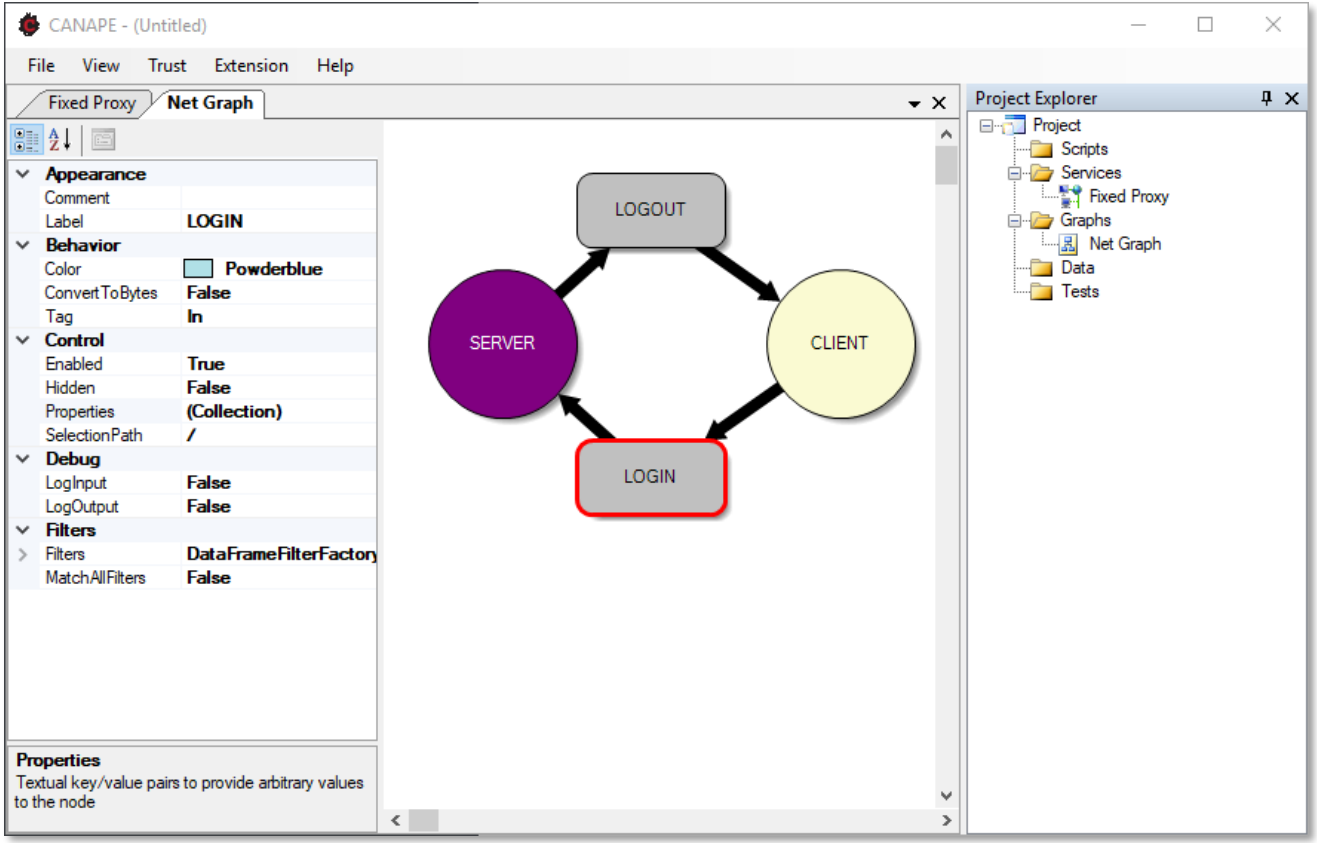
Lets look at some data

Logging packets

Only packets that traverse a log node in a graph end up being logged. The default net graph contains to log nodes for both directions (LOGIN and LOGOUT).

Like for every node, you can set filters to define which packets should be logged.

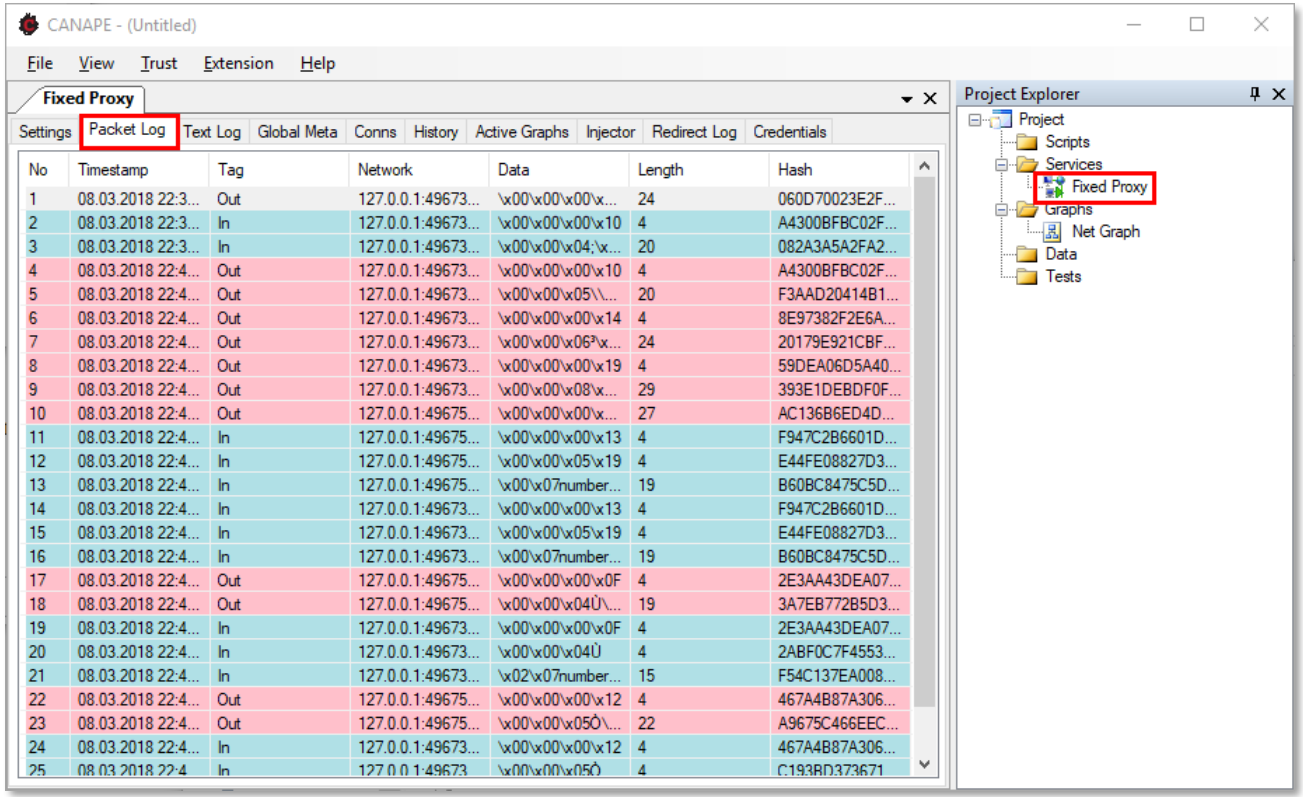
You can also set additional properties like the color or a textual name. This helps you to identify specific packets more easily.



Traffic log

To access the logged traffic, use the “Packet log” tab in the selected service.

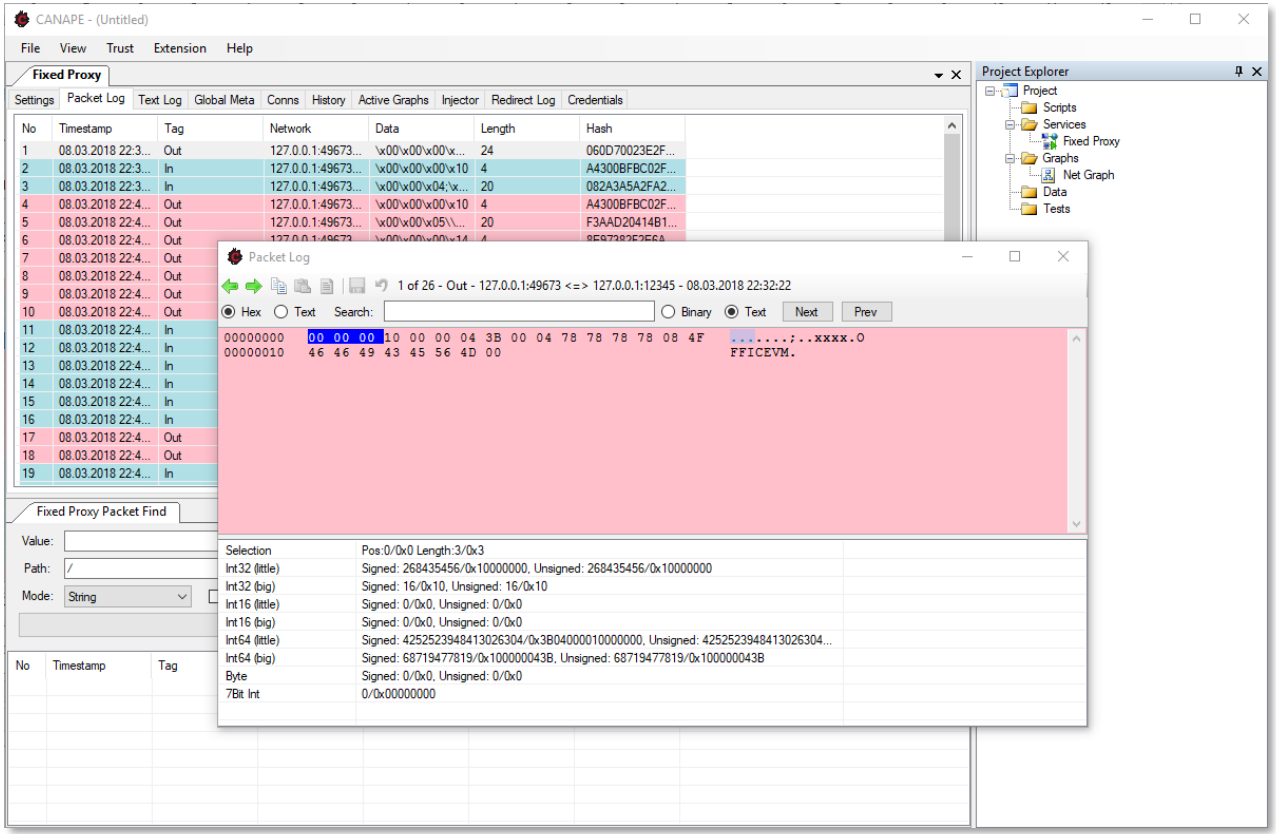
The different colors and tags allow an easy identification of the traffic origin.



Detail view

Clicking on a entry opens the detail view.

The table at the bottom shows you different interpretations of the selected data.

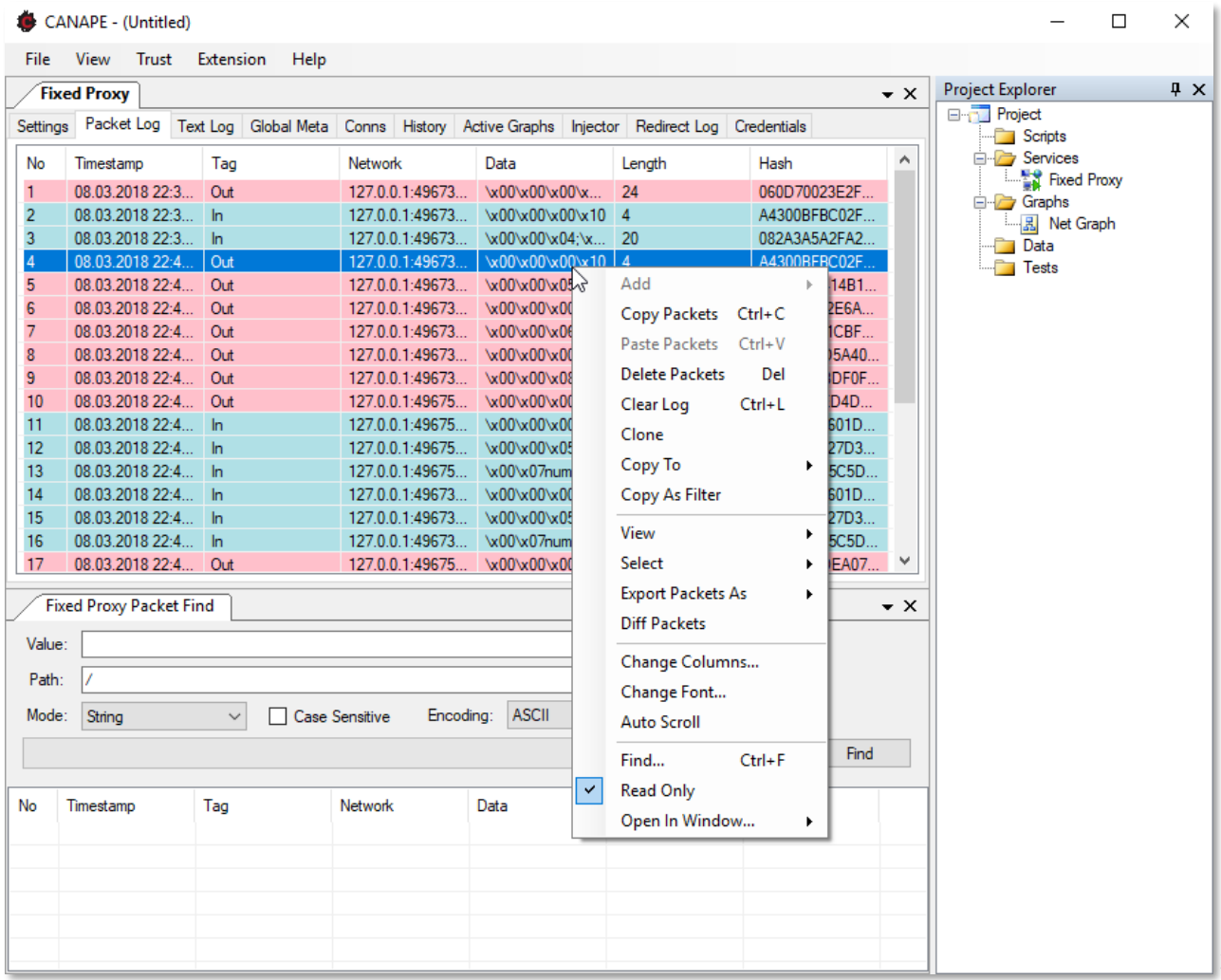


Packet tools

The packet tools help you to analyze packets. You can access them from the context menu.

The packet tools allow you to:

- Compare packets
- Search packets
- Import/export data



Workshop 4: Using logs and log tools

5.

Parser development

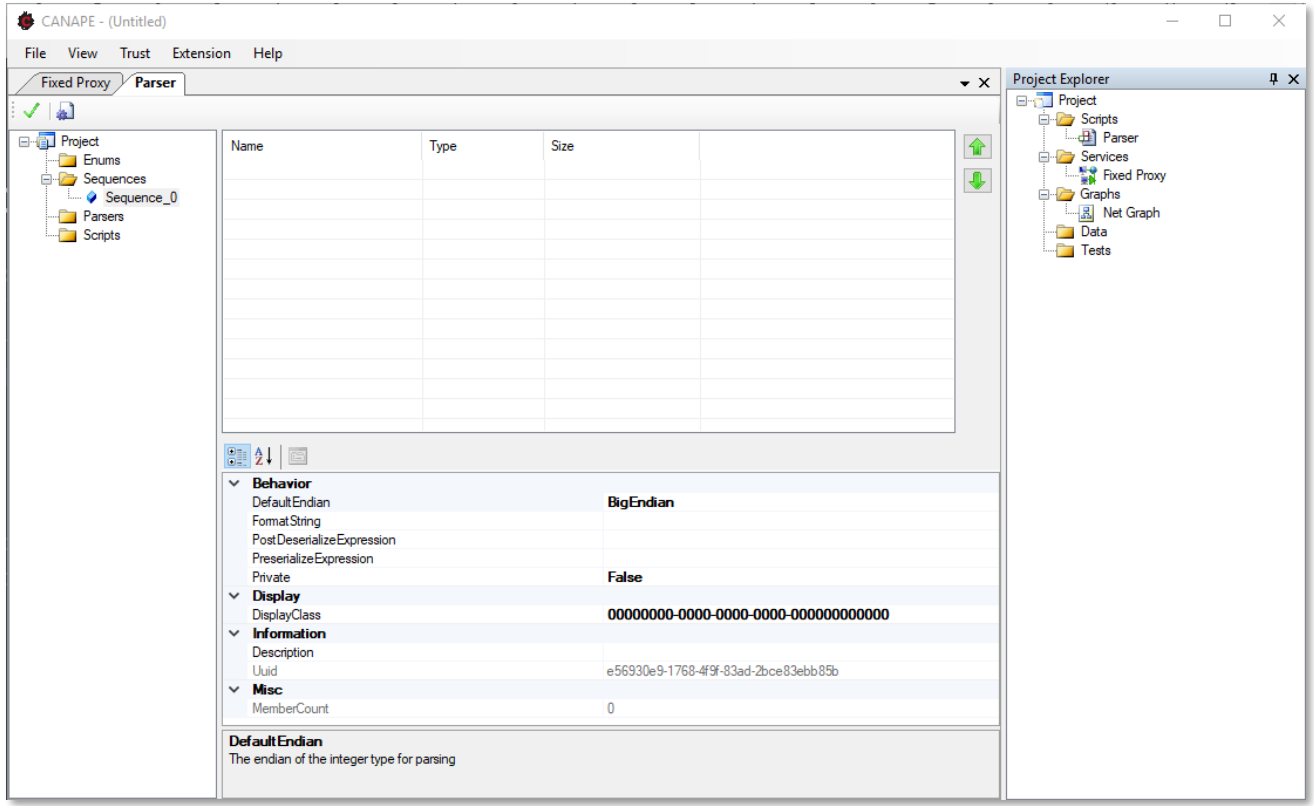
Lets look at some data

Parser editor

CANAPE's data pipeline can be scripted via C# or Python. Depending on the target protocol that must be parsed, this might be an difficult task.

Luckily, CANAPE provides a parsing editor for that task. The C# script is generated in the background.

You can add an new parser in the project explorer via "add" -> "dynamic" -> "parser"



Parser components

A parser consists of the following components:

Sequences

A sequence is the basic description of the packet format.

A sequence contains fields that have different data types. You can access sequence fields.

Sequences support complex expressions, TLV values etc. It is also possible to add sub sequences.

Enumerations

A Enumeration („Enum”) is a data type that contains a set of named values.

CANAPE differs between Value Enums and Flag Enum.

Enums can be used in sequences.

Parser

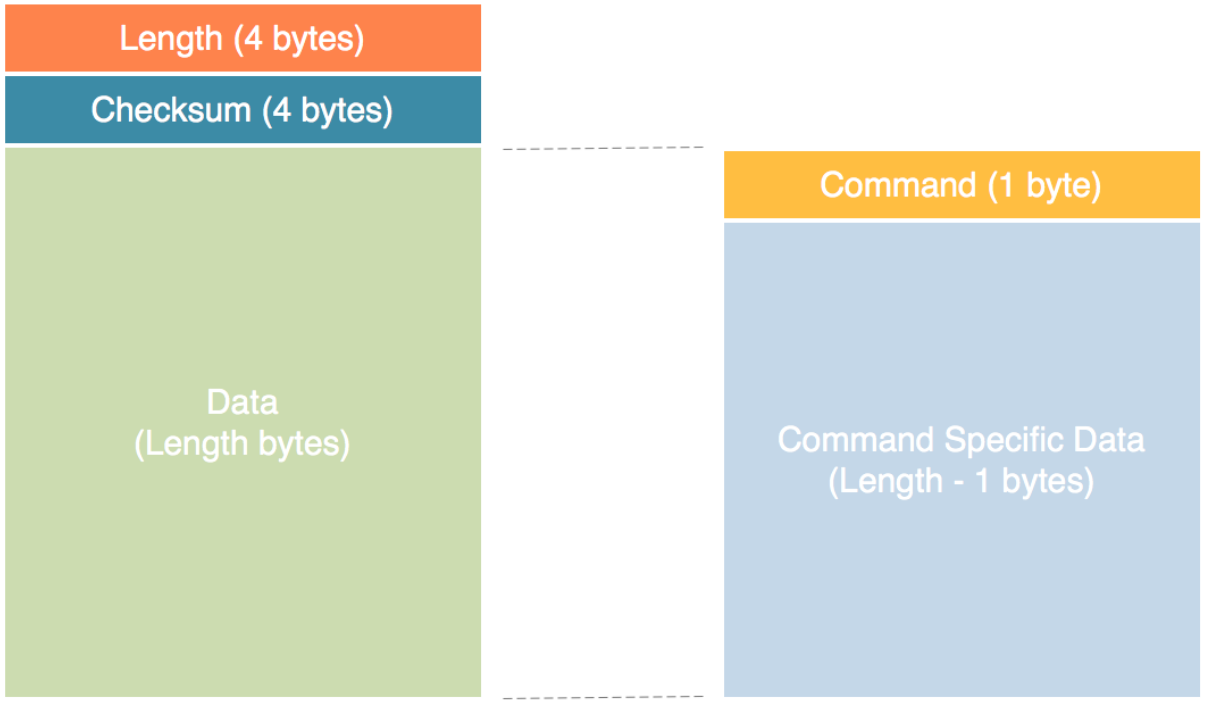
A parser element basically points to the main sequence.

SuperFunky Chat Packet structure

SuperFunkyChat uses a simple packet format shown on the right.

We will describe the packet as a sequence and use subsequences for the actual payload.

The “command” byte will be described as a Enum data type.

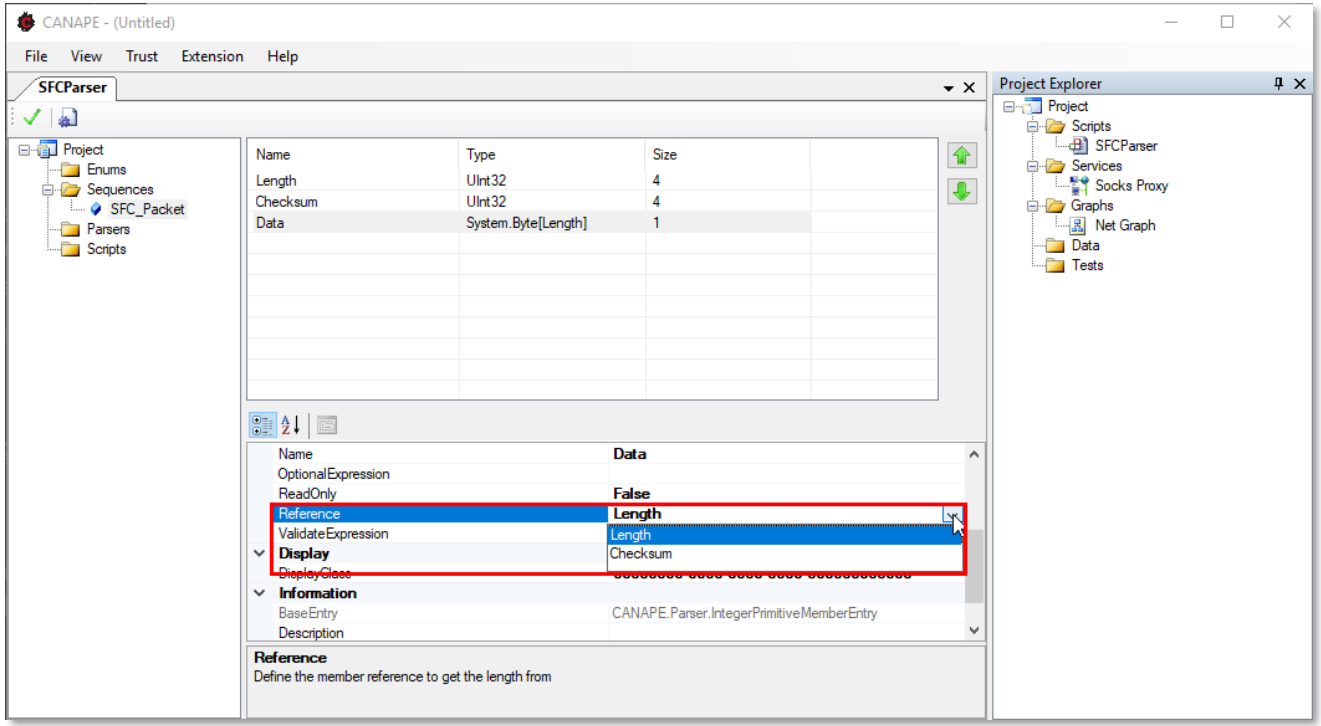


A basic sequence

We start by creating a sequence for the SuperFunkyChat. Use the right mouse button to add fields to the sequence.

The Data field is a Byte Array with a length reference (you can select this type when adding a new fields).

Set the “Reference” property to the previously defined “Length” field.

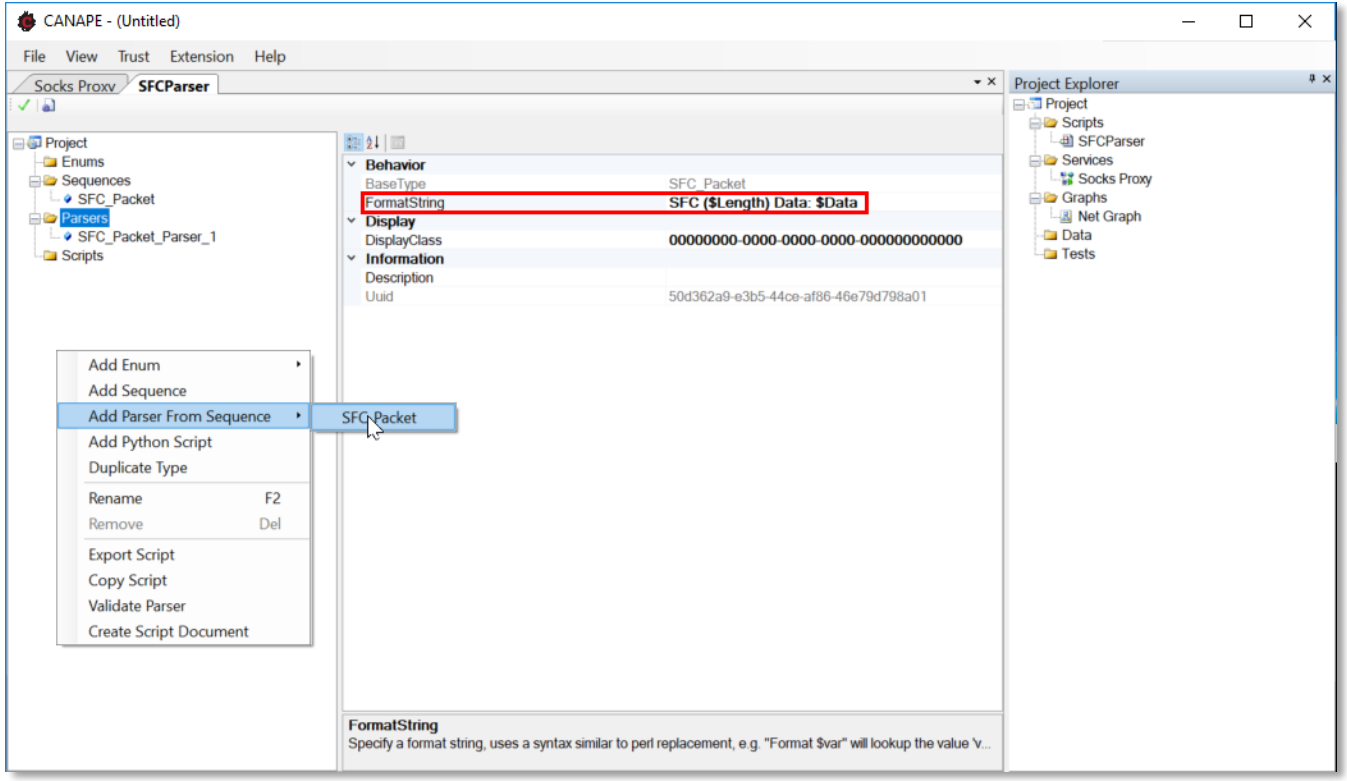


The parser

In the second step, we create a parser from the new sequence.

By changing the “FormatString” property, it is possible to define the way the packet is displayed in the logs.

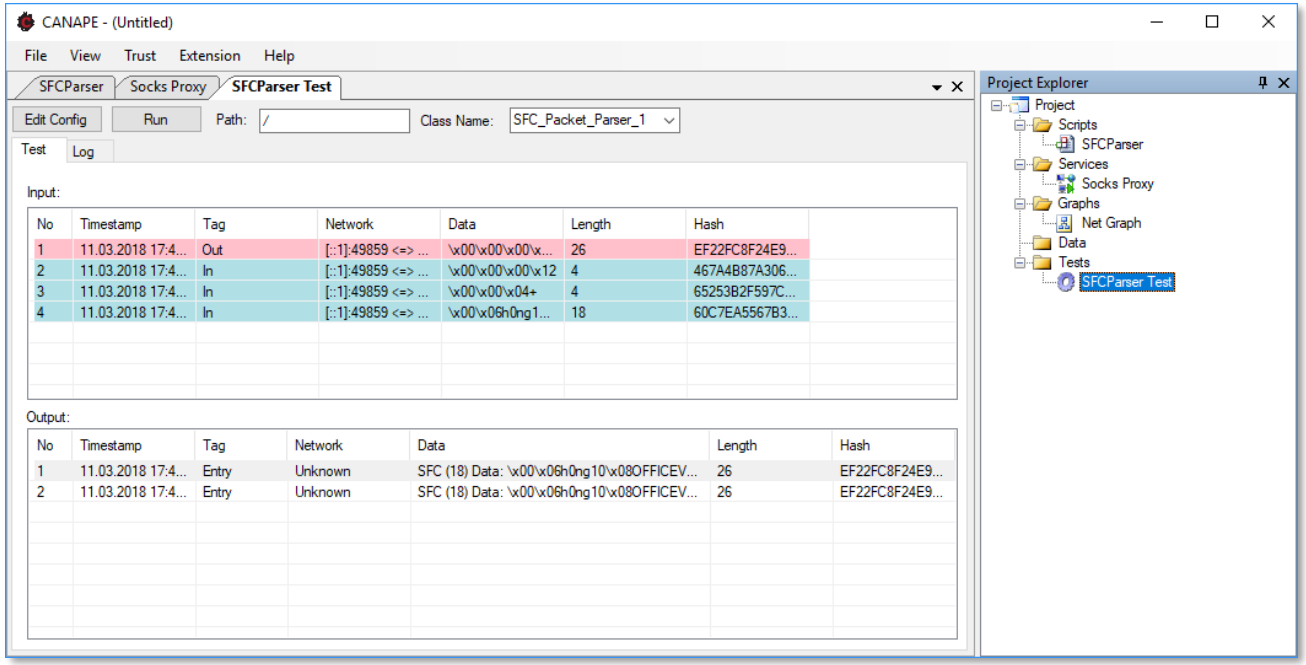
You can use Perl like expressions like $Data to access fields in the connected sequence.



Testing parsers

You can test your new parser by creating a test case via “Add” -> “Dynamic” -> “Test document”.

You can insert existing packets from the services packet log via Copy & Paste.



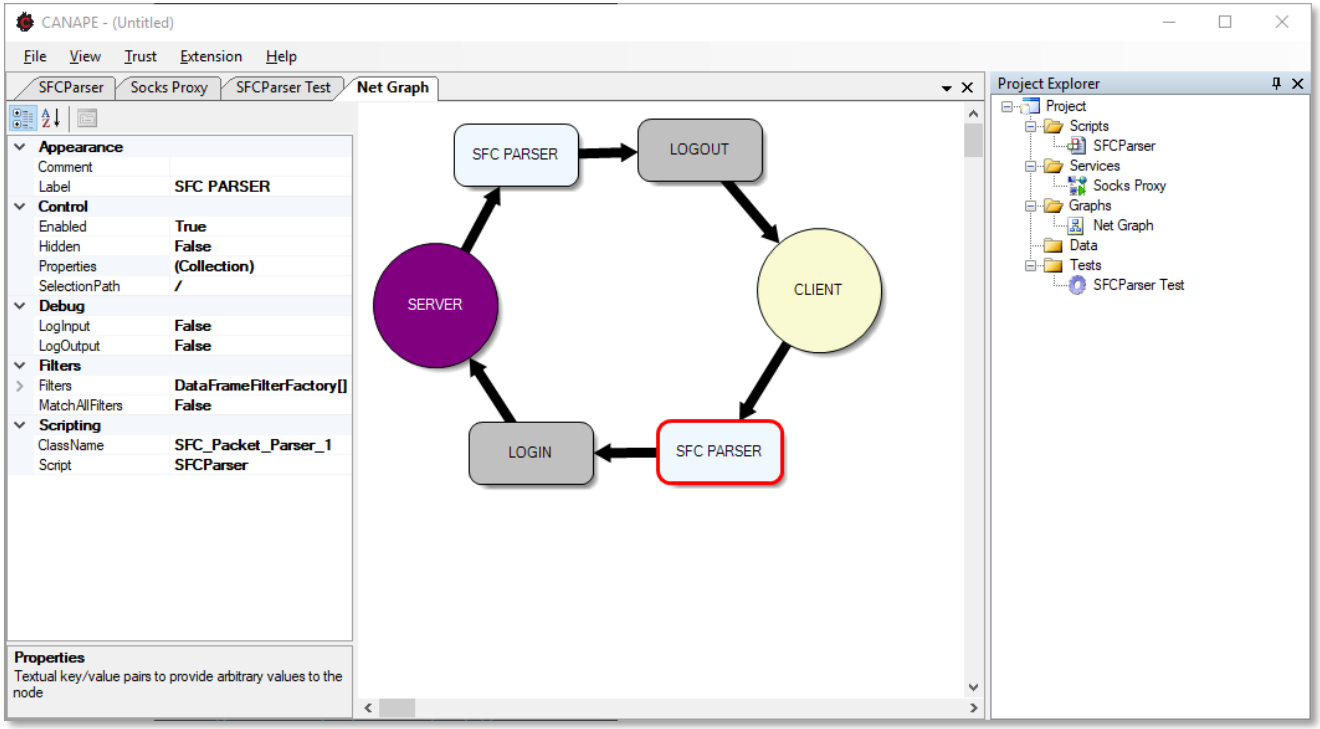
Enabling parsers

To activate a parser, insert a new dynamic node in the Net Graph.

The parser generates a C# script in the background, you can select the script/class in the node properties.

Please note that you need to add a node for both directions.

You must also restart the network service to reload the Net Graph

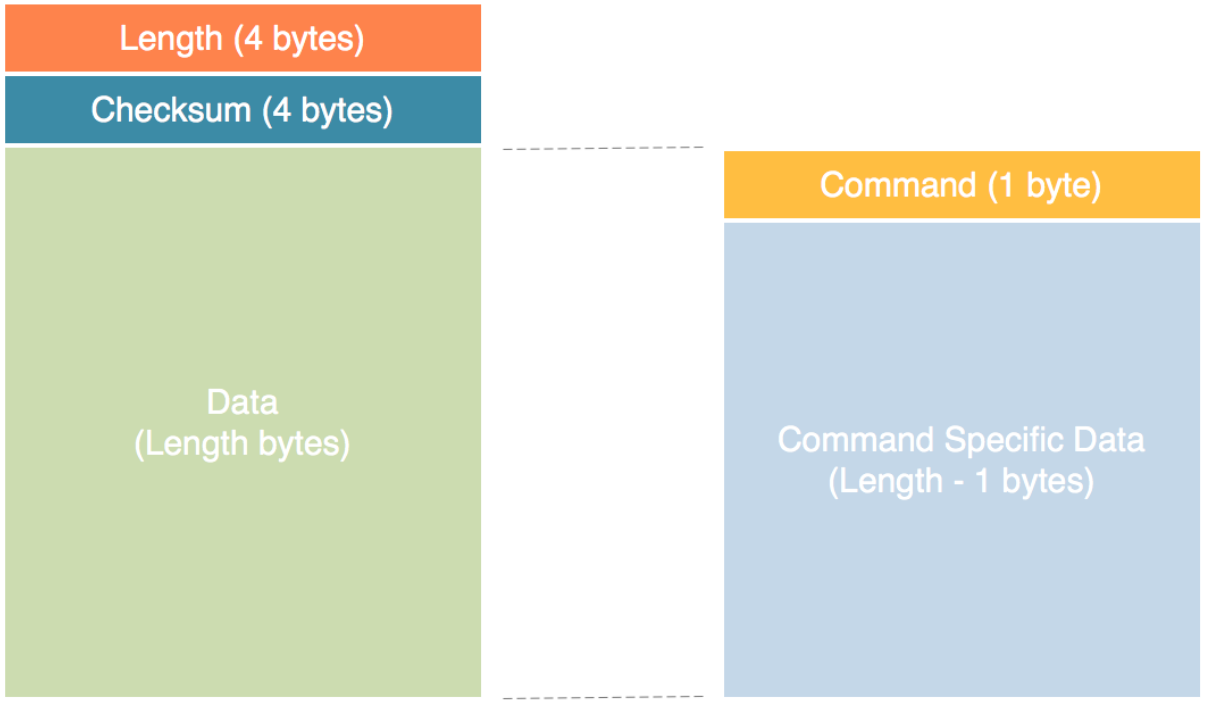


SFC Commands

So far, we only covered the main Packet format. Let's have a look at two specific command packets.

The first byte in the data part indicates the actual packet type, the rest of the data section is command specific.

Depending on the value of the command byte, we must select a different parser (subsequences) for the rest of the packet.



SFC “Hello” Command

The “Hello” Command is the first command from the client to the server.

It contains the user and system name, and if the user enabled XOR encryption

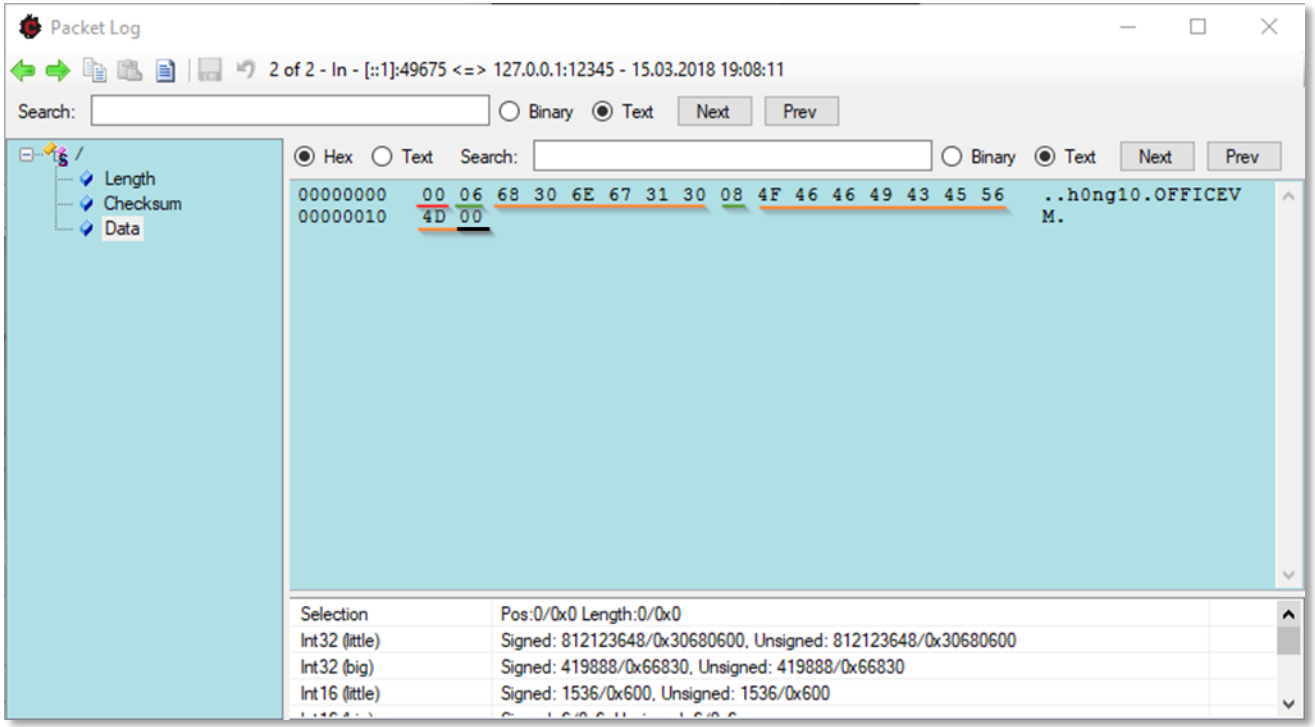
Legend:

Red: Command-Byte (00)

Green: Length (7bit Int)

Orange: Value(s)

Black: XOR Encryption (00)



SFC “Message” Command

The “Message” Command is used when the user sends a new text message.

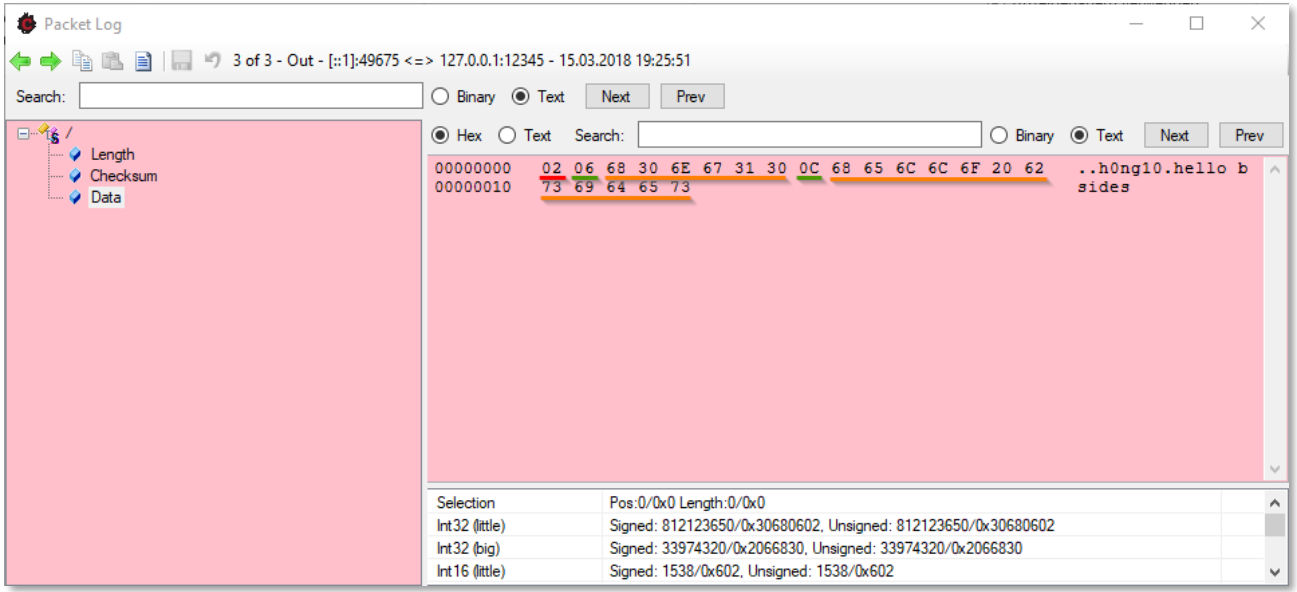
The structure is very similar to the “Hello” command

Legend:

Red: Command-Byte (02)

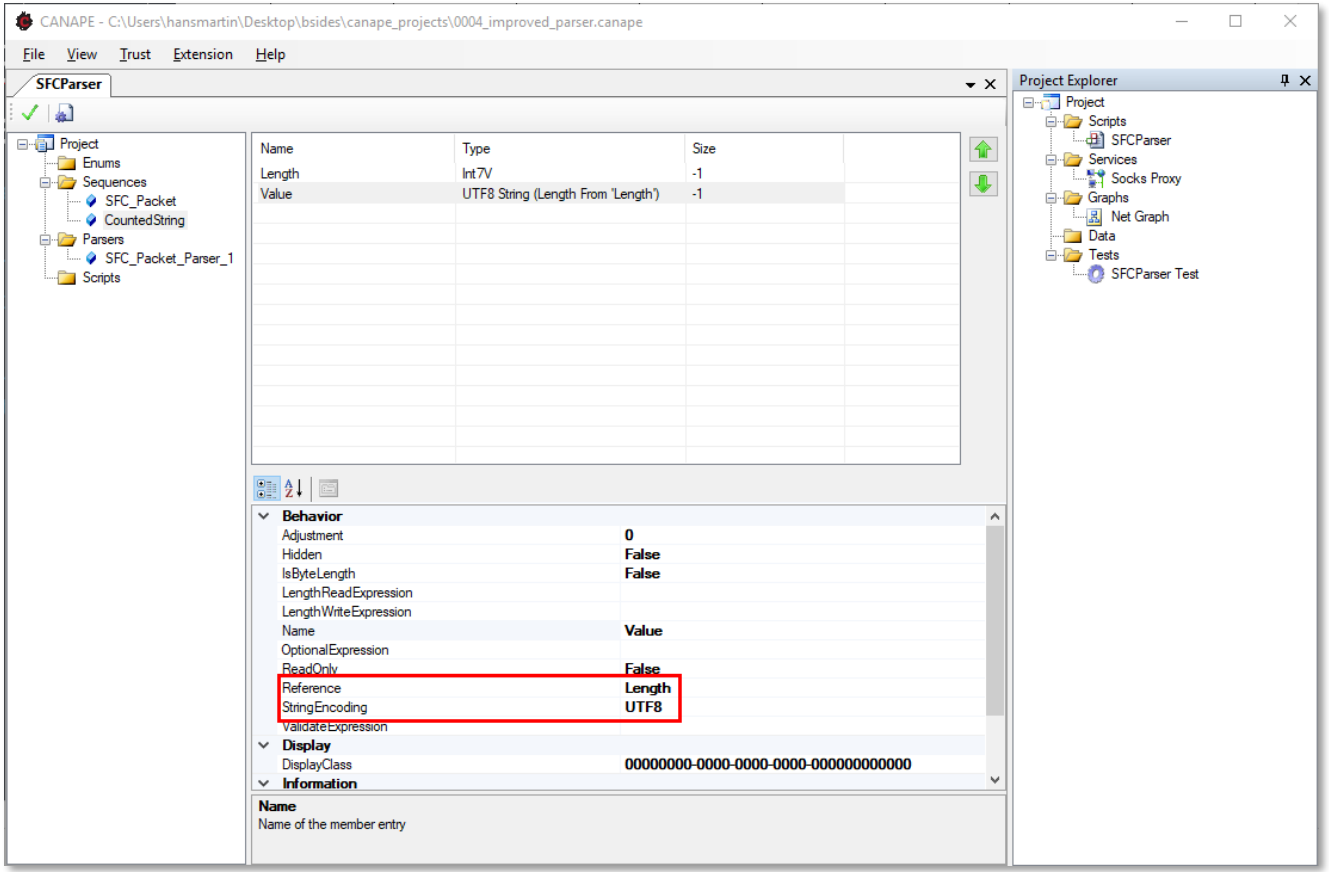
Green: Length (7bit Int)

Orange: Value(s)



CountedString Subsequence

Since most SFC packets use a combination of a 7bit variable INT for length and the actual string, it makes sense to generate a dedicated subsequence for this data type. We can use this subsequence to describe the actual payload packets.

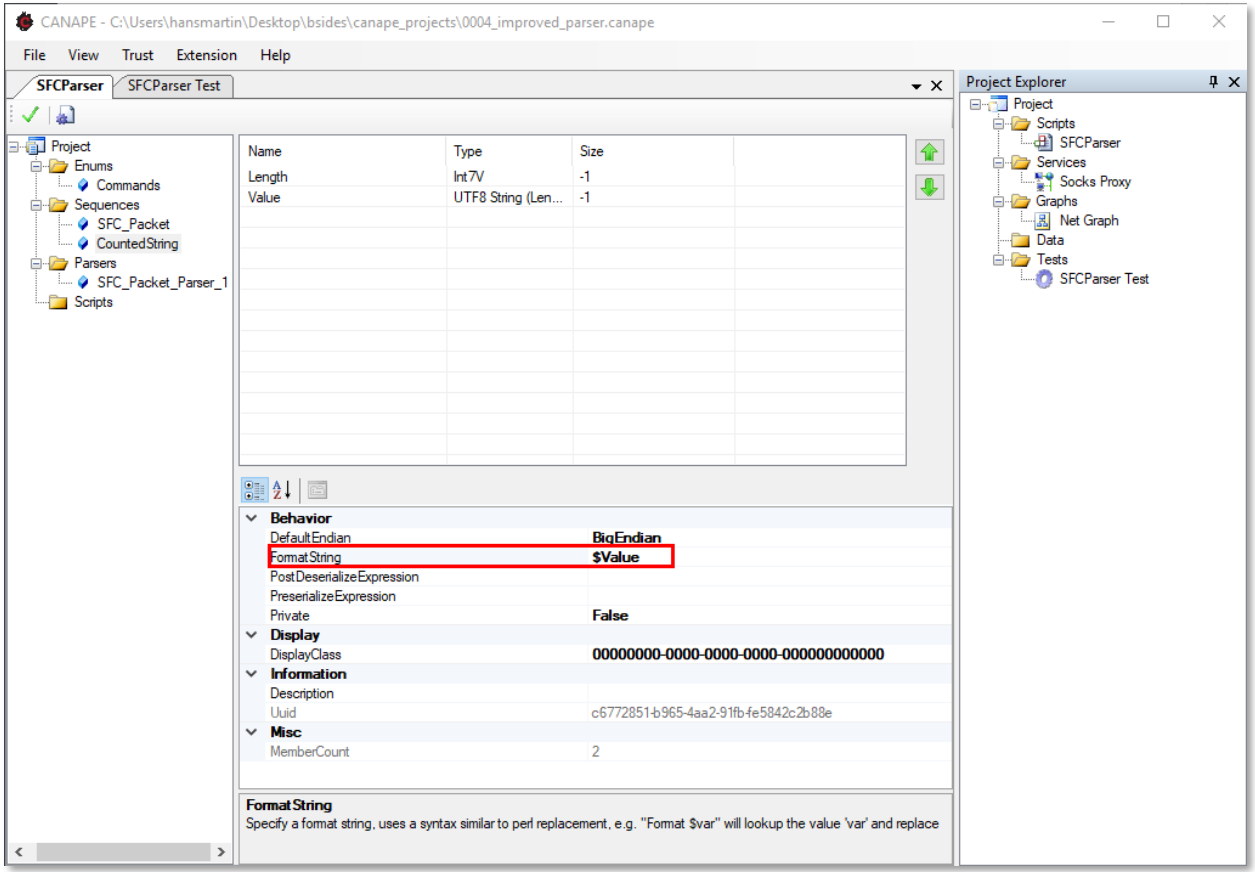


FormatString property

Please make sure that the FormatString is set to “$Value”.

This is important, otherwise the Strings wont be displayed correctly in the packet logs.

Note: Set the FormatString property for **all** sequences, including the ones that we create next.



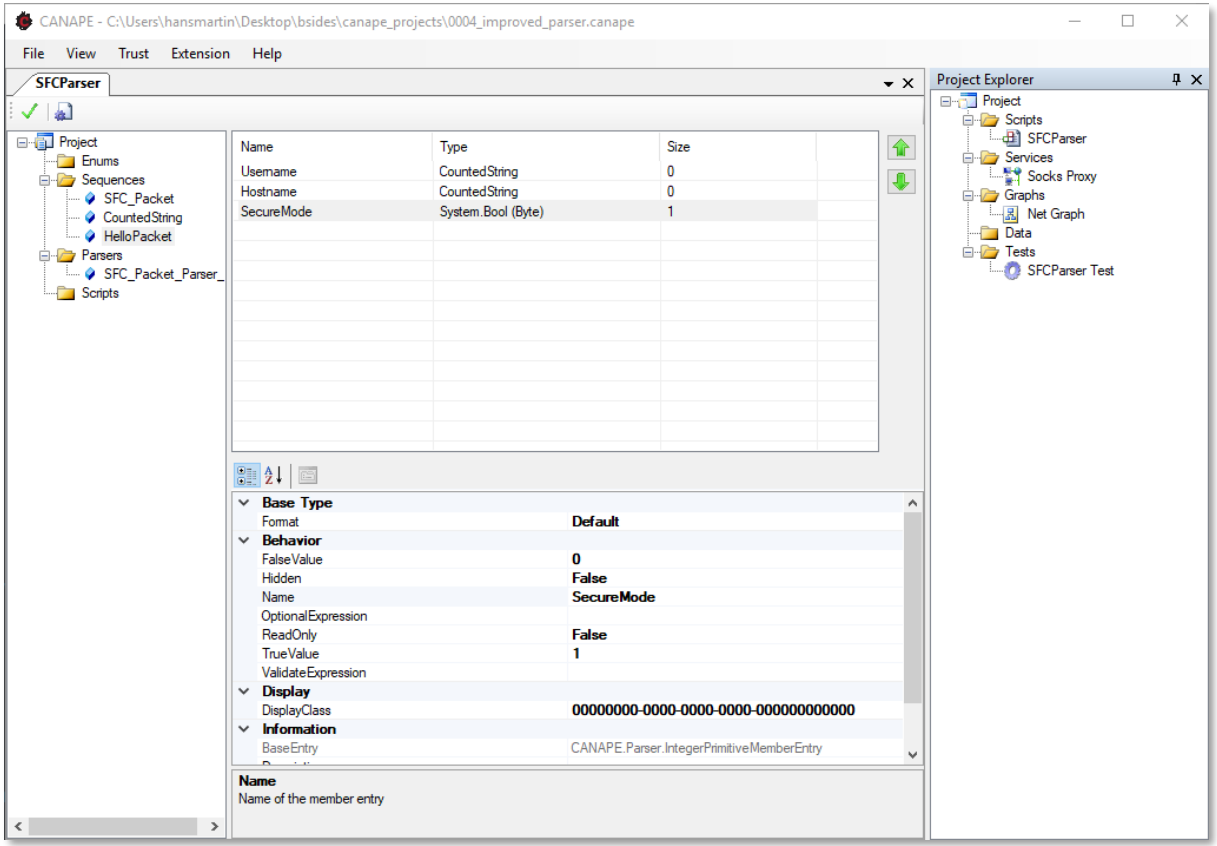
Hello Packet

The “Hello” Packet can be described with the subsequence on the right.

Please take into account that it doesn’t include the actual command byte.

This sequence also contains the previously generated “CountedString” sequence, used as a data type.

Use a unsigned byte for the SecureMode and convert it to a Boolean in a second step.

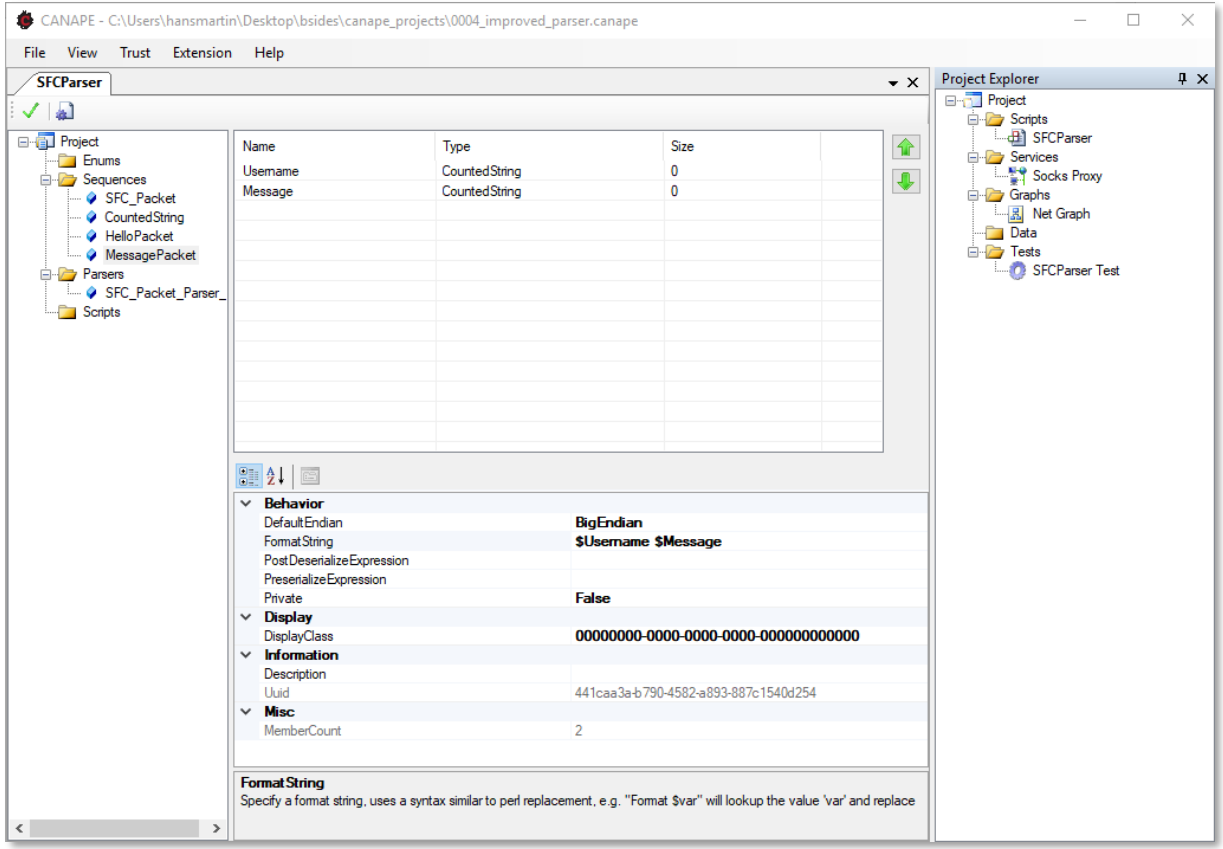


Message Packet

The “Message” Packet can be described with the subsequence on the right.

Please take into account that we didn’t include the actual command byte

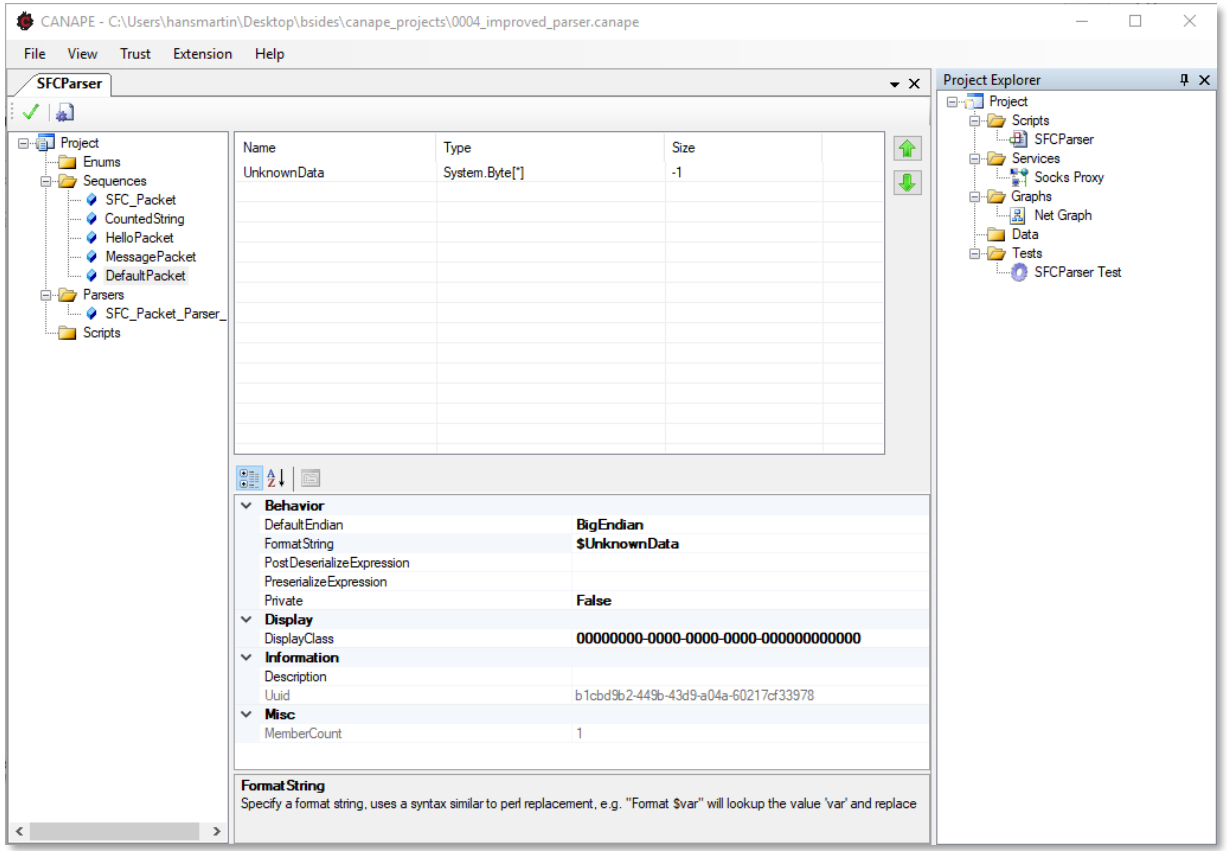
The sequence also contains the previously generated “CountedString” sequence as a data type.



Default Packet

As we didn't describe all existing commands as packets, we will need a "default" packet which is just a simple byte array (please use "read to end").

This sequence will later be used for payloads that don't match a known command.



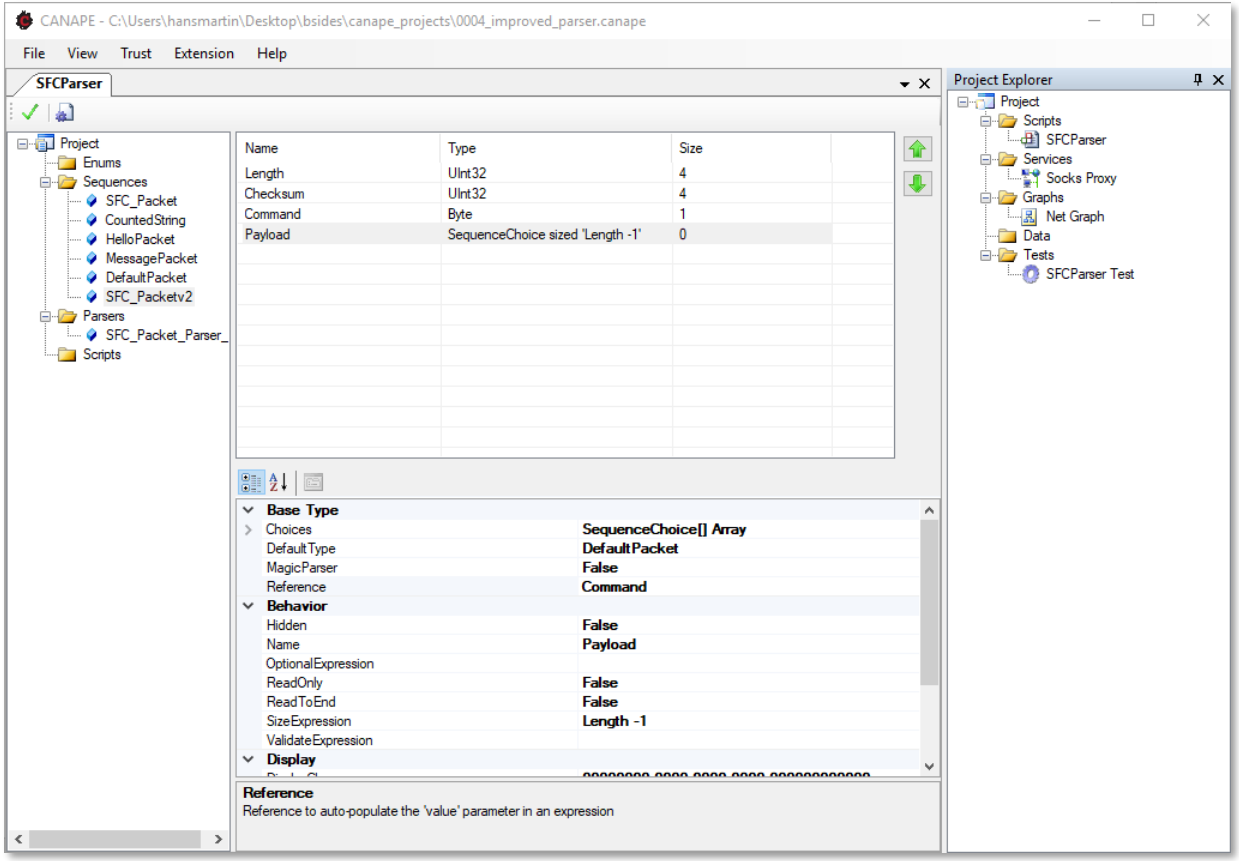
SFC_Packetv2

We now have everything prepared to generate a new main sequence.

The structure is similar to the previous SFC_Packet sequence but we use a SequenceChoice to select the actual payload subsequence.

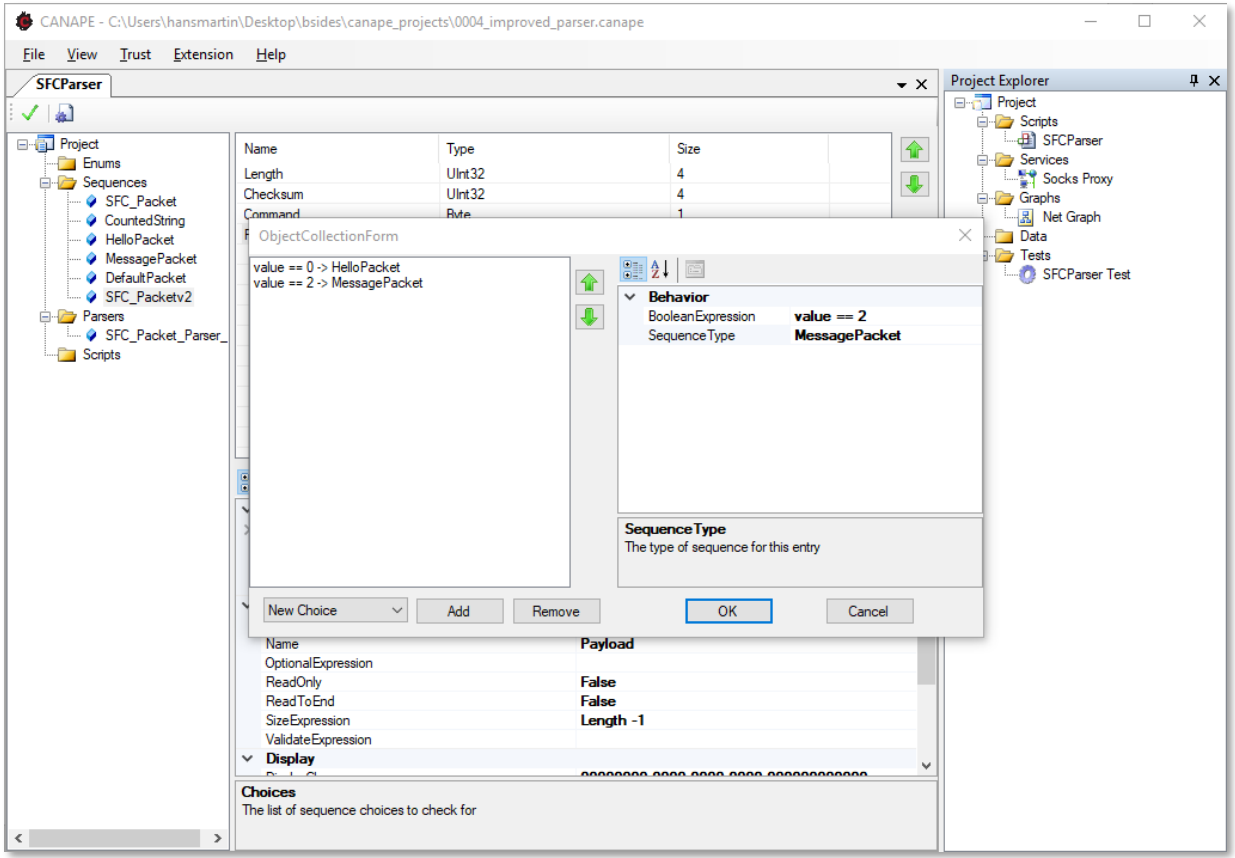
Please select “convert to sized” in a second step. You can the use a simple expression to describe the size.

You must also select the default packet sequence for “Default Type”



Sequence Choice

If you click on “Choices”, you can define different subsequences for the Reference values.



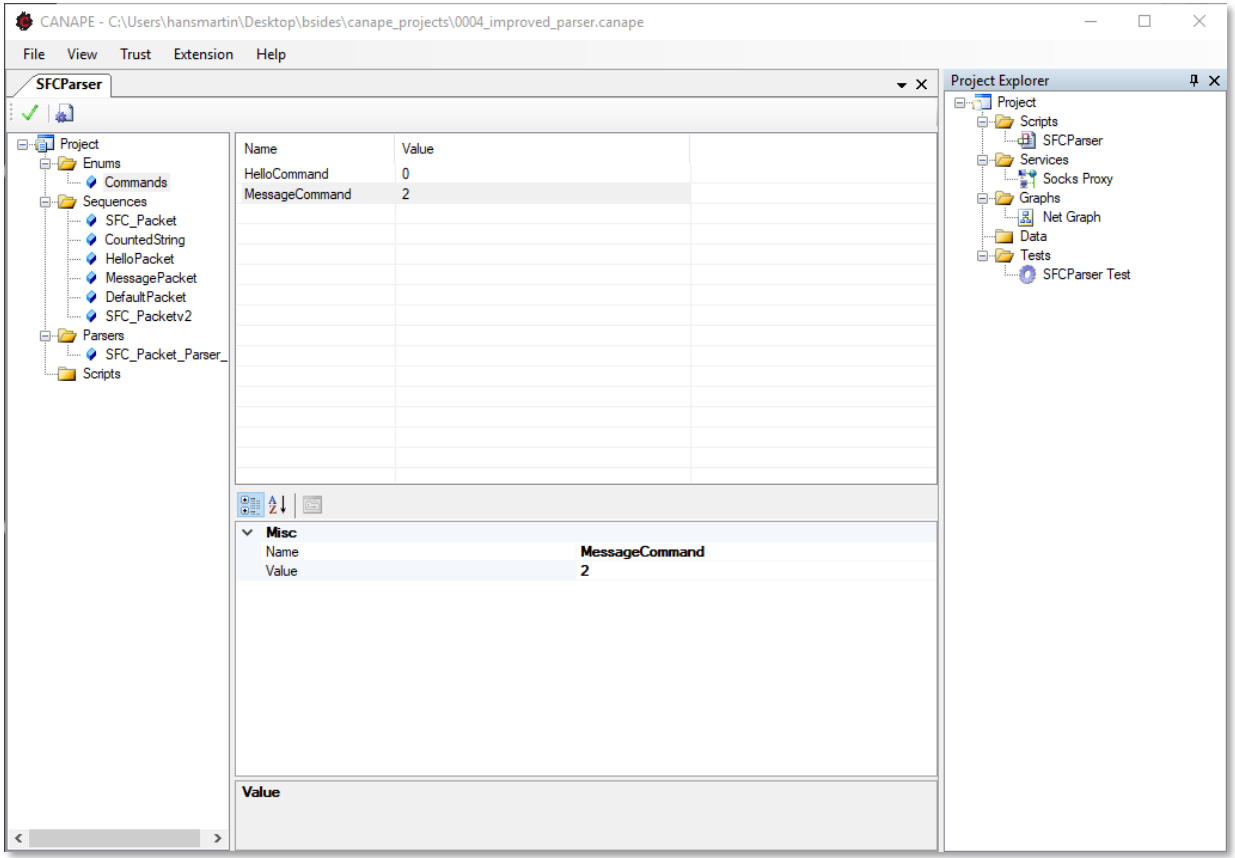
Commands Enumeration

We can generate a Enum data type to represent the different commands from the command byte.

CANAPE supports two Enumeration types:

- Value enums for single values
- Flag enums for bitwise enumerations (combi-nations of multiple values)

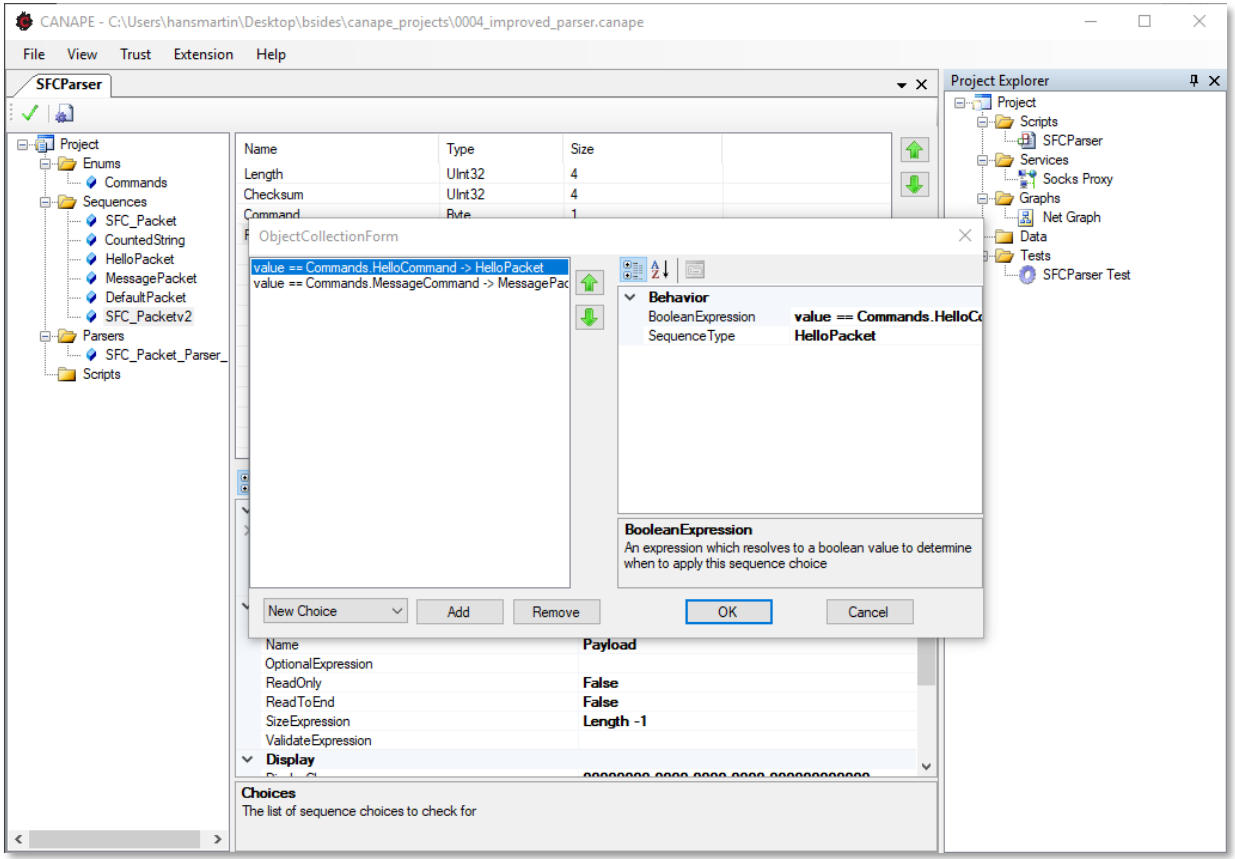
Please create a new “Value Enum” and populate him as shown in the screenshot.



Sequence Choice

We can use the “Commands” Enum in the BooleanExpression of the Payload selection.

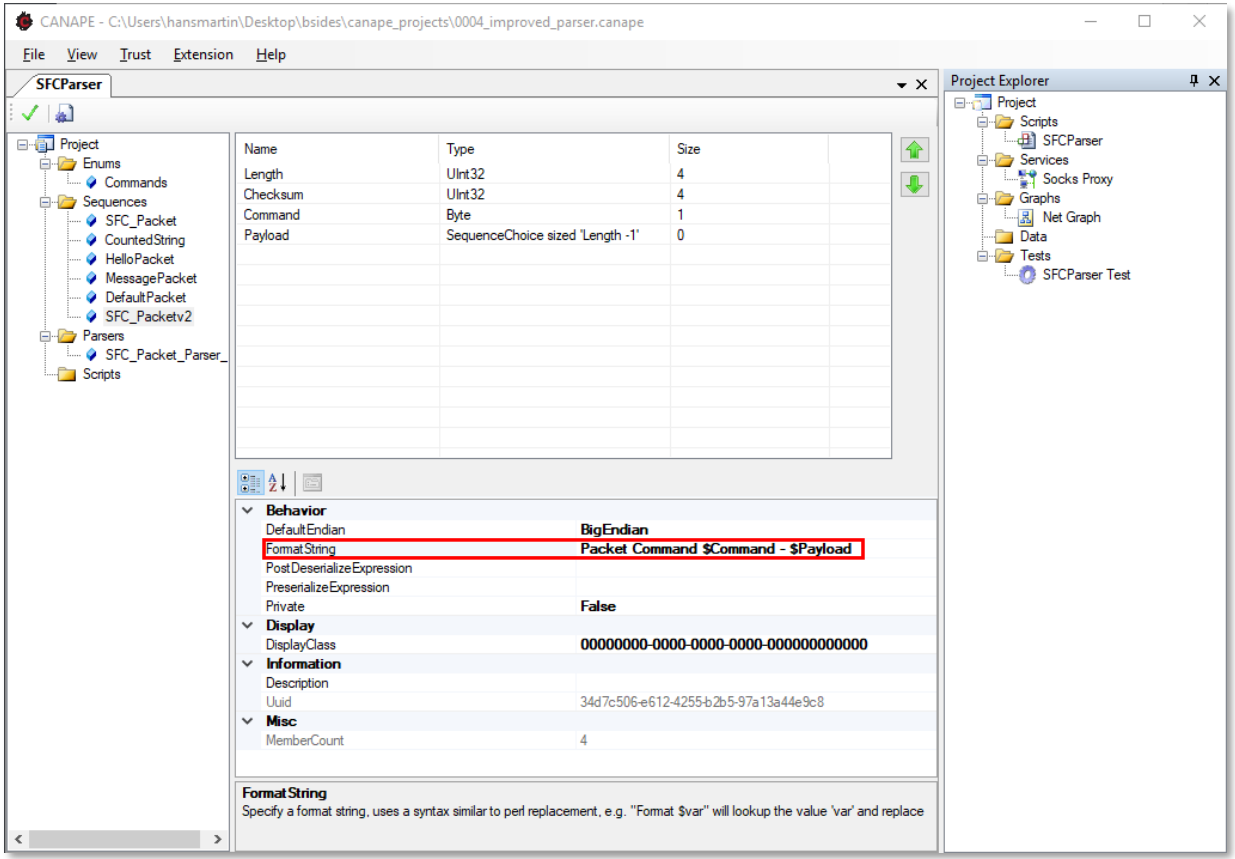
After that, we only need to generate a new parser from the SFC_Packetv2 sequence



SFC_Packetv2 FormatString

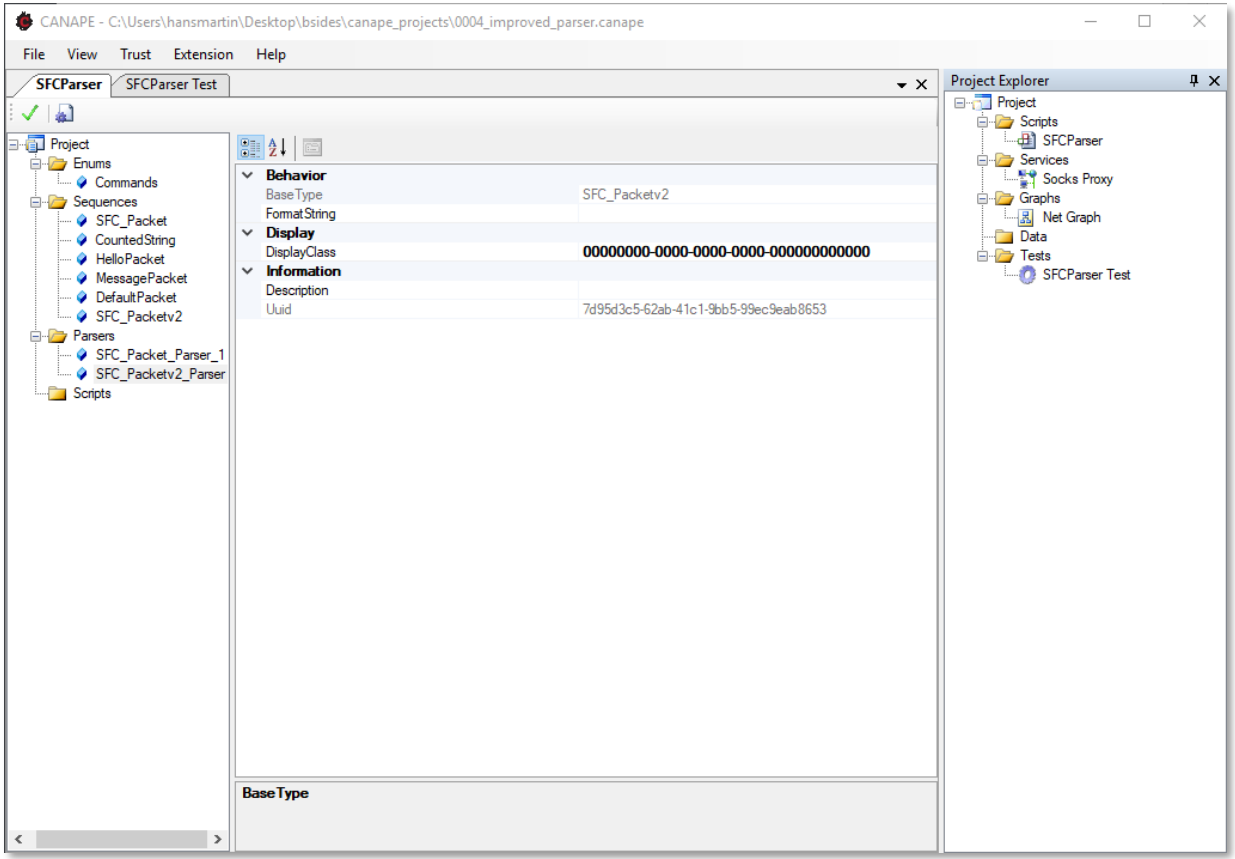
Last but not least, we update the FormatString property of the SFC_Packetv2.

The displayed format string will first print the general packet information and then the payload details in the logs



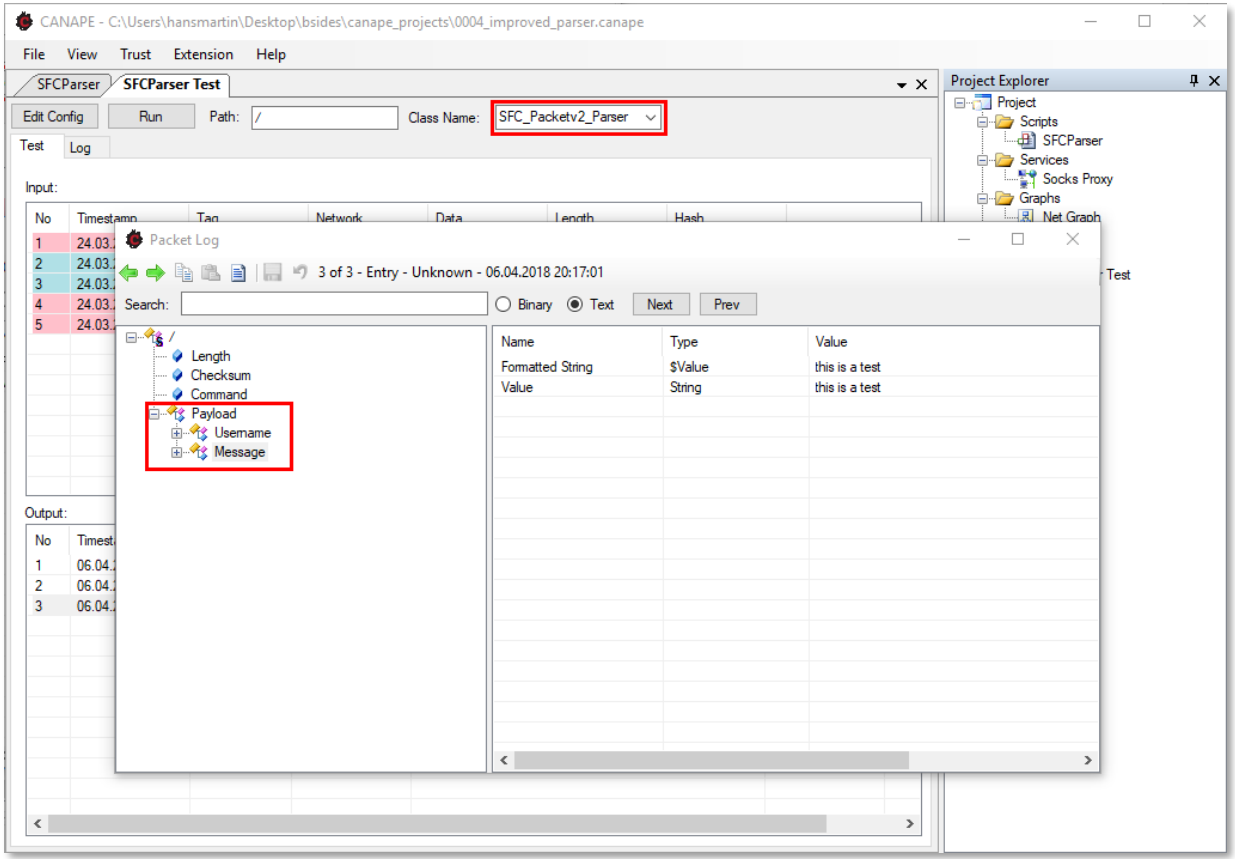
Create a second parser

Create a new parser from the SFC_Packetv2 sequence, similar to the existing one.



Testing the new parser

Select the previously created test case “SFCParserTest” to verify if every packet gets parsed correctly and all values are displayed.



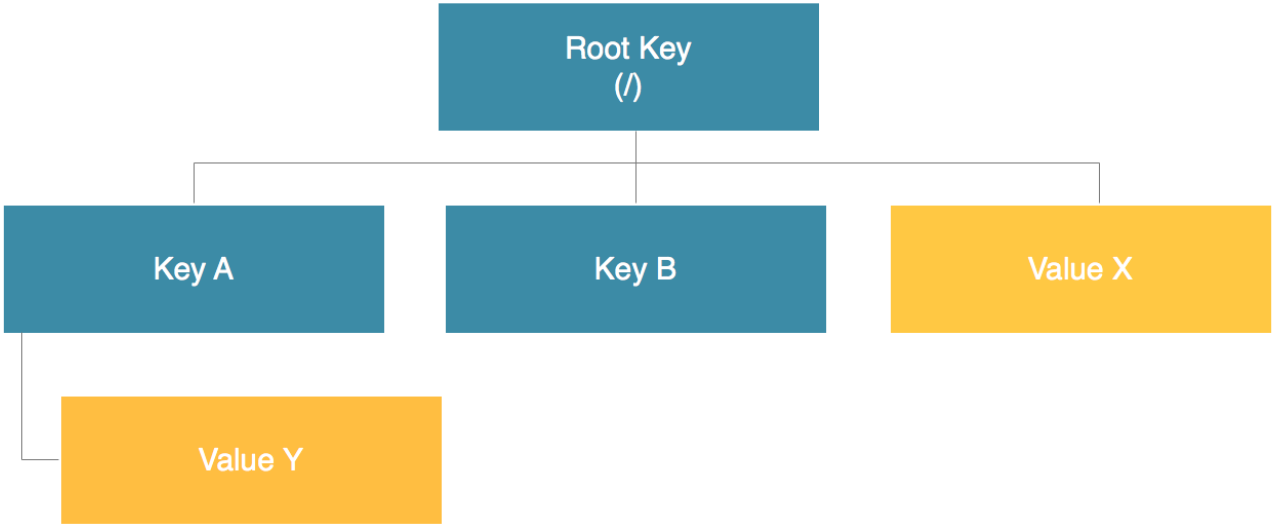
Parsed packet format

Nodes and filters support a selection path that allows you to easily access the different fields in a parsed packet.

The syntax is similar to XPATH. For example, you can use:

/payload/username

to access the username from the payload.




Workshop 5: Creating parsers

6.

Working with layers

Going down the rabbit hole..

Protocol Layers

Sometimes binary protocols are wrapped into others. The most common example for a layered protocol is TLS/SSL which tunnels arbitrary protocols (for example HTTP,SMTP) into an encrypted channel.

To deal with this problem, CANAPE provides a “layer” support which allows you to unwrap the connection on the fly. TLS/SSL en-/decryption is fully supported.

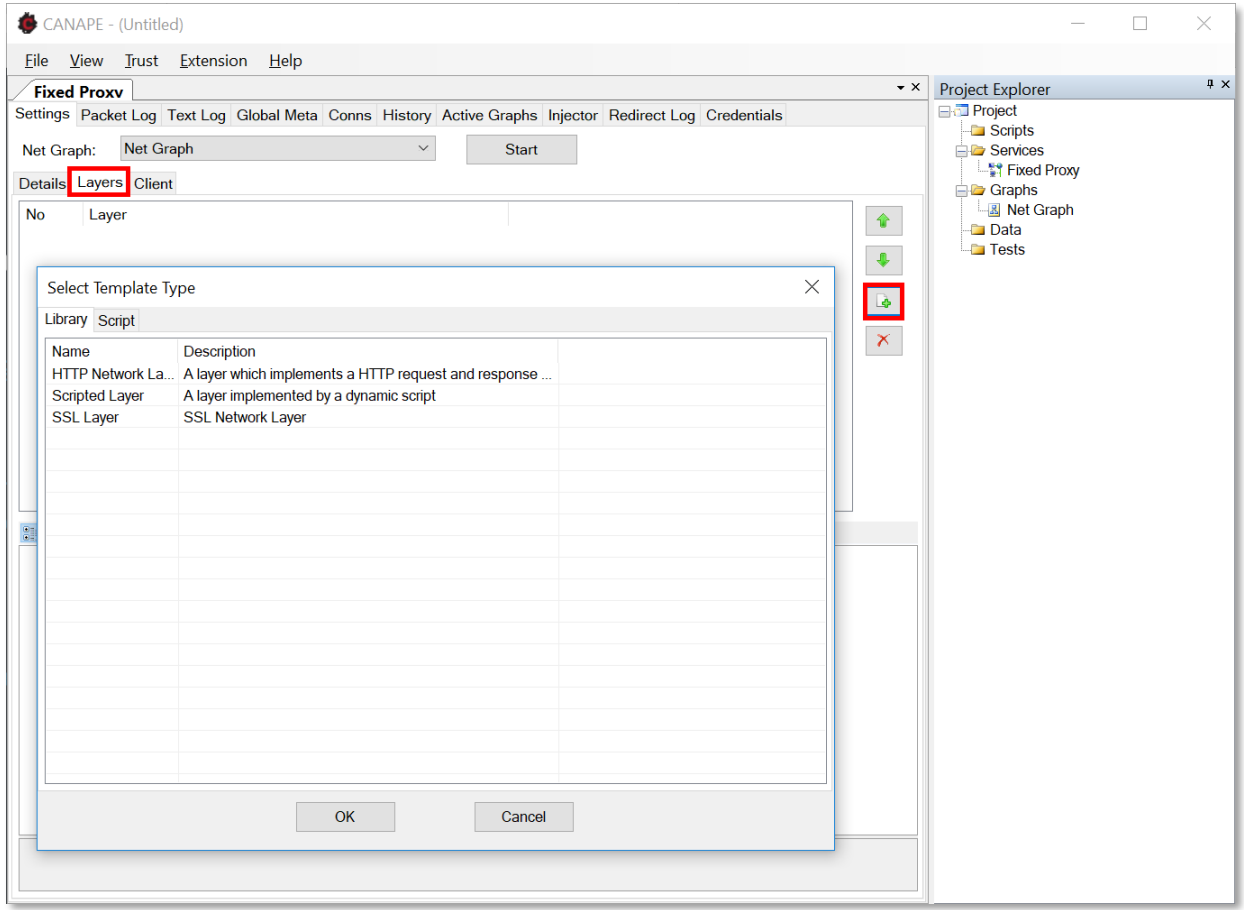
CANAPE doesn’t assume it knows better than you if a connection is TLS encrypted or not. You have to define this inside the service configuration.

Fixed proxies

The settings of fixed proxy services allow a direct configuration of layers.

CANAPE provides out of the box support for TLS/SSL encryption layers.

However, you can also add a “scripted layer” here and get fully creative ☺



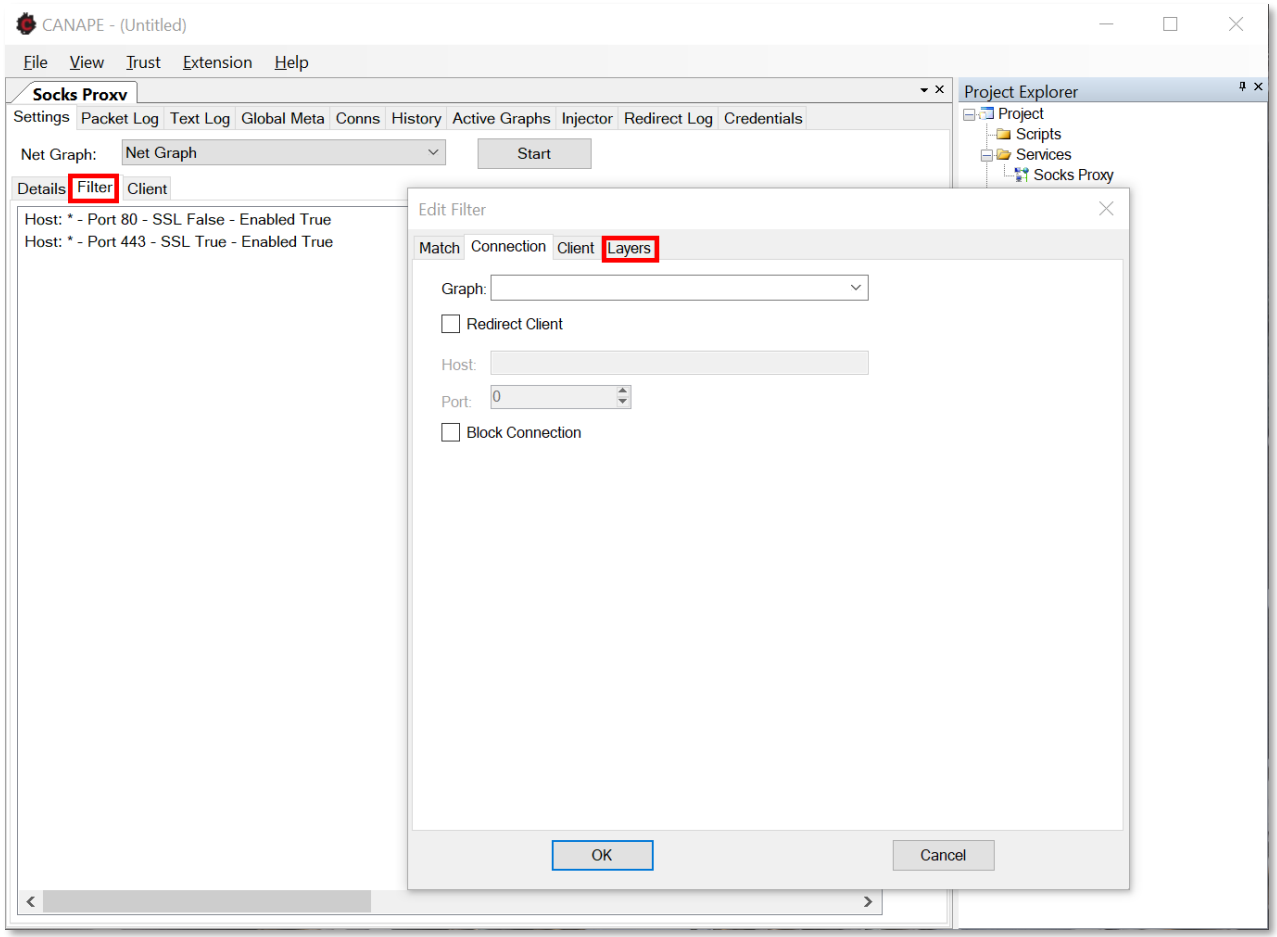
SOCKS proxies

If you configure a SOCKS proxy service, you have to deal with the problem that it might be used by multiple protocols.

The SOCKS configuration in CANAPE allows you to define filters to deal with different ports.

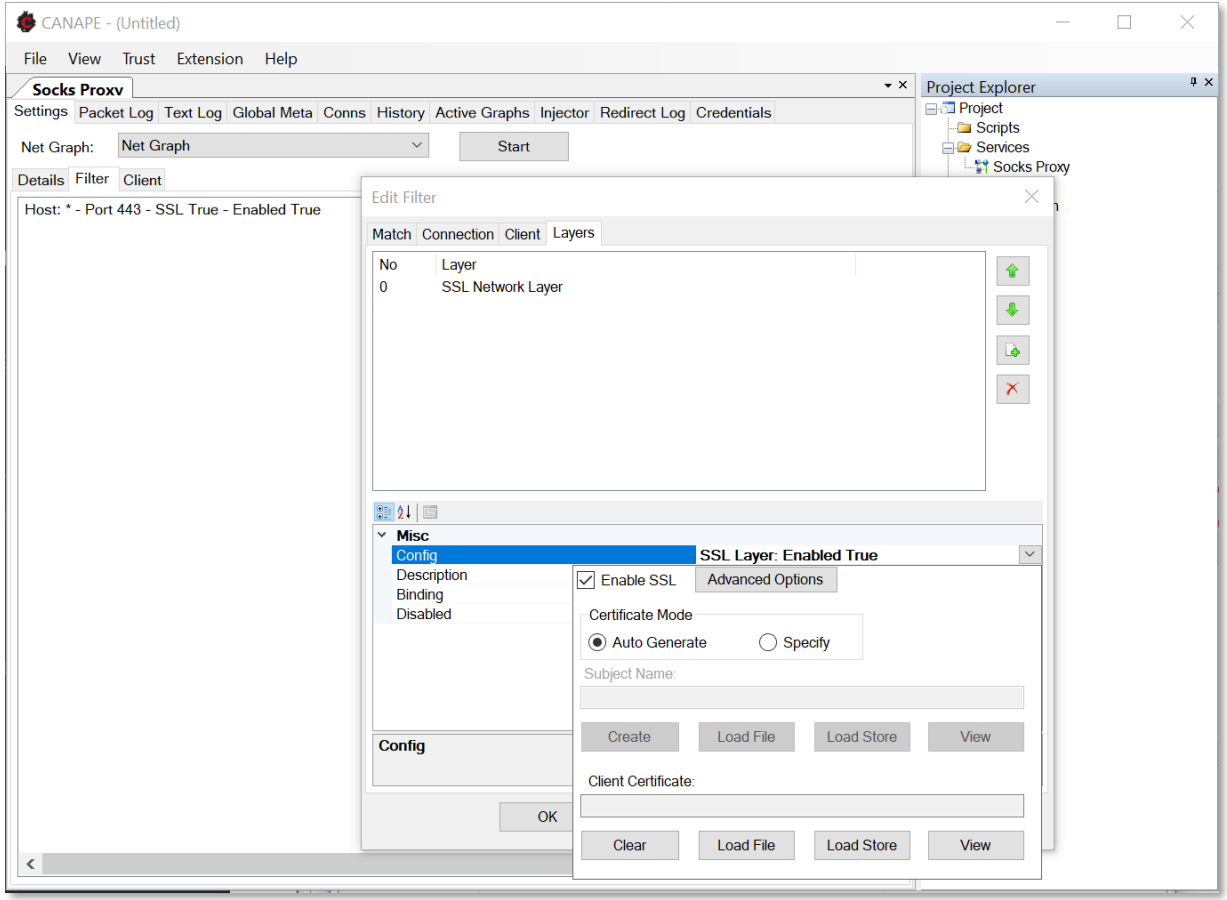
You can filters services for specific hosts/ports and even select a separate NetGraph for each service.

You can also define Layers for the protocol there...



SSL settings

The SSL layer configuration allows you to select a server and client certificate, however CANAPE can also auto generate certificates for you.



Workshop 6: Removing TLS layers

7.

STATE MODELLING

State Modelling

Most protocols aren‘t simple (stateless) streams of data, were every stream is processed independently. So we need a way to handle the protocol state.

The main way of handling state in CANAPE is through Net Graphs. However, CANAPE also provides a simpler way via „State Graphs“.

If we want to support the XOR encryption, of SuperFunkyChat, we need to deal with two different states:

- No encryption (default)
- XOR encryption

The client informs the server in the HelloPacket if he want to use XOR encryption. The server then responses with the XOR key. The client (and CANAPE) must save this XOR key.

Meta Data Storage

Depending on the target protocol and your project it is often necessary to store meta data with sate information.

Canape provides three meta storages locations for this task.

Meta Storage	Description
Local Meta Storage	Meta Data is only accessible from the current service instance, similar to a a private class variable
Global Meta Storage	Meta Data can be shared between different instances of the same service, similar to a protected class variable
Project Meta Storage	Meta Data can be accessed from every service, similar to a global variable. As using the Project Meta Storage is pretty niche, you can‘t easily access these values from the GUI, only via C#/Python.

Accessing Meta Data

You can access the Meta Storage from C# or Python code. Some Python examples on the right.

CANAPE also provides useful methods to increment/decrement counters in the Meta storage.

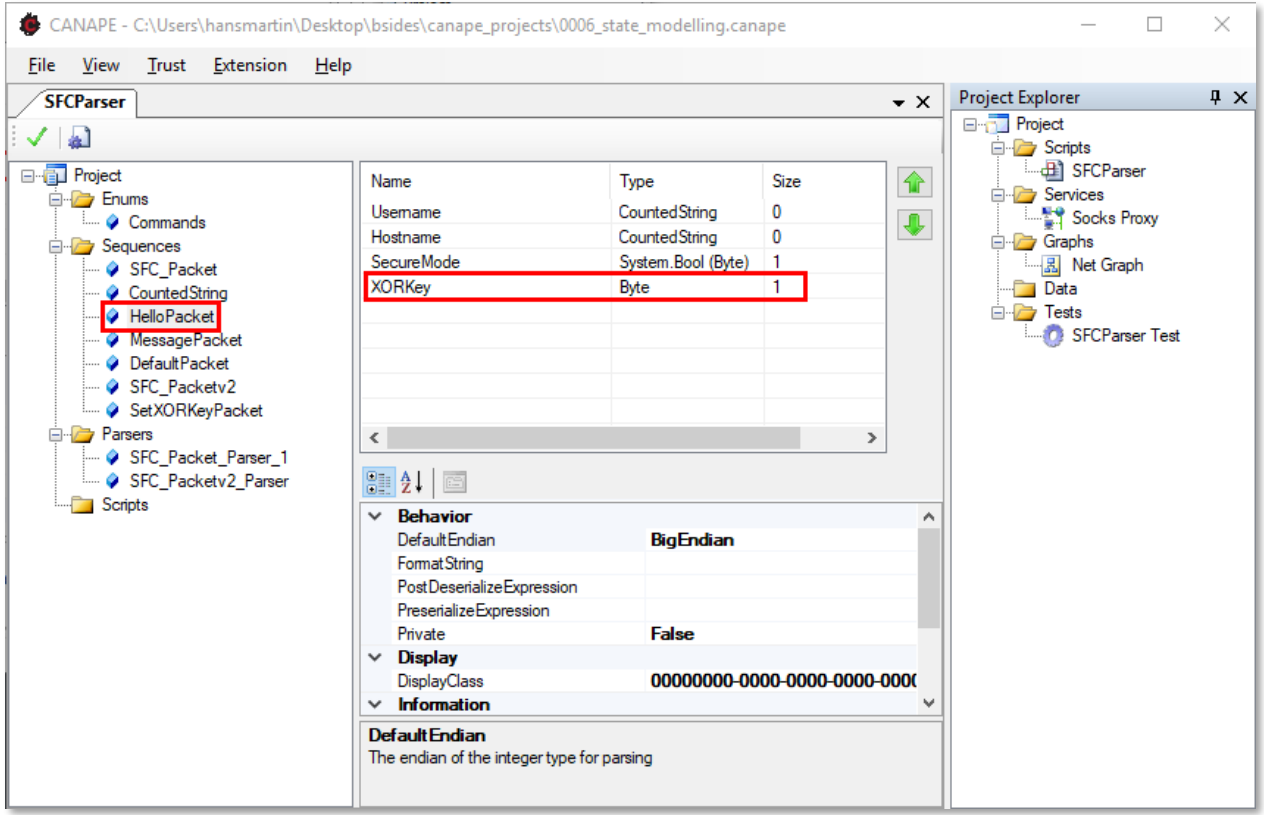
Hint: Look at the CANAPE source (NetGraph.cs) for available functions.

```python
# Access local Meta storage
myValue = self.Meta.GetMeta('myKey')
self.Meta.SetMeta('myKey', myValue)

# Access global Meta storage
myGlobalValue = self.Meta.GetGlobalMeta('myGlobalKey')
self.Meta.SetMeta('myGlobalKey', myGlobalValue)

# Get local counter value from local Meta storage
counterValue = self.Graph.GetCounter('localCount')

# Get local counter value, return 0 if counter doesn't exist
counterValue = self.Graph.GetCounter('notThere', 0)

# Set value of new counter to 1
self.Graph.SetCounter('newCounter', 1)

# Increment global counter py 2
self.Graph.IncrementGlobalCounter("CurrentCommand", 1)
```

Upgrading HelloPacket

The key exchange packet only contains the XOR key. We use a minimal sequence similar to the existing packets.

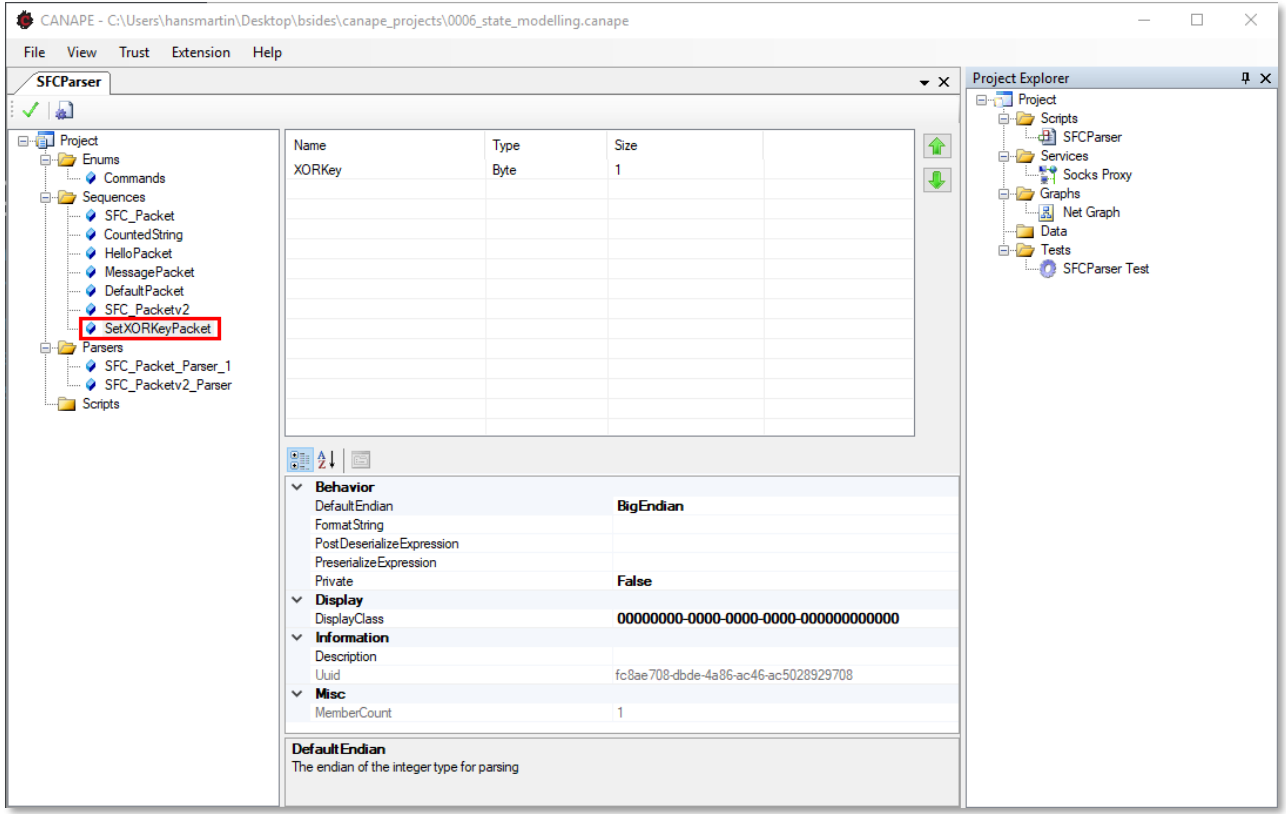
The XOR key is just one byte long.



Key exchange

The key exchange packet only contains the XOR key. We use a minimal sequence similar to the existing packets.

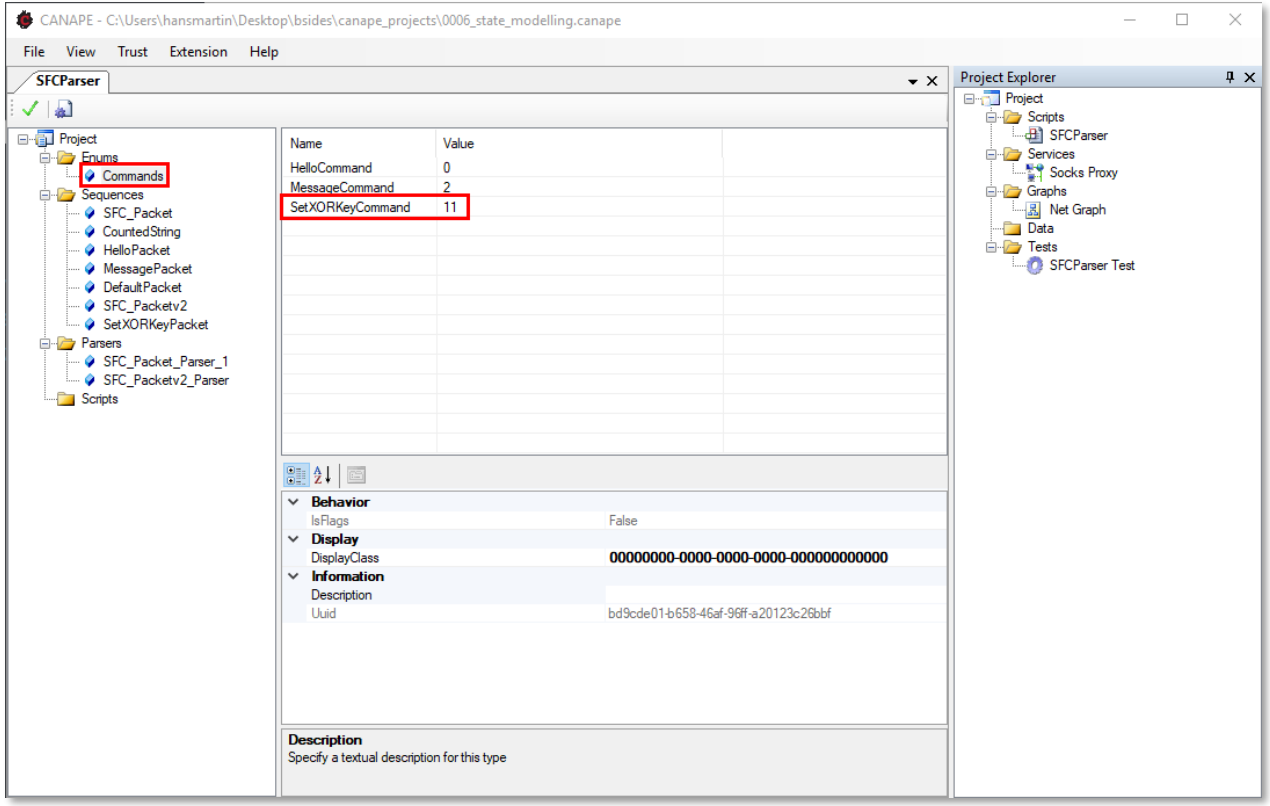
(this step is optional)



Updating Commands

We update the “Commands” enumeration by adding a “SetXORKey” command. Set the value to 11.

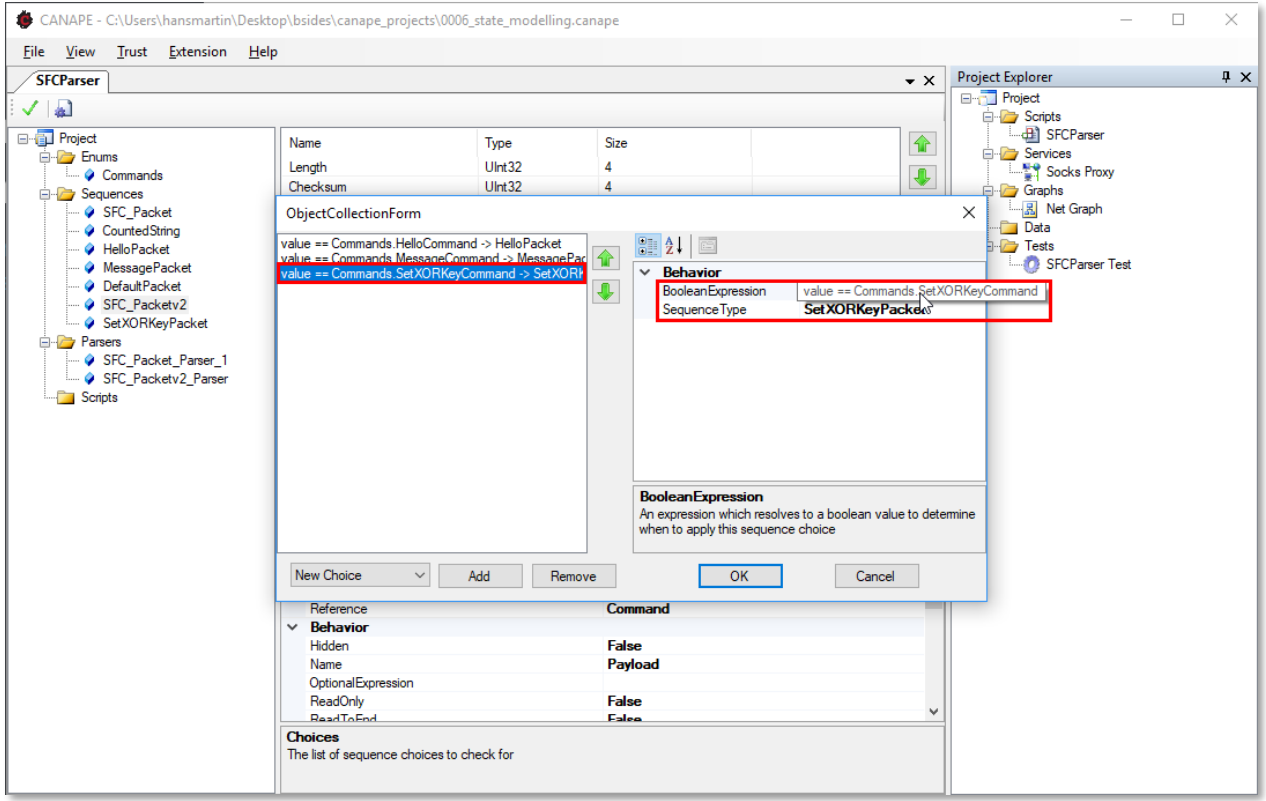
(this step is also optional)



Updating SFC_Packetv2

We finally adding the sequence to the Payload Selection .

(this step is also optional)



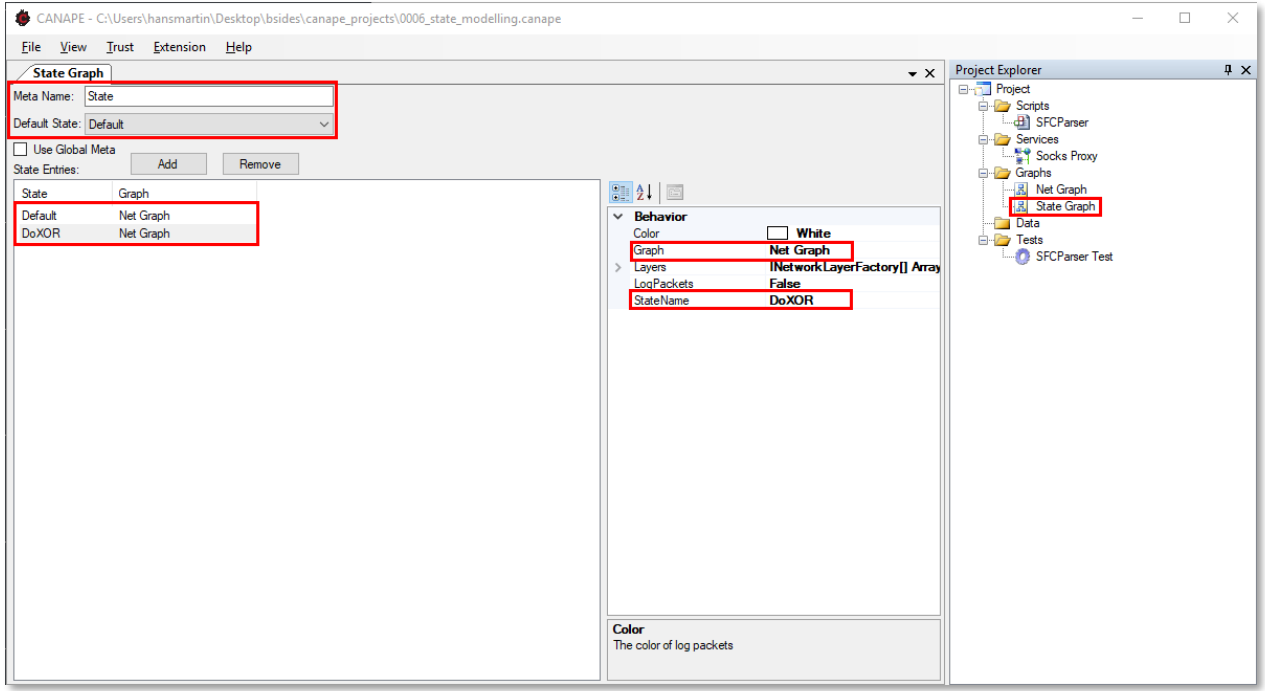
Adding the state graph

Add a new state graph (add -> net graph -> state graph) and name it “State”.

SFC has two states:

- Default (no encryption)
- DoXOR (XOR encryption)

Set the Default state to “Default”. Both states use the same Graph (“NetGraph”).

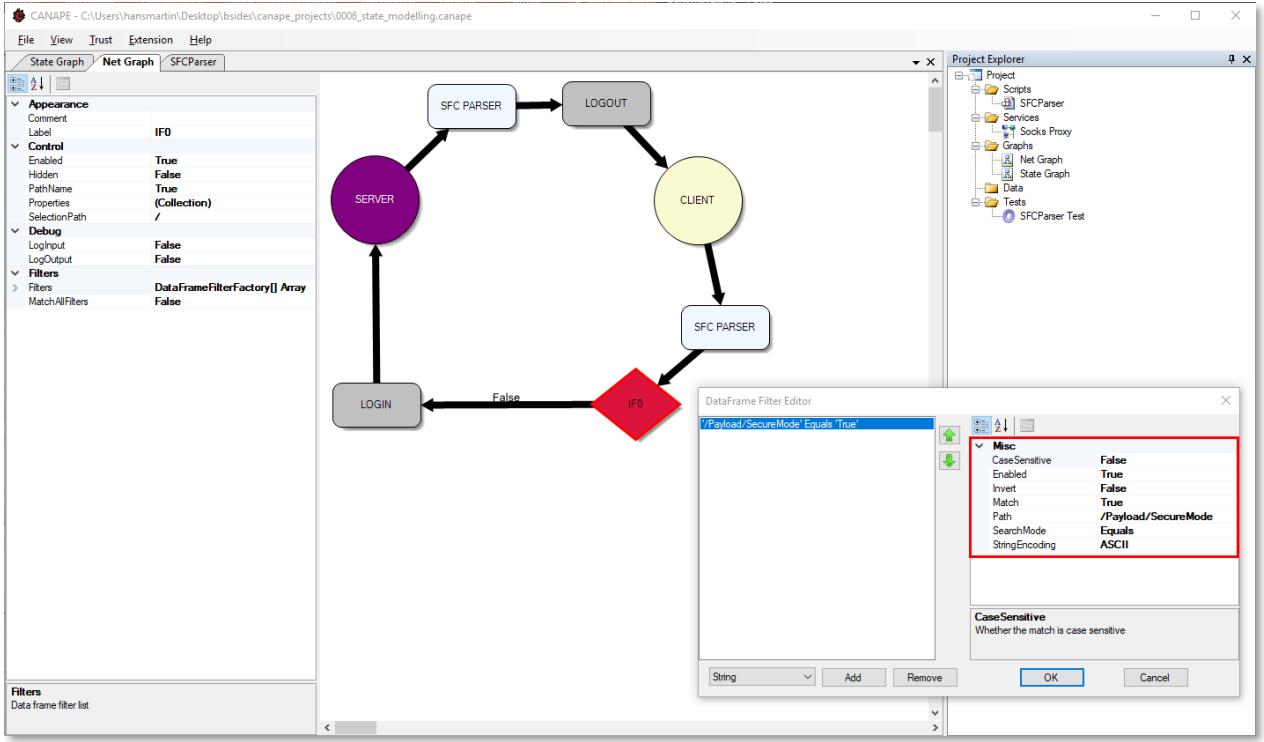


Adding the Decision Node

Add a Decision Node between the Parser- and the Log Node.

Please label the connection to the Log Node with “False”.

Add a filter to the Decision node. If the payload, filter for the secure mode instruction.



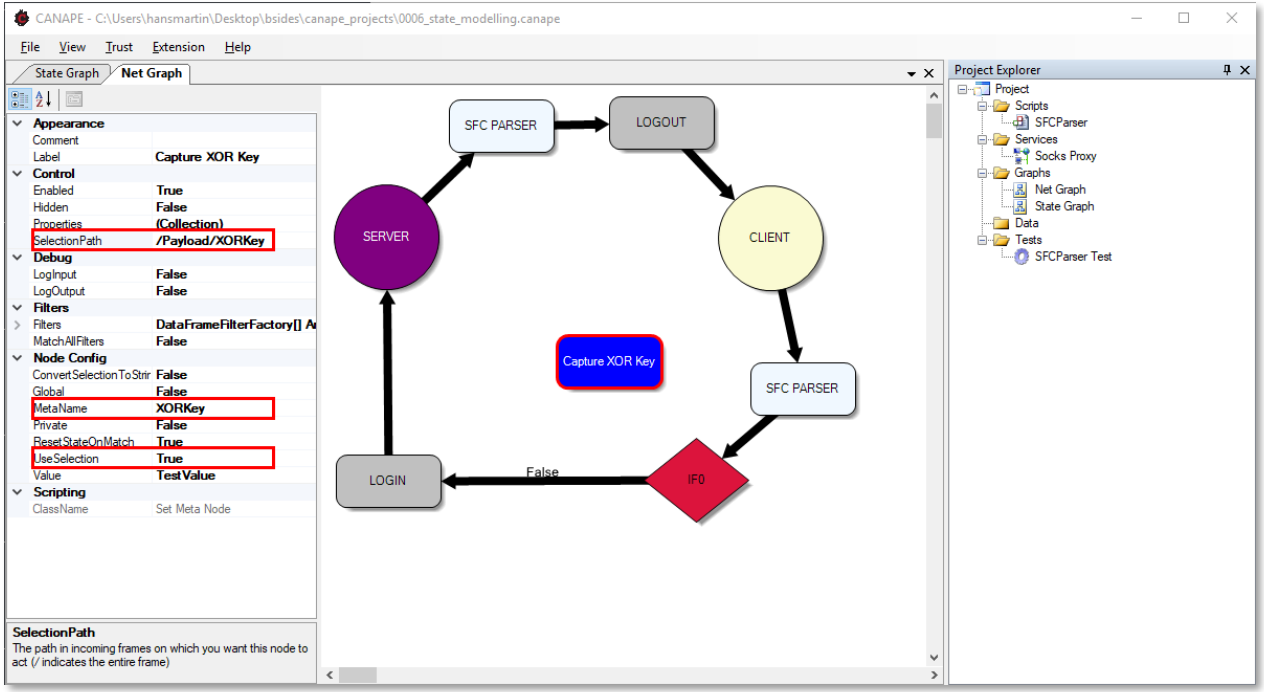
Saving the XOR key

In the Net Graph, add a new “Set Meta Node” from the Library. We will use this node to store the XOR key in the local meta storage.

Set the SelectionPath to

/Payload/XORKey

and the property “UseSelection” to true. Name the Meta value to “XORKey”.



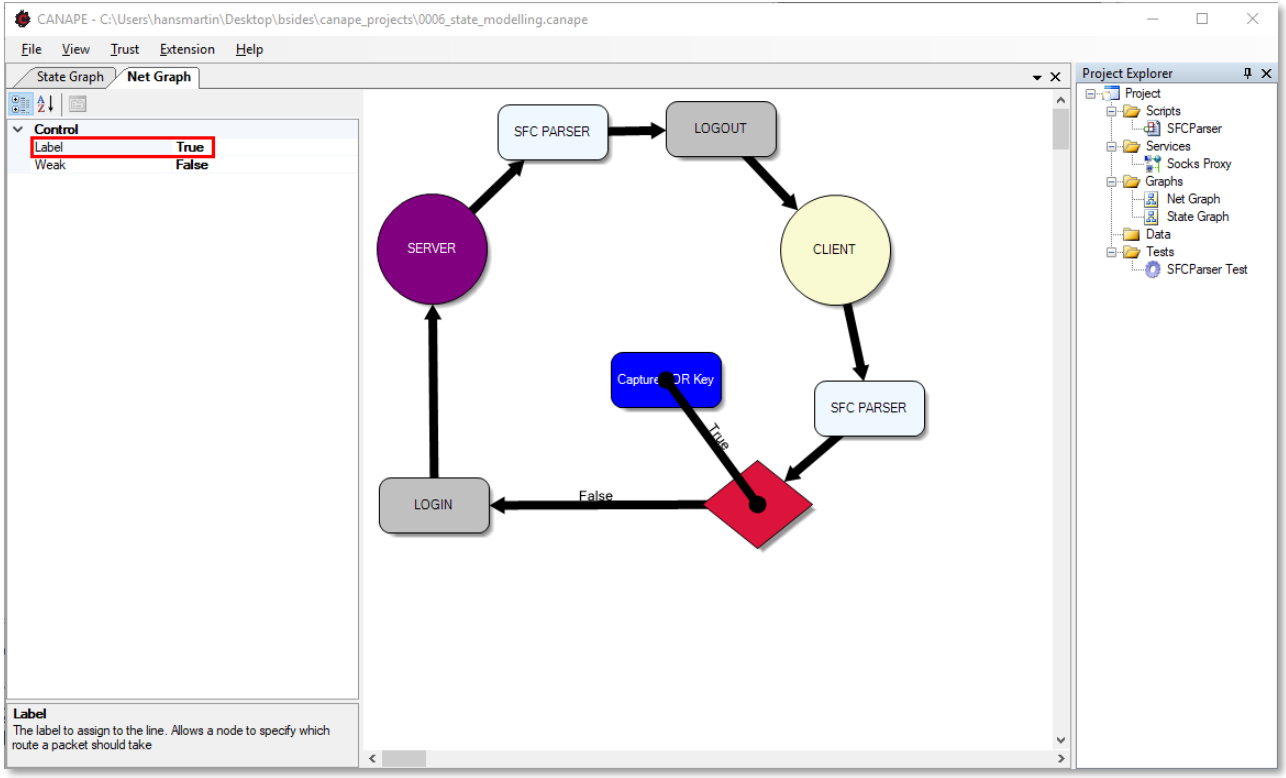
Connecting the nodes

Connect the Decision node with the “Capture XOR key” node. Label the line “True”.

Note:

Labeling this connection correctly is important as the decision node will use the Label for path identification.

For convenience, label the other connection “false”.

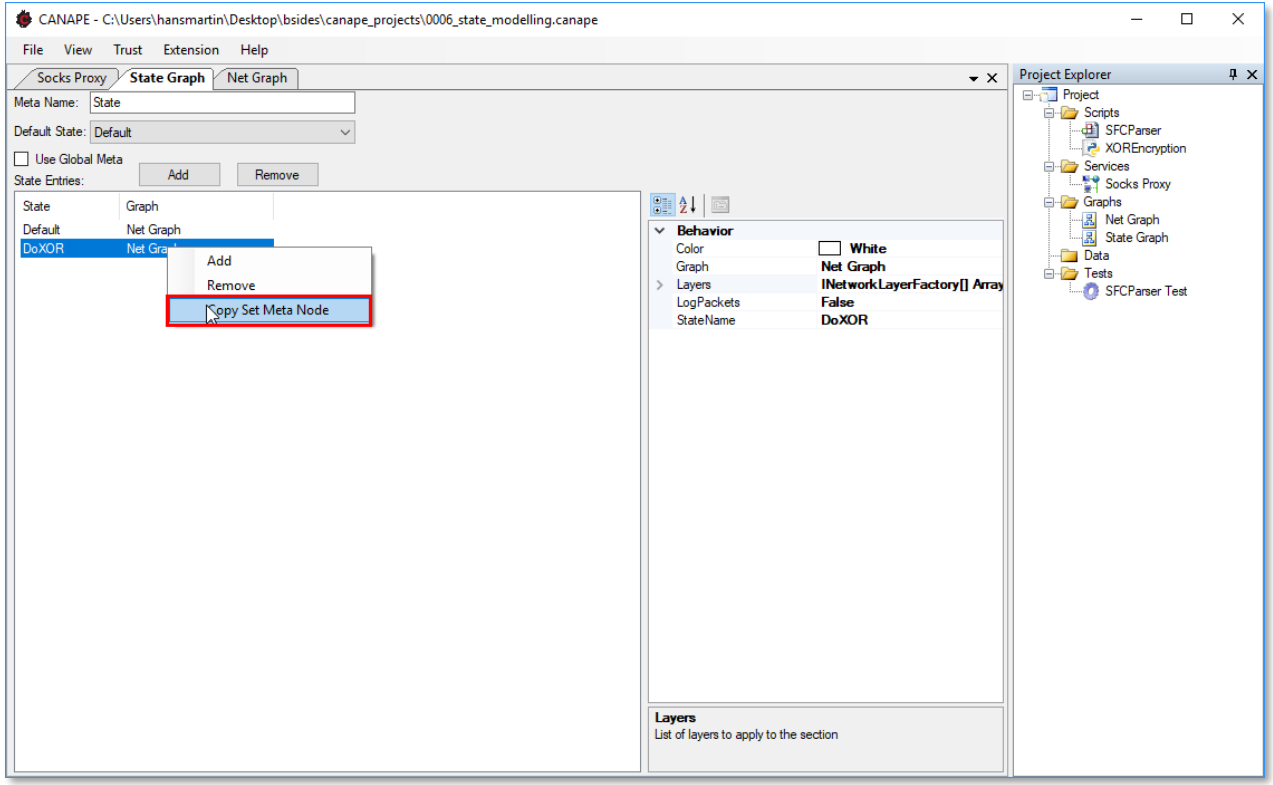


Creating a state node

Creating a node to change the state manually can easily go wrong,

Go back to the state graph and select the “DoXOR” state.

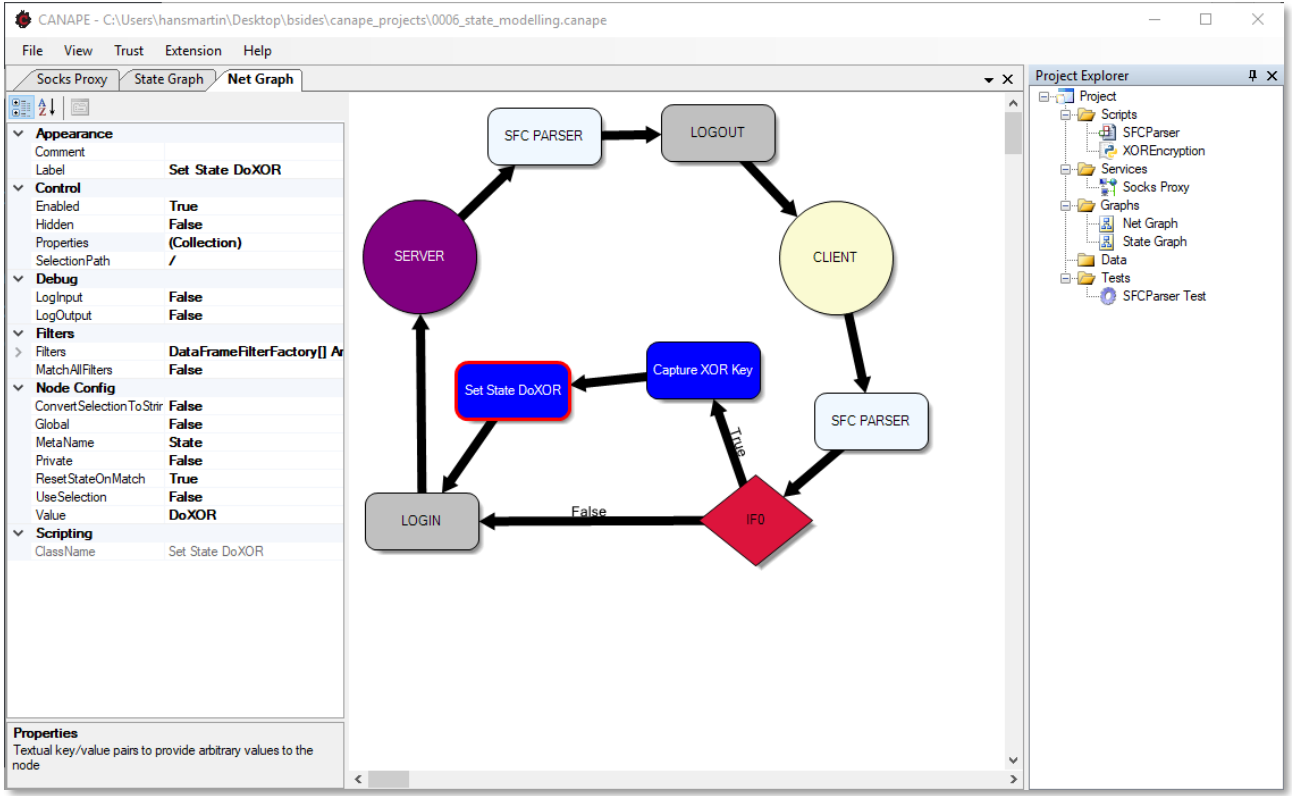
Select “Copy Set Meta Node” from the context menu (right mouse click)



Changing the state

Paste the “Set State” Node in the Net Graph.

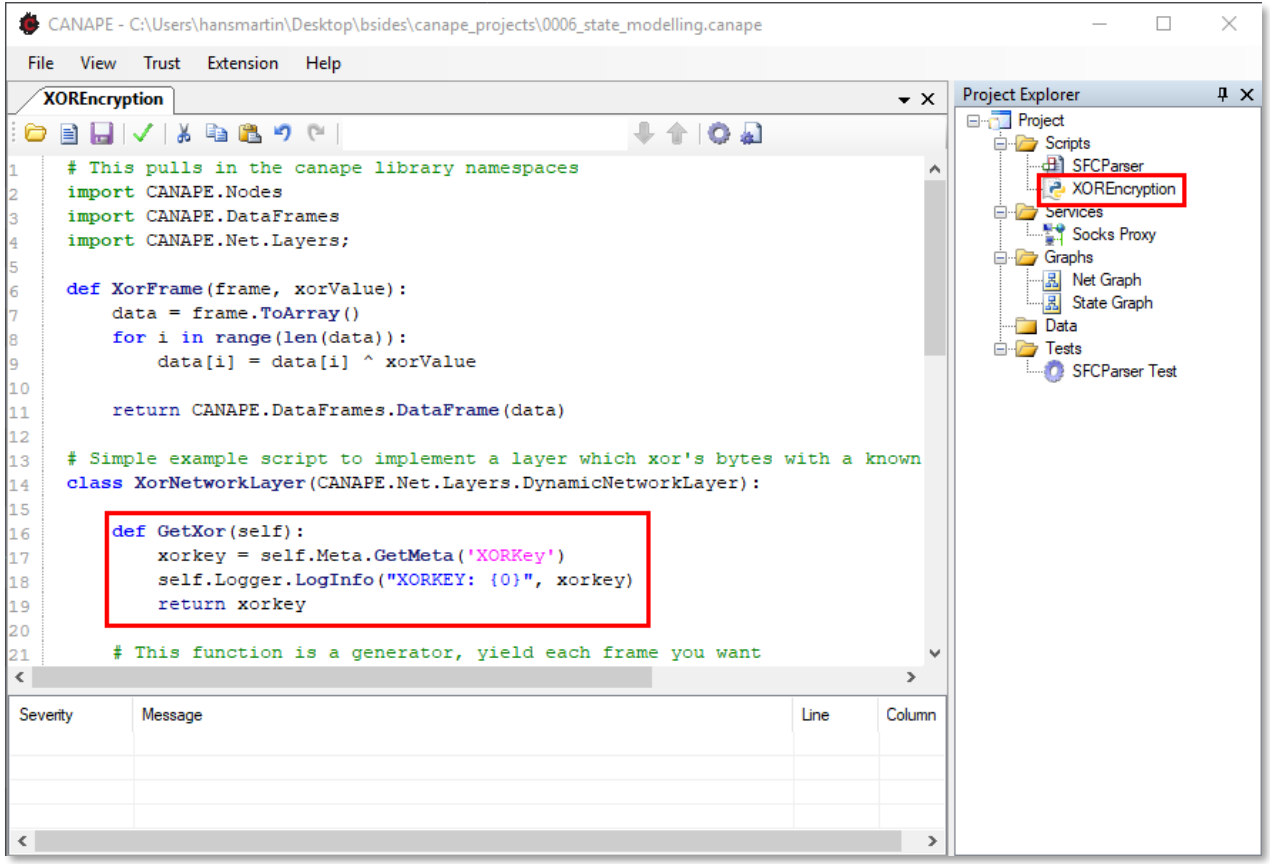
Connect the new node with the previous one (Capture XOR Key) and the LOGIN node.



Adding the Python script

Add a new empty python script in the scripts section and name it “XOREncryption”. Copy and paste the code from the “xor-encryption.py” script into the code editor.

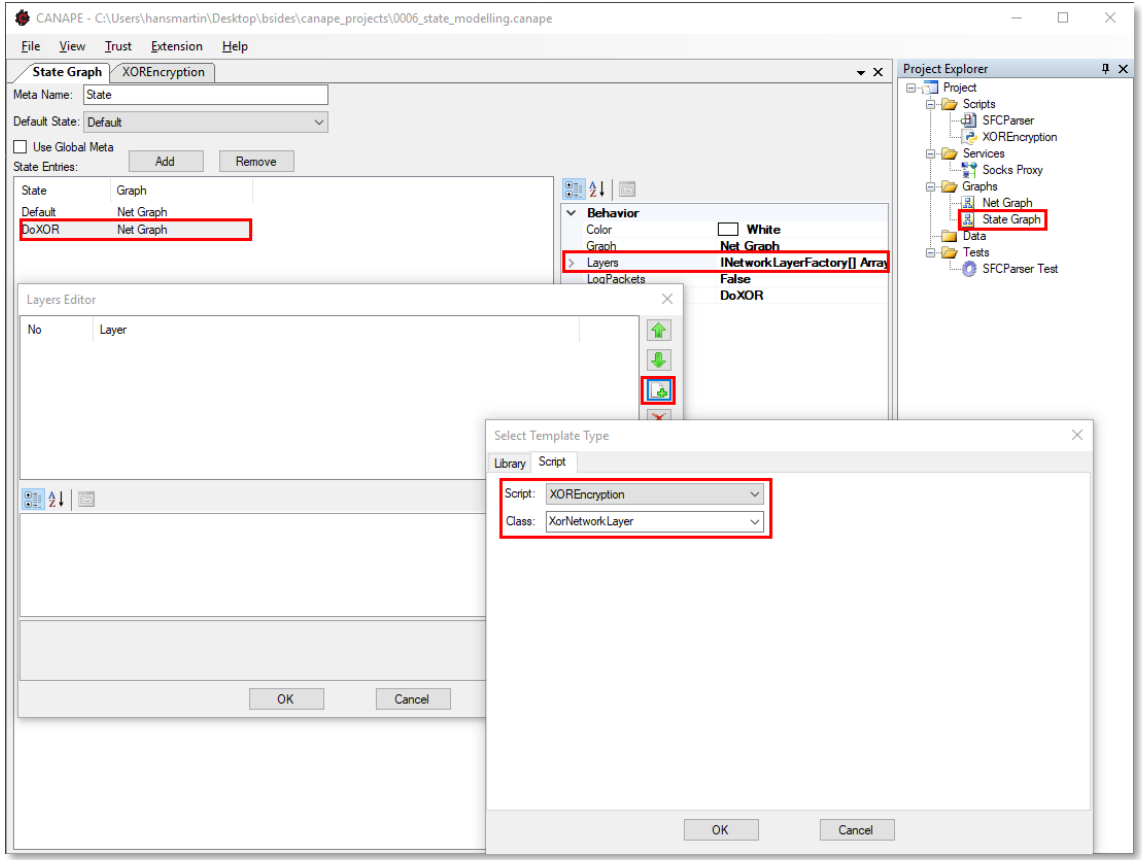
The highlighted code shows you how to access data from the Meta storage.



Adding the XOR layer

In the State Graph, add a new layer to the DoXOR graph.

Select the XOREncryption script and then the “XorNetworkLayer” class.

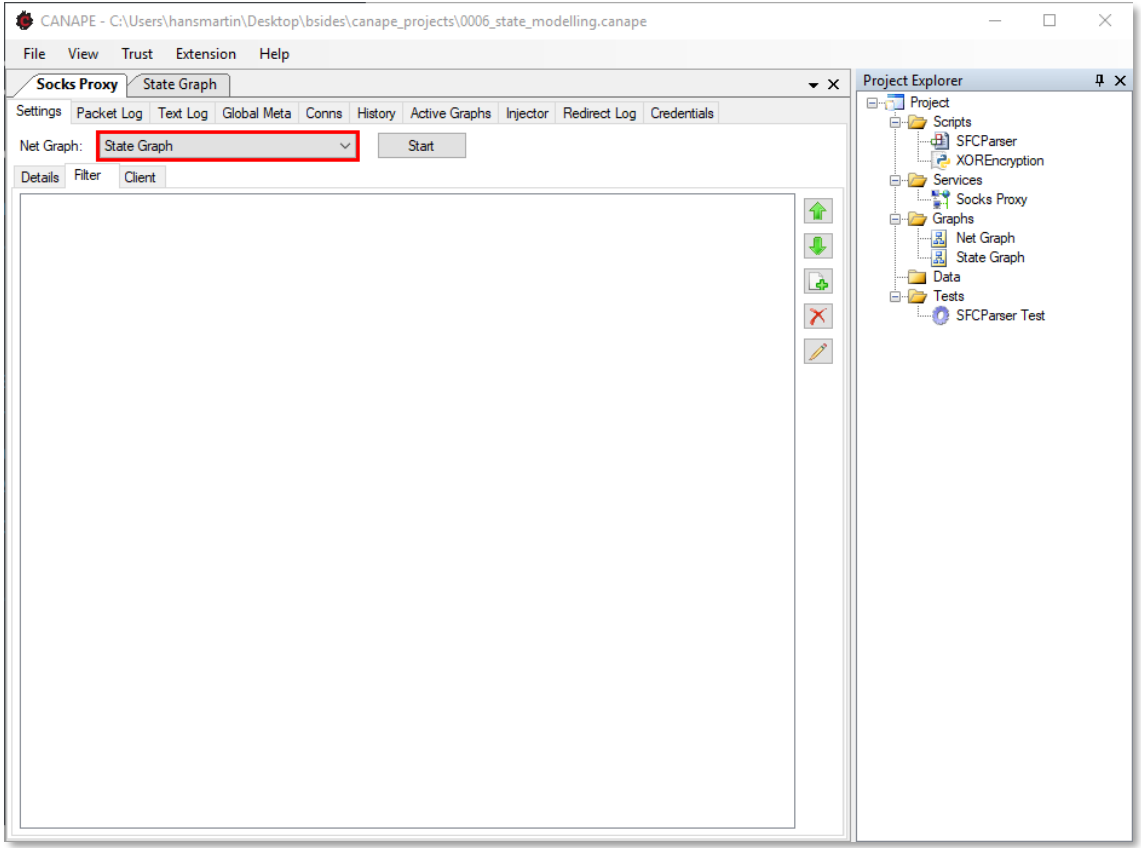


Changing the Net Graph

Change the default Net Graph of the Socks Proxy to the new State Graph.

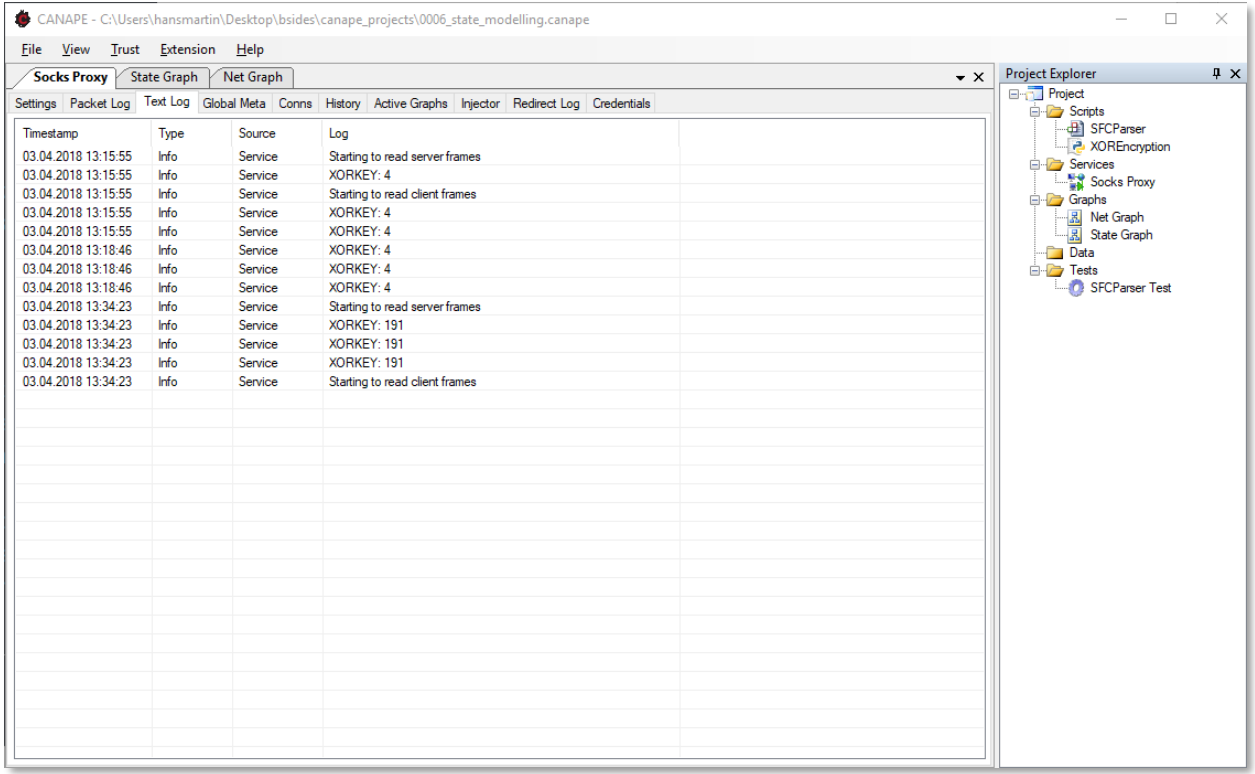
Important:

Please make sure that you remove the filter settings from the proxy, otherwise CANAPE runs into problems due the different filter settings in the proxy and the state graph.



Checking log entries

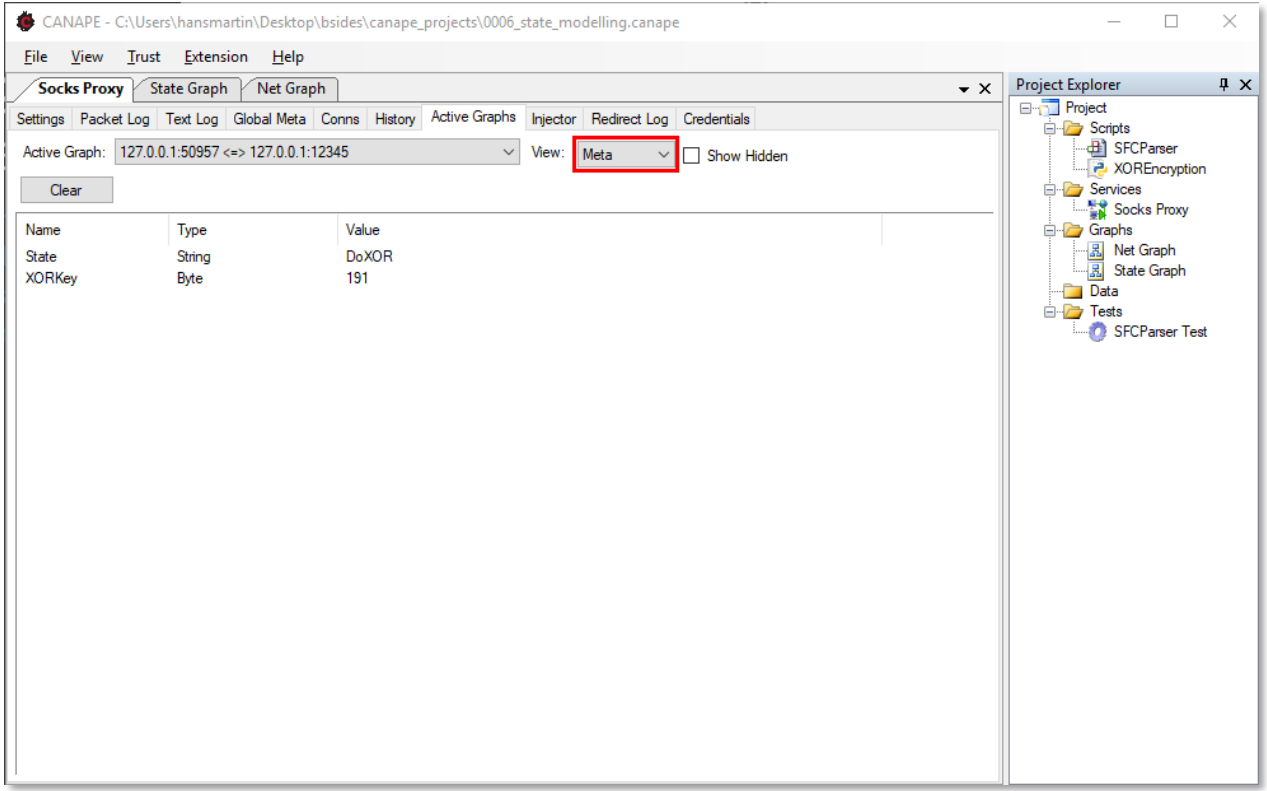
If everything works as planned you should now see the log entries from the Python script in the Text Log tab.



Inspecting meta data

You can also use the “Active Graphs” tab to inspect the meta data that CANAPE stores for each connection.

Just select the connection of your choice and switch to the “Meta” view.



Workshop 7:
Removing XOR encryption

8.

TRAFFIC INJECTION, MANIPU-LATION AND REPLAY

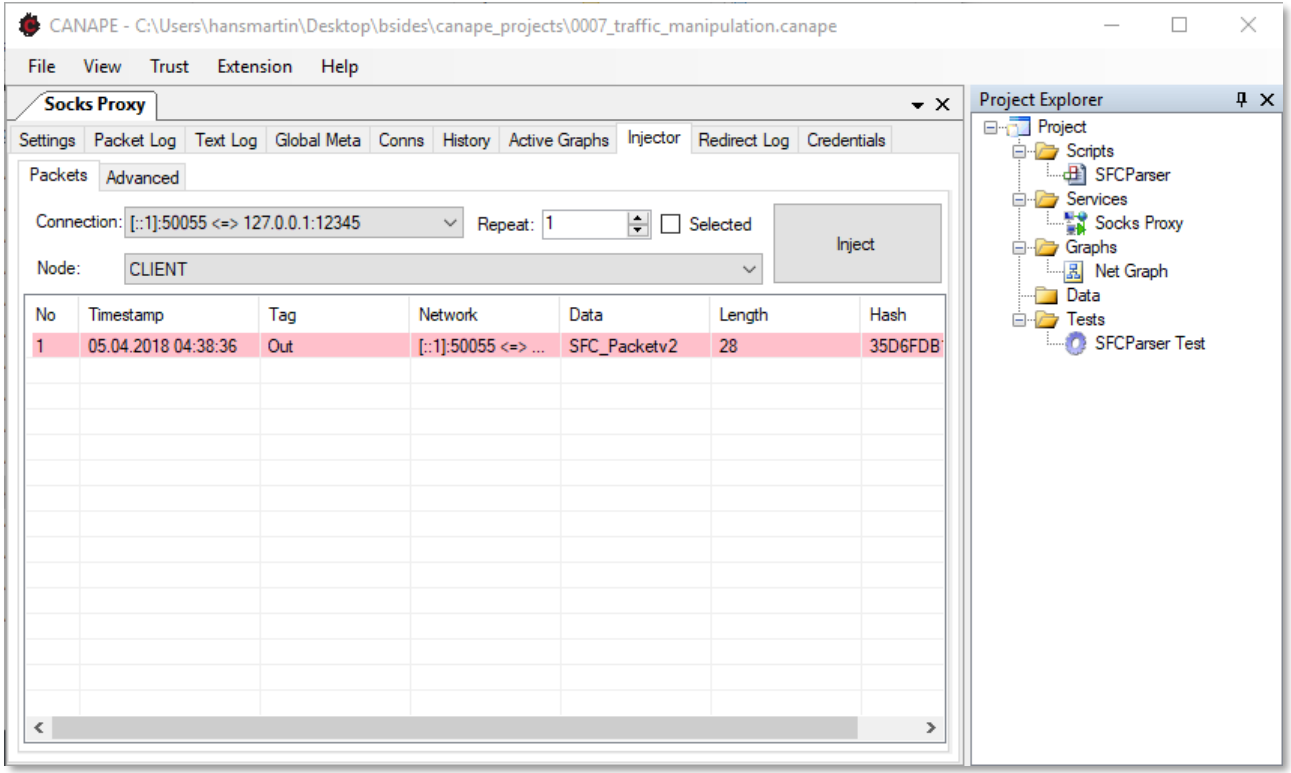
Let's change data...

CANAPE Injector

The network service provides an “Injector” tab that allows you to inject packets/data into existing network connections.

You can also select the node where the packet should be injected in the Net Graph. For example, you can avoid logging of the injected data by directly injecting the packet into the CLIENT node.

You can copy & paste packets to the injector.



Dealing with Checksums

Until now, we only looked at SuperFunkyChat packets but never changed them. To do that, we must deal with the checksum first. Here is the Python code to calculate the checksum.

```
1  def Calculate(command, payload):
2      chk = command
3      for b in payload:
4          chk = chk + b
5      return chk
6
```

We could write a script that recalculates the checksum/length field and place that in an additional dynamic node. However, the parser editor also provides some basic functions to do this.

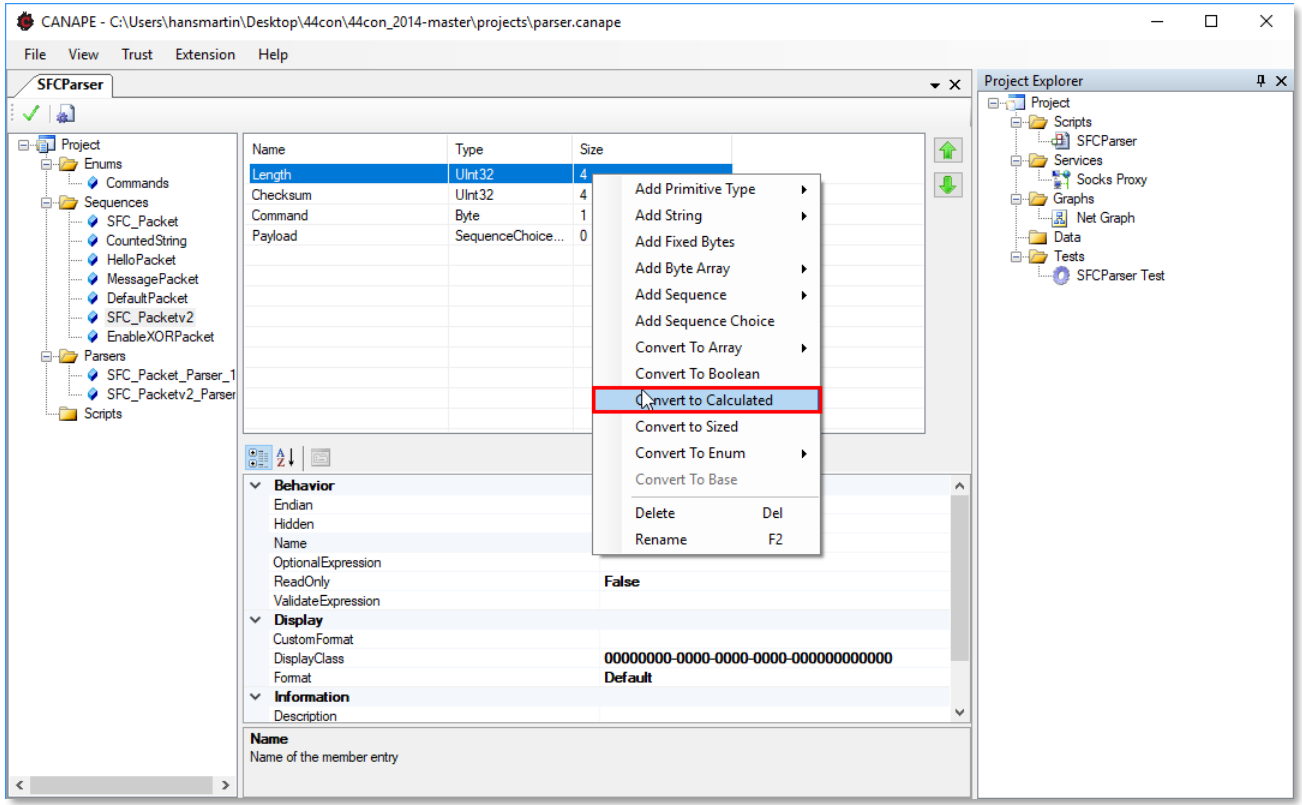
Calculated fields

The parser editor supports a Python-like syntax expression evaluator.

It allows you to recalculate the checksum based on other calculated values.

It is even possible to use python snippets like the one from the previous slide.

To enable this, we must first convert the Length- and Checksum field to “calculated”.

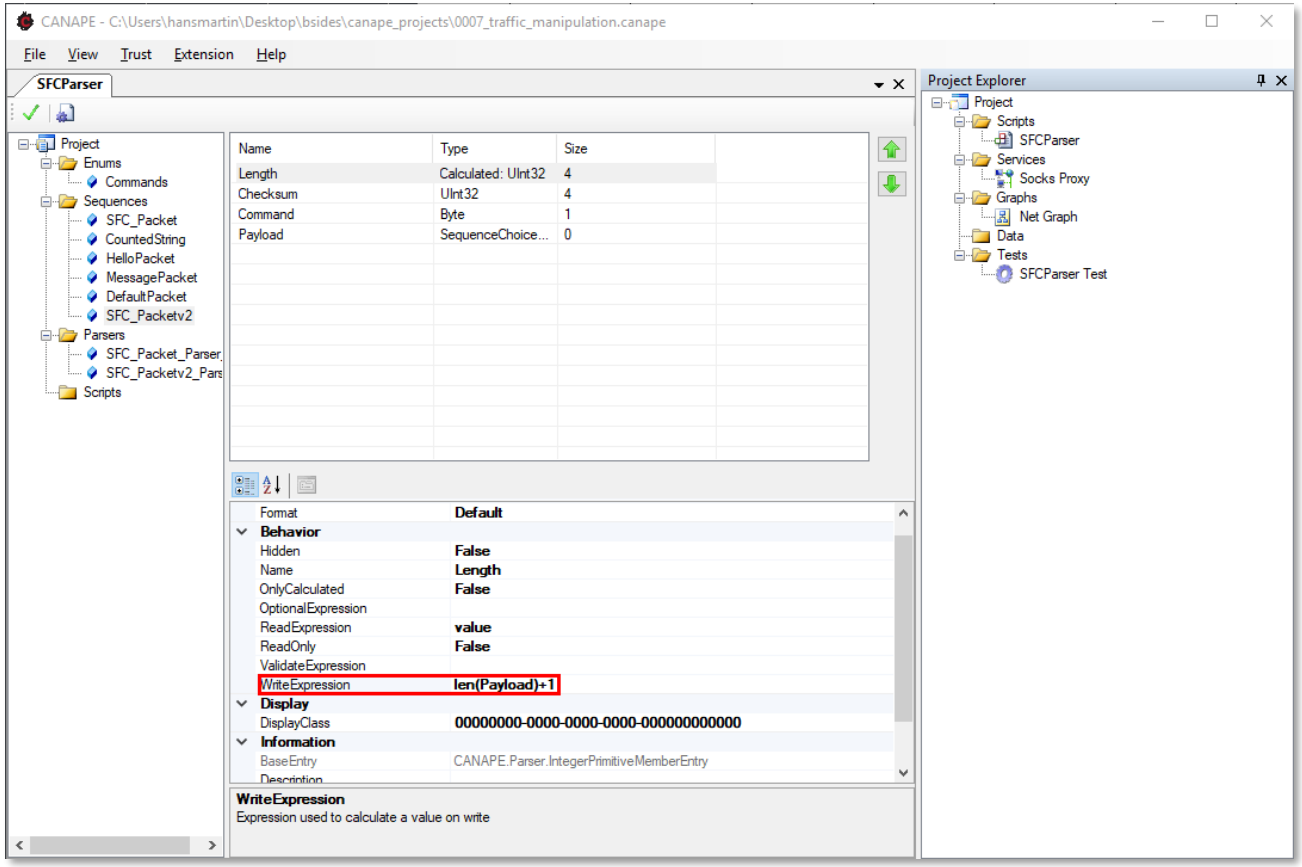


Calculating the length

The length can be calculated with a simple expression.

Place the following expression in the “WriteExpression” field:

len(Payload)+1

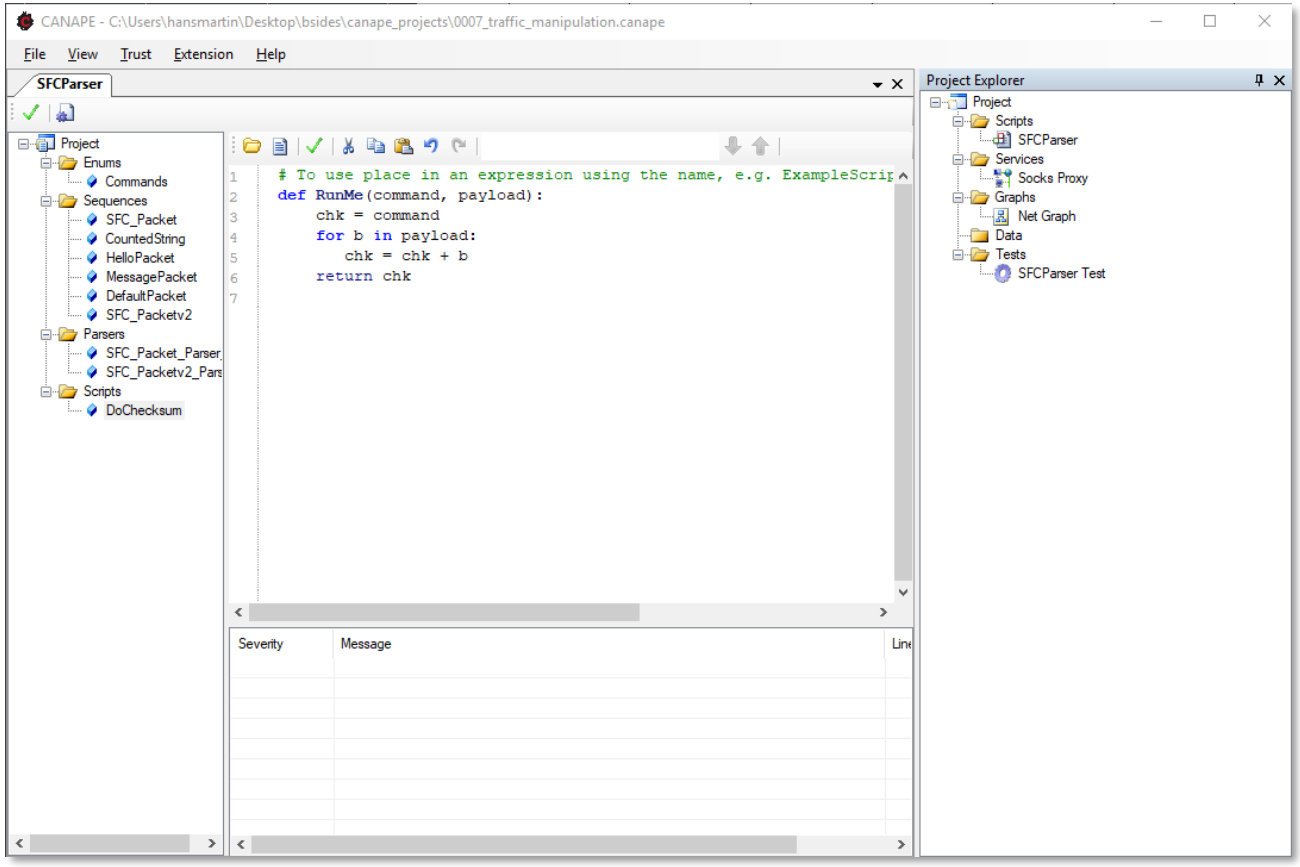


Checksum script

The checksum calculation is to complicated for a simple expression. Hence we use a Python script for that.

Use “Add Python Script” in the Parser Explorer to create the initial script.

Use the script validation to ensure that you didn’t introduce any “tab/space” bugs ☺

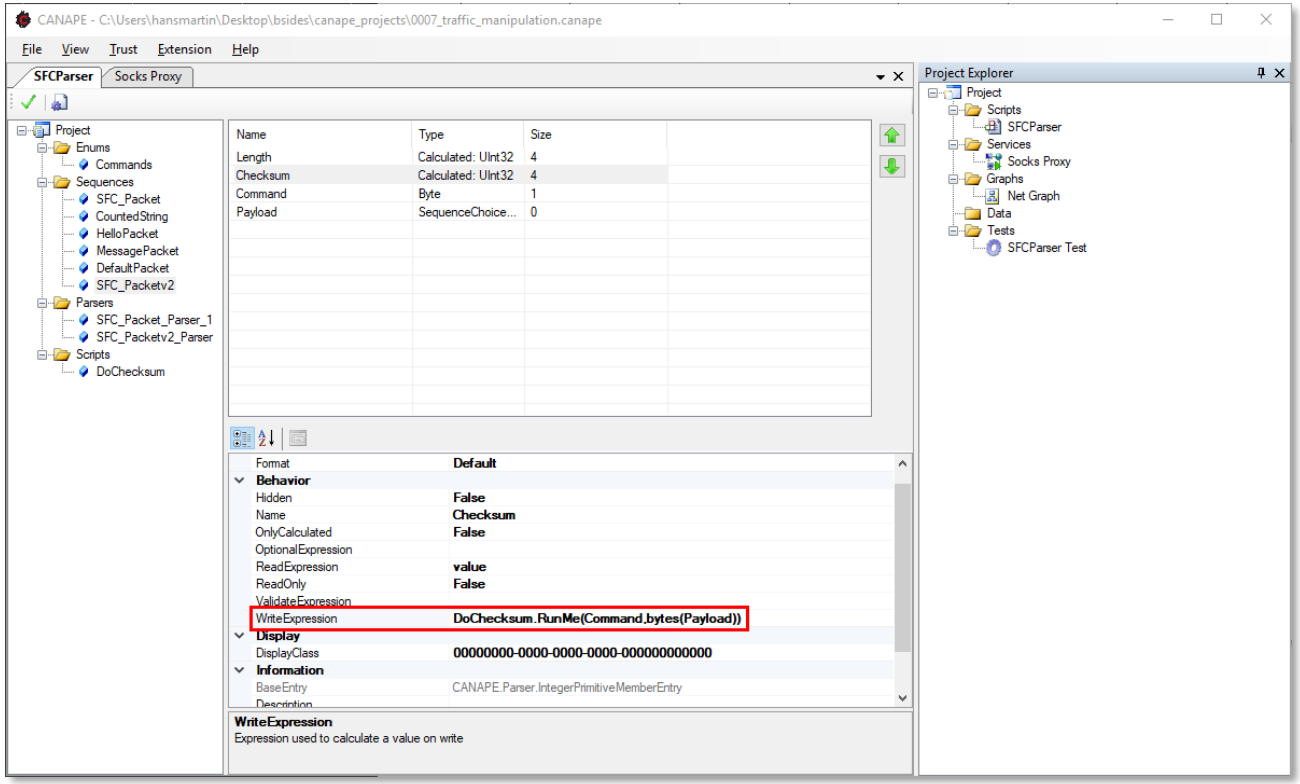


Checksum Write Expression

Convert the Checksum field to a calculated value.

You can call the added script from the WriteExpression as follows:

DoChecksum.RunMe(Command,bytes(Payload))



Workshop 8:
Packet injection/manipulation

9.

Developing network clients

and fuzz everything…

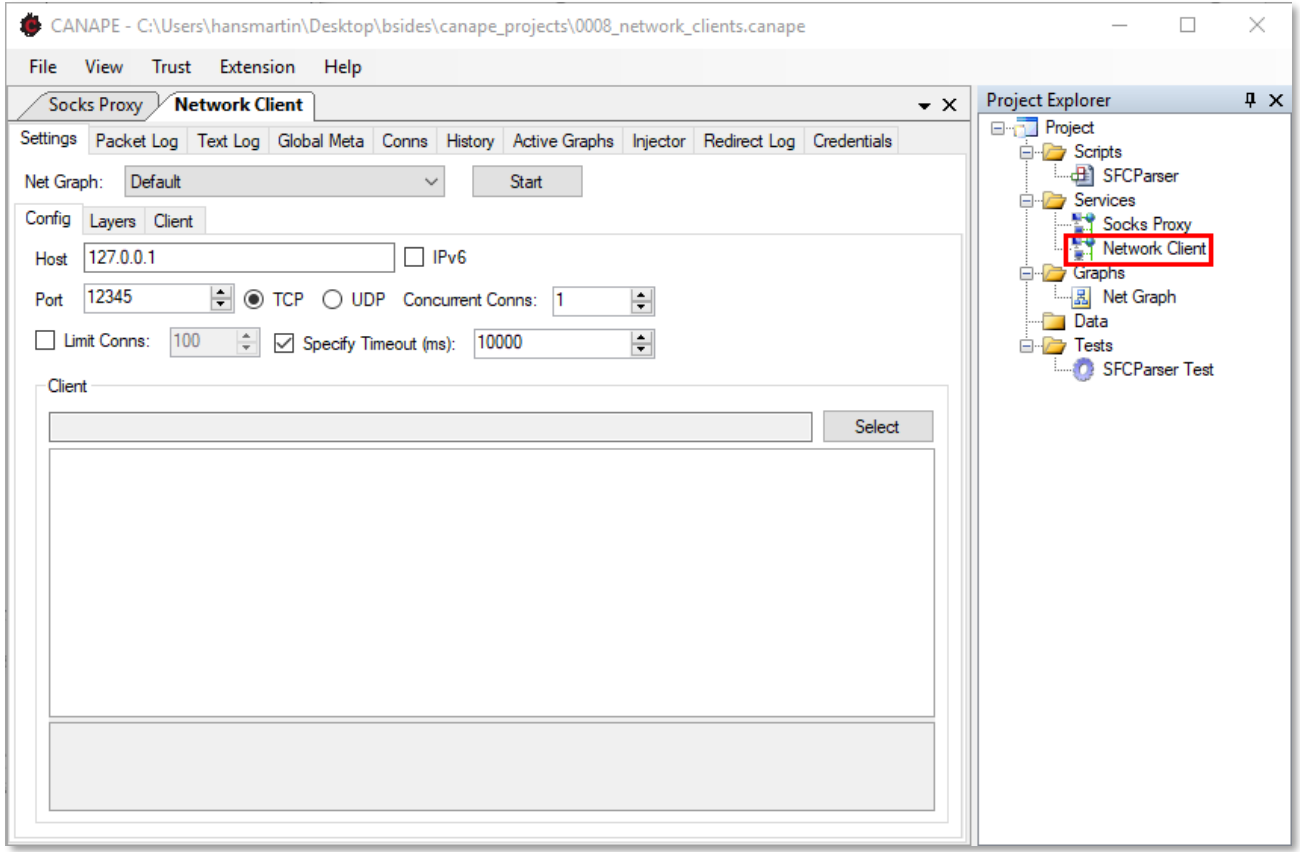
Network clients

CANAPE supports developing minimal network clients.

The process is designed to allow reuse of previous work from MitM sessions (graphs/scripts) with a minimal effort.

The client can also be used for basic fuzzing or other automated tasks.

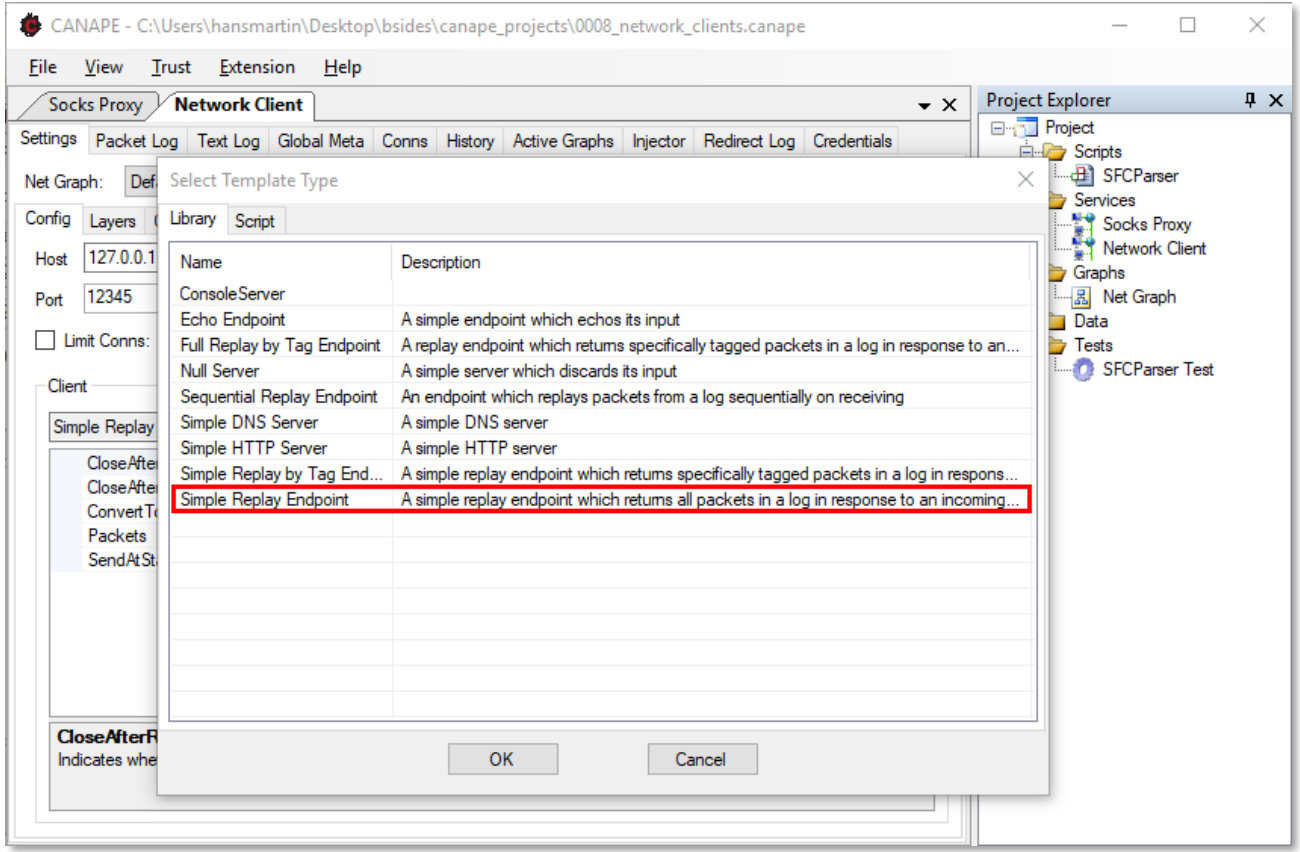
The tabs/options are quite similar to the service settings.



Client types

CANAPE provides several pre-configured client types, however you can also script your own client.

We will use a Simple Replay Endpoint here. This client “blindly” sends previously recorded network packets to the server, similar to the injector feature.

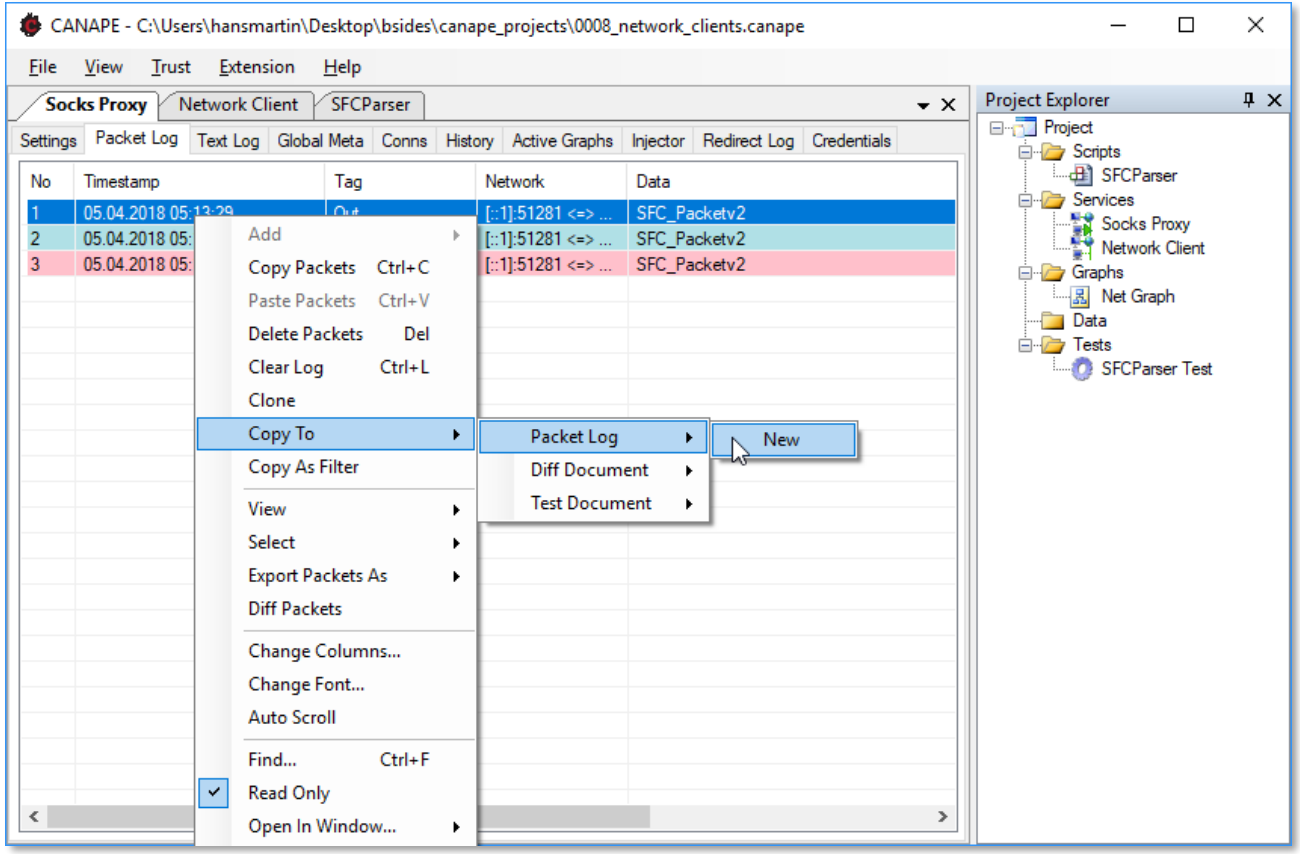


Saving packets

The client needs some packets to replay. You can get them from the logs of the socks proxy.

For a simple client, we need two packets:

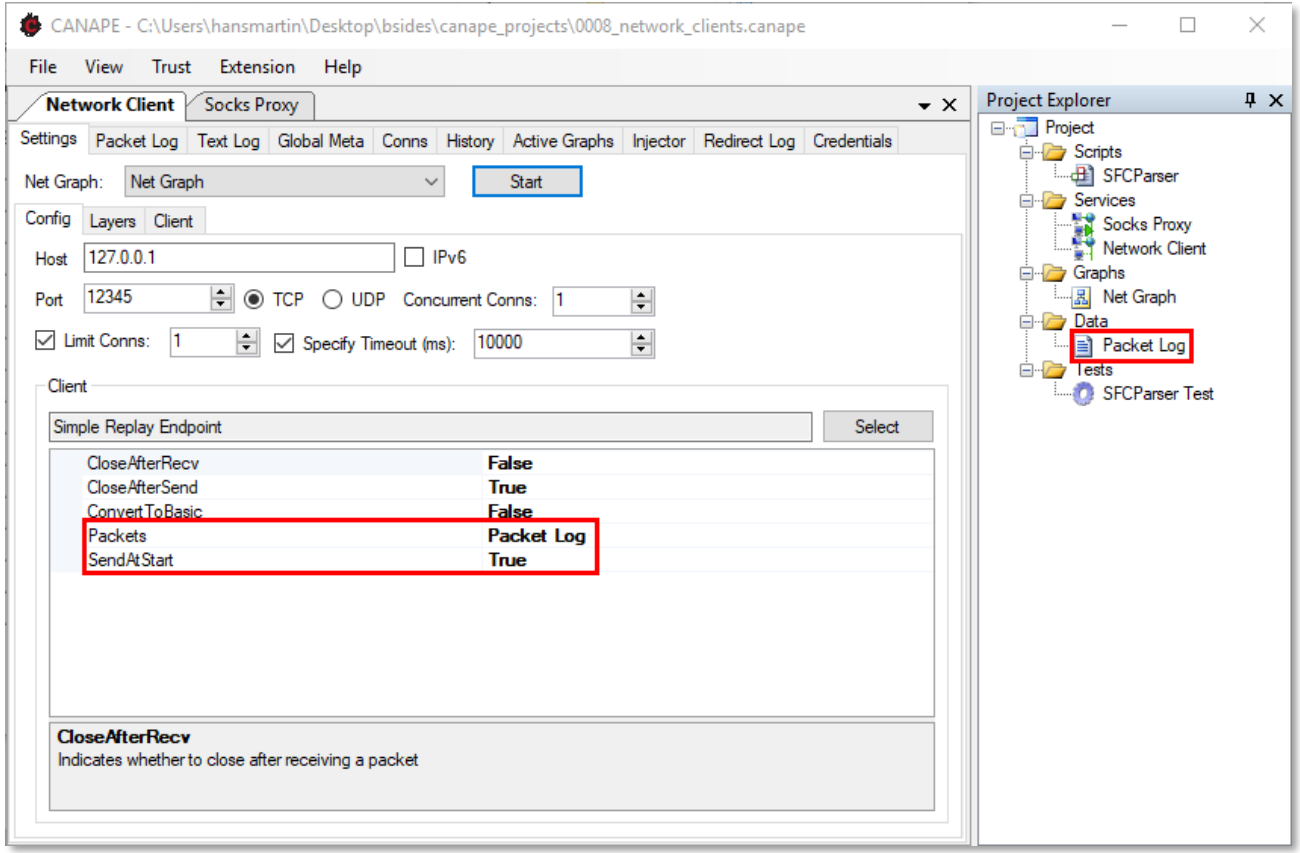
- HelloPacket
- MessagePacket



Client settings

Select the saved packets in the packets property. Also make sure that SendAtStart is set to “true” as we initiate the connection.

You should also limit the number of connections to a smaller number ☺



A basic command fuzzer

As described previously, every SFC packet contains byte that defines the actual command. A byte can have 255 different values, making it an ideal candidate for a very simple fuzzer.

What we need to do:

- Store a counter for the current command in the State
- Create a Python script that increases replaces the command number in our reference packet with the number from the counter. Then we increase the counter.
- Create a dynamic node that calls our fuzzing script
- Use a client to send 255 packets

Fuzzing script

Here our basic fuzzing script (from James). You can find it in the “misc” folder

CANAPE provides a basic counter function (“IncrementGlobalCounter”) that increases the counter for us.

Additional functions can be found in the NetGraph.cs source at GithHub:

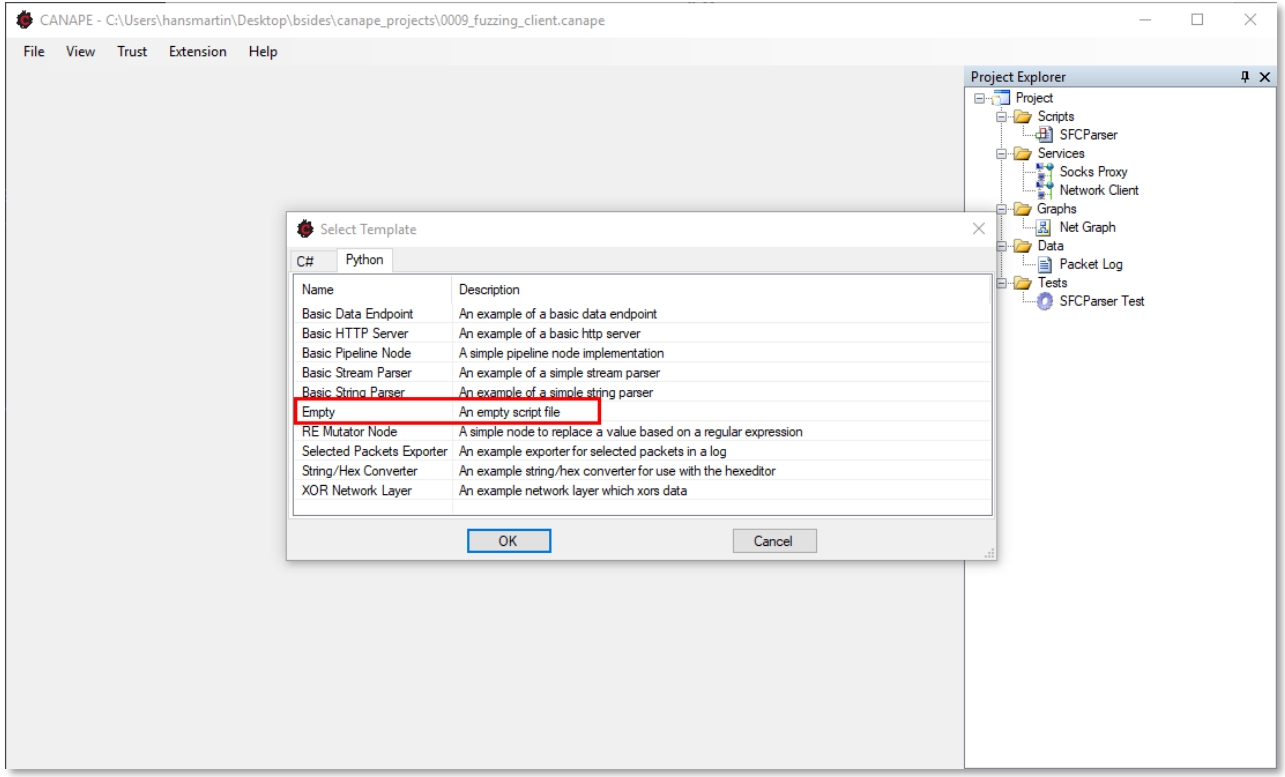
https://github.com/tyranid/canape/

```
# This pulls in the canape library namespaces
import CANAPE.Nodes
import CANAPE.DataFrames

# Simple pipeline node
class PipelineNode(CANAPE.Nodes.BaseDynamicPipelineNode):

    # Called when a new frame has arrived
    def OnInput(self, frame):
        cmd = self.Graph.IncrementGlobalCounter("CurrentCommand", 1)

        self.LogInfo("Current Command {0}", cmd)

        # XPATH Method
        #cmdval = frame.SelectSingleNode("/Command")
        #cmdval.Value = cmd

        # Dynamic Method
        frame.DynamicRoot.Command.Value = cmd

        self.WriteOutput(frame)


```

Python script

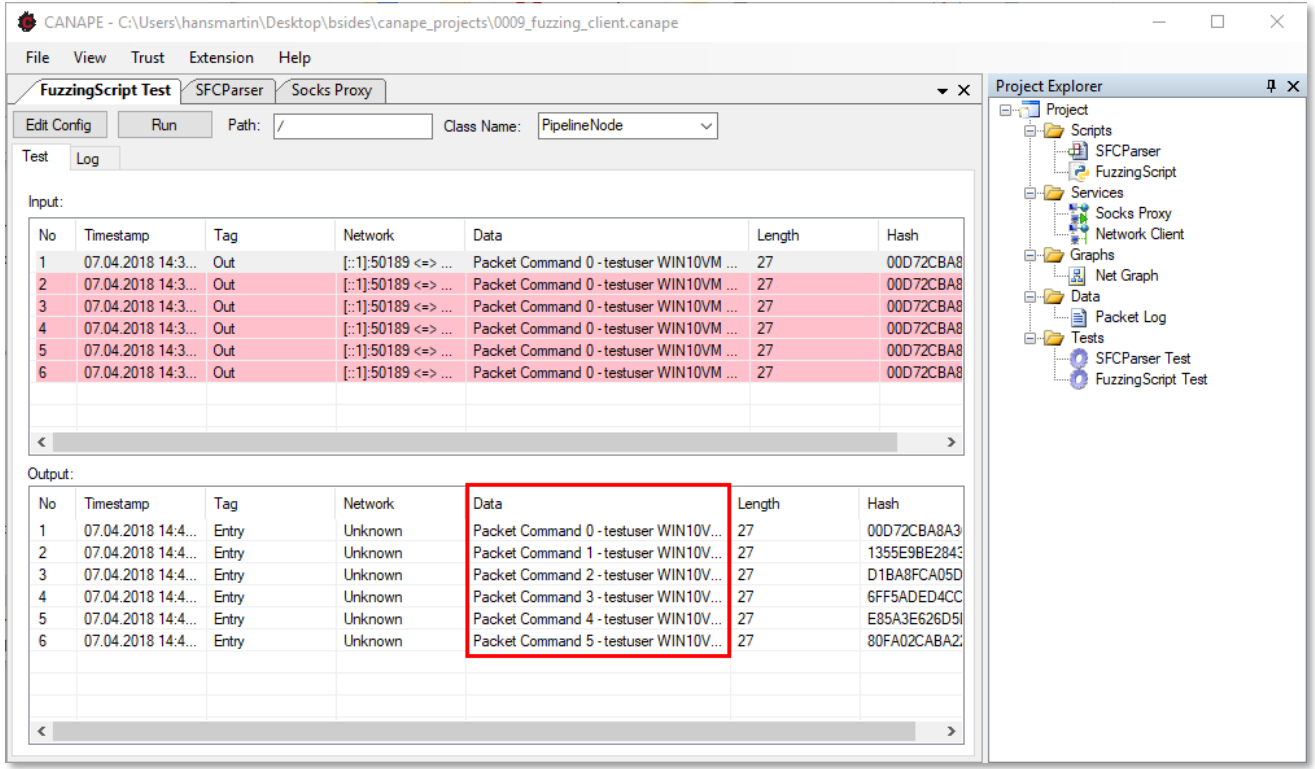
Create a empty Python script and name it “FuzzingScript”.

Copy & Paste the python code of the fuzzer into the script editor.



Fuzzer test

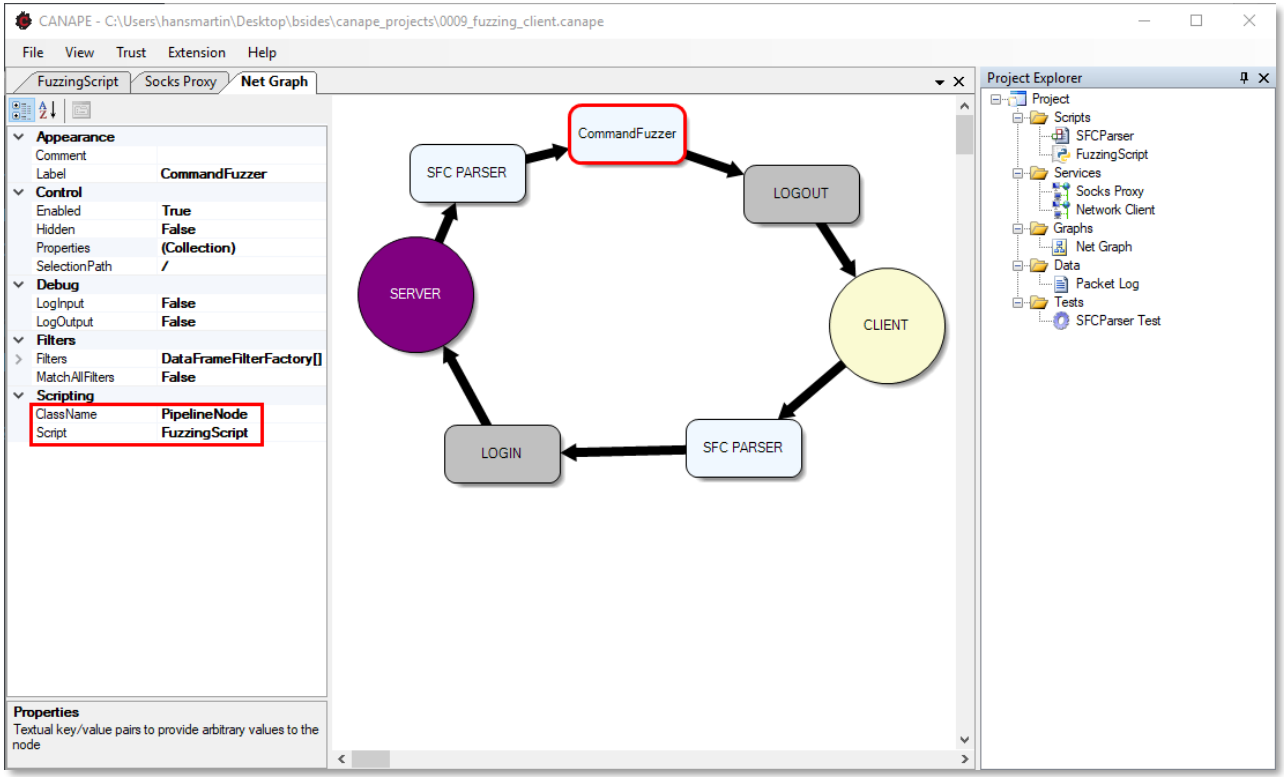
Create a new test case for the fuzzing script. Copy multiple instances of the “HelloPacket” from the proxy logs into



Fuzzing node

As the fuzzer is implemented in Python, we use a dynamic node to call the code from our Net Graph.

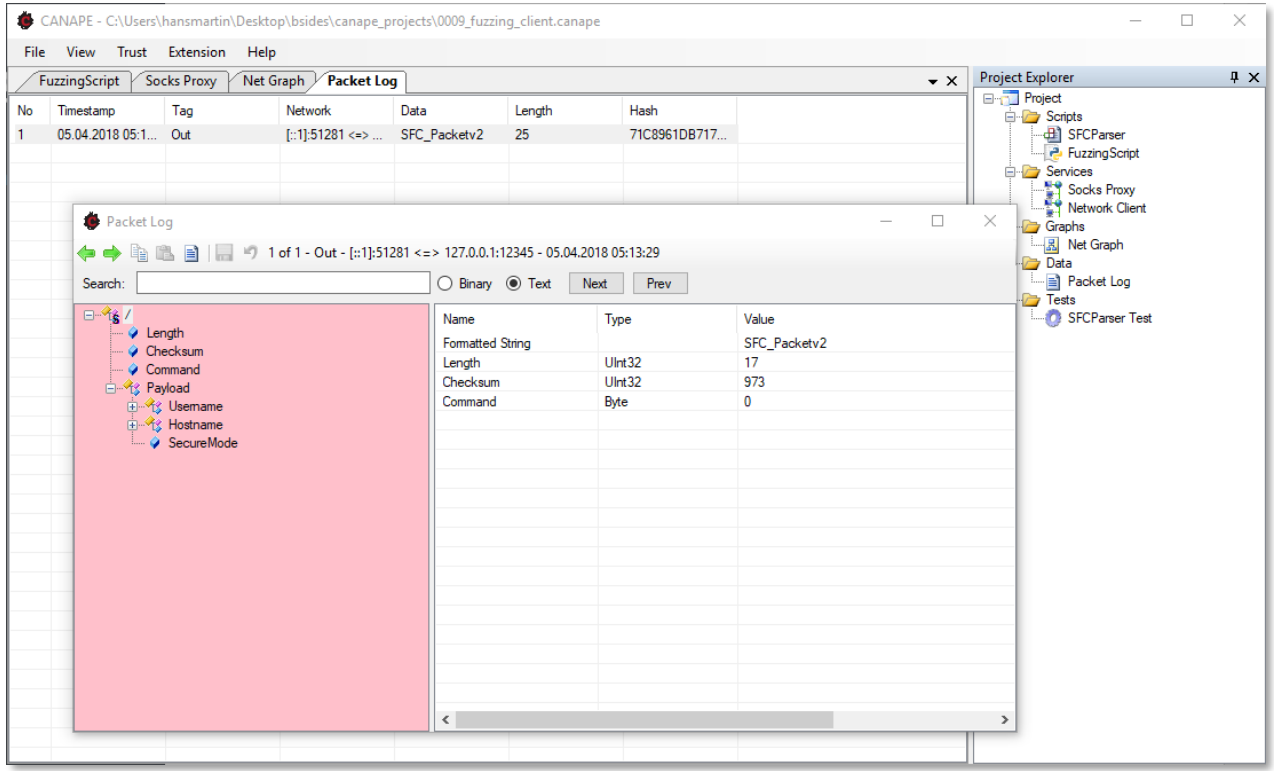
This time, we only include the node for data that we send to the server, as we don't intend to fuzz the client.



Packet log

We will use the previously captured packets from the “Packet Log” dataset as fuzzer input.

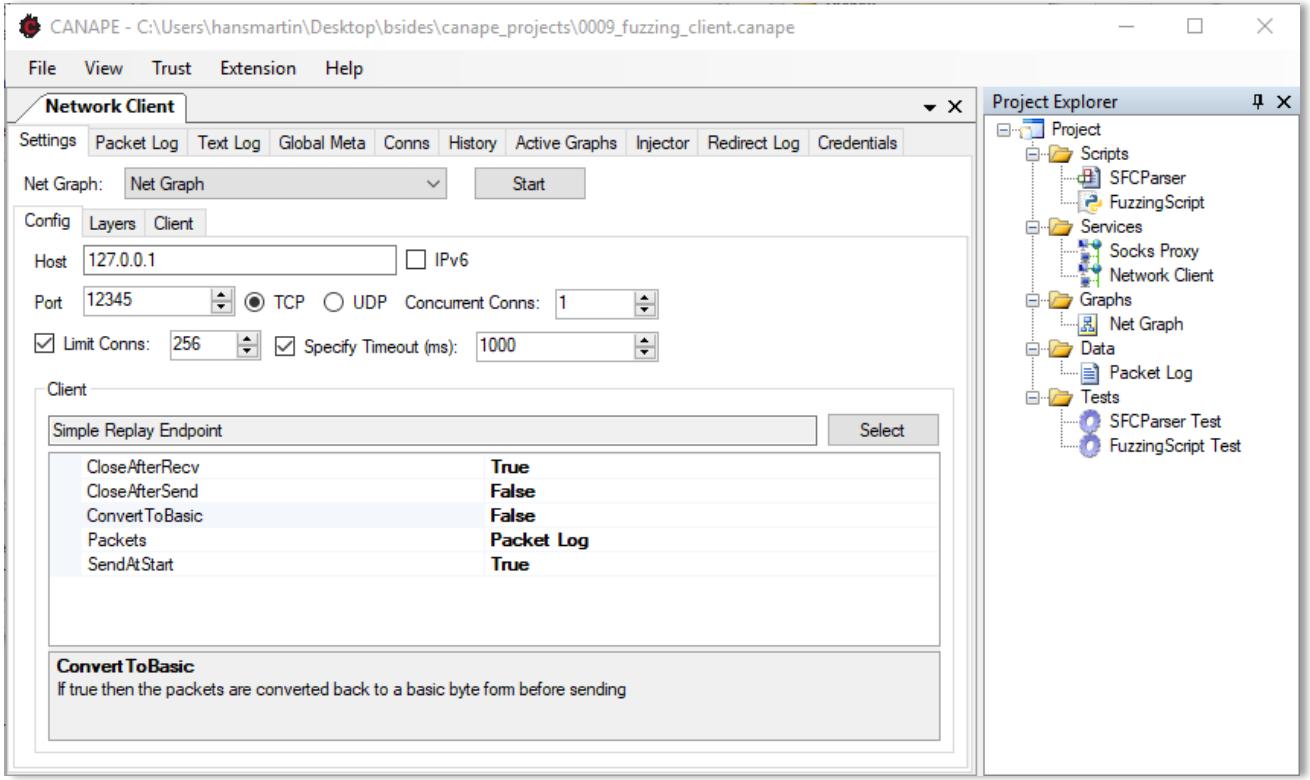
However, we only need one “HelloWorld” packet, which we will send multiple times.



Connection settings

We will use the previously captured packets from the “Packet Log” dataset as fuzzer input.

However, we only need one “HelloWorld” packet, which we will send multiple times.

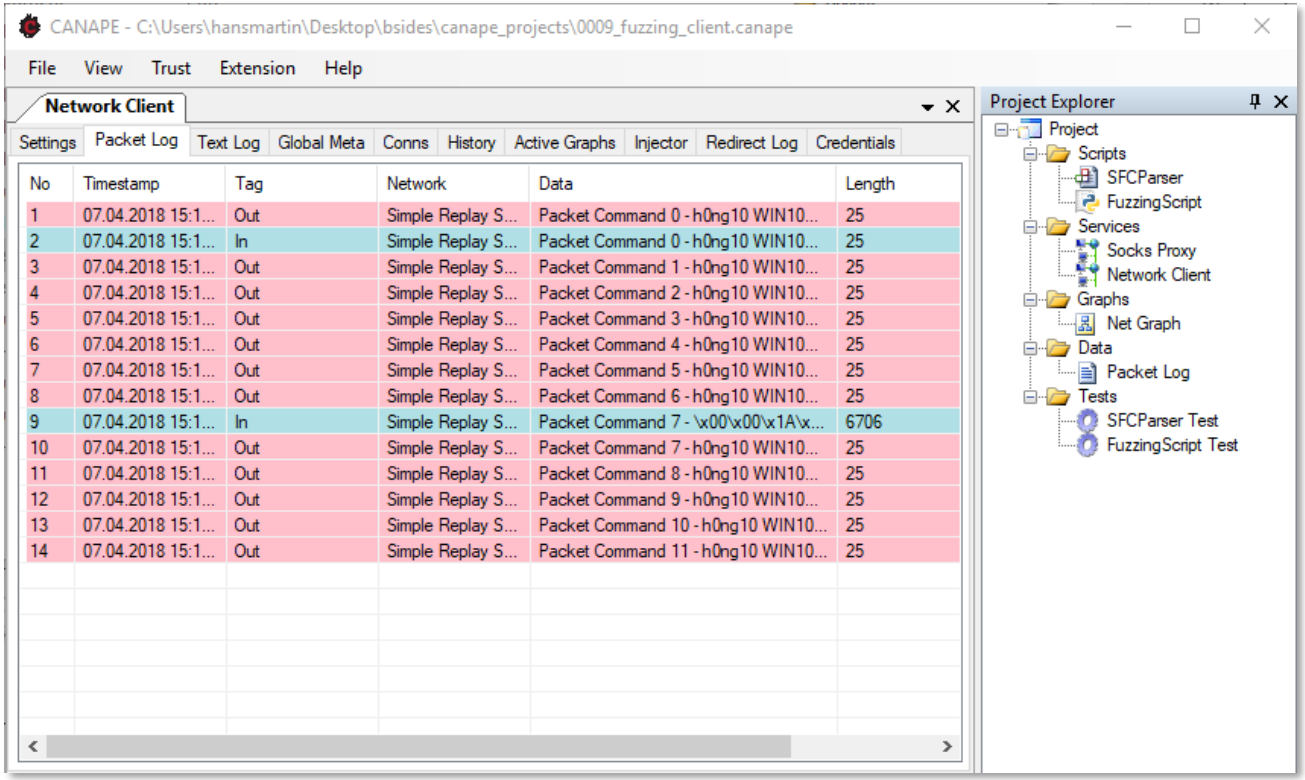


Fuzzing log

You can see your fuzzing results in the Packet log from the network client. Also check the text log for messages from the fuzzing script.

As you might see, fuzzing the command byte actually brought some additional results. You can even discover a hidden flag here.

Hint: Check if the server actually responses to your requests. If not, set the counter in the Meta tab and restart the fuzzer!



Workshop 6:
Creating a basic client / command fuzzer

Roundup

Some last words…

Thank you very much for your time

If you have any questions about me, CANAPE or this workshop please don’t hesitate to contact me at:

Twitter: @h0ng10
hmm@mogwailabs.de
https://mogwailabs.de

Links and references

Some source code:

- CANAPE source at GitHub
- James Forshaws original material from the 44con workshop

Security conference videos recordings:

- 29C3: ESXi Beast (EN) – Analyzing ESXi with CANAPE
- Blackhat 2012 EUROPE – CANAPE Bytes your Bits
- RuxCon 2013 Examination of the VMWARE ESXi Binary Protocol Using Canape

Picture References

These slides contain multiple free images from the page “unsplash” site

https://unsplash.com

The theme is based on a free PowerPoint template from Slides Carnival:

https://slidescarnival.com

Red bug by https://unsplash.com/@ritchievalens
Hello by https://unsplash.com/@lemonvlad
Capture by https://unsplash.com/@elliotteo
Look by https://unsplash.com/@ratushny
Traffic by https://unsplash.com/@mischievous_penguins
Pastels pot by https://unsplash.com/@dhudson_creative
Kids draw by https://unsplash.com/@aaronburden
Books by https://unsplash.com/@erol
Book library by https://unsplash.com/@zeak
Workbench by https://unsplash.com/@ugmonk
Tools by https://unsplash.com/@yer_a_wizard
Vatican museum by https://unsplash.com/@nhoizey
Vatican museum by https://unsplash.com/@jadlimcaco
Pottery by https://unsplash.com/@krysalex
Pottery by https://unsplash.com/@quinoal
Controller by https://unsplash.com/@pawelkadysz
Drone pilot by https://unsplash.com/@wilcovanmeppelen
Airplane by https://unsplash.com/@hharvey
Airplane by https://unsplash.com/@ryanrichards
Notebook by https://unsplash.com/@faceline
Thanks by https://unsplash.com/@hannynaibaho