Extending Bro Covert Channel Detection (BroCCaDe) with New Plugins

Hendra Gunadi (Hendra.Gunadi@murdoch.edu.au),
Sebastian Zander (s.zander@murdoch.edu.au)

7 December 2017

Abstract

This report is a tutorial on how to write new Feature Extraction plugins for Bro Covert Channel Detection (BroCCaDe). The report also briefly discusses how to extend the current Analysis plugin. In each section, we detail how to implement the plugin, and then show the code of a simple working prototype which follows the implementation steps.

1 Introduction

In technical report [5], we presented the design of BroCCaDe. In this report we will go into the details on how to write additional Feature Extraction plugins including a discussion on the basics of writing Bro plugins. Our main focus is on implementing feature extraction plugins for BroCCaDe and as part of that we will also explain how to write protocol parser code needed in case there are some features that Bro does not provide directly, e.g., writing a plugin to extract the value of the TCP urgent flag and urgent pointer. We will also briefly discuss how to extend the Analysis plugin.

For our initial case study, we will use Ping Tunnel [2], a covert channel embedded in ICMP echo payload. Ping Tunnel falls under the category of reserved/unused pattern in Wendzel et al. [6]. We use Ping Tunnel as our case study as it is a simple channel and we already have the existing ICMP protocol parser shipped with Bro. However, we also provide an example of another plugin which extracts the value of the TCP urgent flag and urgent pointer. This tutorial is written for Linux Ubuntu 16.04. Some adjustments may be needed when following the steps in another environment. Section 2 describes in detail how to construct a Bro plugin from scratch (with the help of "init-plugin"). This method of constructing a plugin is applicable to every plugin used by BroCCaDe except the Analysis plugin. The analysis plugin has a different infrastructure and extending it with a new analysis metric requires a different approach which is detailed in Section 3. Section 4 concludes the report.

2 Writing Plugin

Generally, there are 5 steps taken to create a plugin:

1. Create the plugin skeleton (Section 2.1).
2. Write the core of the plugin (Section 2.2). Section 2.2.1 details how we can create a feature extraction plugin and Section 2.2.2 details how we can create a plugin to parse incoming packets.
3. Add built-in functions and events declaration (Section 2.3).
4. Write the script which makes use of the built-in function and events (Section 2.4).
5. Build the plugin and make Bro recognize the plugin (2.5).

In the following sub sections we describe the different steps in more detail. A comprehensive documentation on writing Bro plugins is provided in [3].

2.1 Create the Plugin Skeleton

Creating the plugin skeleton is really easy in Bro, as a Bro script called “init-plugin” exists for that purpose. Assuming that <bro-dist> is the root directory of the Bro source code, the script is in the directory:

```
<bro-dist>/aux/bro-aux/plugin-support/init-plugin
```

This script takes three arguments: destination directory, namespace, and plugin name. The destination directory is where the plugin skeleton will be created. The namespace and plugin name will uniquely identify the plugin, but namespace also has another usage in the context of C++, i.e. it identifies the namespace where the plugin object is defined. A sample command for creating a plugin in a directory called “PTunnel_plugin”, with the namespace “PTunnel” and plugin name “FeatureExtraction” is:

```
./init-plugin <directory>/PTunnel_plugin PTunnel FeatureExtraction
```

This script will copy the plugin skeleton from the Bro source directory to <directory>/PTunnel_plugin. Below are the files and directories that we are interested in:

scripts/ contains the Bro script relevant for the plugin.

src/ contains the source code files for the plugin.

CHANGE is change log file for the plugin.

configure is the script to configure the plugin compilation. It requires access to the Bro source code directory.

CMakeLists.txt can be modified to also include additional C++ code that needs to be compiled.

COPYING.edit-me is the standard template of the BSD license. Only COPYING is included included in the distribution, so we have to edit and rename the file.

Makefile is the makefile to build the plugin.

README contains information about the plugin, how to build, how to use, etc.

VERSION contains the major and minor version of the plugin.

2.2 Write the Core of the Plugin

After we have created the plugin skeleton, the next step is to write the content of the plugin in C++. Whatever code we want to implement, we will need to hook it to the “Plugin.cc” contained in the “src/” subdirectory. That is, the code has to be visible from “Plugin.cc”. The simplest way to do this is implementing the code inside the “src/” subdirectory using C++, since that is where “Plugin.cc” is located. In the subsequent Section 2.2.1 we go into detail on how we can implement a Feature Extraction plugin and Section 2.2.2 is dedicated to explain more about how to implement a Feature Extraction plugin with protocol parser.

When writing the core of the plugin, it is important to interface the script and the plugin properly through the .bif file and to create a proper reference to the values to return so that Bro can clean up objects properly. In particular, since Bro allocates and deallocates objects based on their types, if we create a custom type that Bro does not recognise, then Bro will not be able to deallocate the object properly. This can result in a memory leak. For example, assume that *vl* is a variable used to hold the event’s parameters, if we want to add a vector to *vl*, then the following line is not a proper way to create a new vector:

```
vl->append(new VectorVal(new VectorType(new BroType(TYPE_VECTOR))));
```

Since the type is not a standard Bro type, Bro could not deallocate the vector after the event handler has finished handling the event. Instead, we should use Bro's internal_type function. In our particular example, the correct way of doing it is:

```
vl->append(new VectorVal(internal_type("FeatureAnalysis::feature_vector")->AsVectorType()));
```

2.2.1 Writing Feature Extraction Plugin

The Ping Tunnel plugin is simple, it just extracts several bytes from the ICMP payload based on the parameters "position" and "len". We take the first 4 bytes from the ICMP payload and compare it with Ping Tunnel's magic number. The main thing of interest here is the mechanism to communicate with the Bro script. Bro is event driven, that is, the execution revolves around handling events from the event queue. We make use of this approach and queue the event to be processed by the Bro script layer through calling the function

```
mgr.QueueEvent(<event handler>, <parameter list>);
```

where <event handler> is defined in the script and the <parameter list> is an object that Bro script recognises as parameter(s) of the event. Note that if <event handler> is not defined in the script, then it will be NULL. Furthermore, if <event handler> is NULL then the call to queue the event will not have any effect. This can make debugging quite hard, since a typo in the event handler name will not cause an explicit error, but it will only look like the plugin is not working (did not queue any event). Algorithm 1 is our example proof of concept implementation for the Ping Tunnel Feature Extraction plugin.

Except for adding the reference to the .bif file ("featureextraction.bif.h") and the ExtractFeature() function, everything else that is not marked in bold is automatically generated when the plugin skeleton is created. In this particular example, the code takes the first few bytes of the payload and converts it into a vector of double. This vector is then bundled along with the connection UID, the 5-tuple ID, and the direction as event arguments (in the form of "val_list"), which is then queued into the event queue with "mgr.QueueEvent()".

2.2.2 Writing a Feature Extraction Plugin with Protocol Parser

An ICMP protocol analyser is already provided by Bro, so we do not need to create another ICMP parser. Nevertheless, in other cases it may be necessary to create a particular protocol parser, so here we provide an example of how to create one. In this example, we show how to parse and extract the URG flag and URG pointer from a TCP packet.

When writing a new protocol parser, it is important to keep in mind how Bro analysis specific protocols, as we will need to hook into the Bro analyser tree to parse the new protocol of interest. Whenever Bro detects a new flow, Bro will assign an analyser for it. Bro can dynamically detect whether the protocol type is TCP, UDP, ICMP, or UNKNOWN. For each protocol type, Bro will assign the base protocol analyser as the root analyser. An analyser for a next higher-layer protocol will be added as a child of this root analyser. For UDP and ICMP, the new packet analyser can be inserted using the function AddChildAnalyzer(). However, if we want to analyse protocols on top of TCP, then we have to use the function AddChildPacketAnalyzer() which is specific to the TCP analyser. This is due to the nature of TCP which treats the payload as a stream of bytes rather than as packets.

Algorithm 2 shows the plugin code and Algorithm 3 shows the protocol analyser. The main role of the plugin code is to initialise the plugin and attach the the new protocol analyser to the Bro analyser tree. The new analyser inherits Bro analyser and implements the functions to parse incoming packets, which Bro will call whenever it traverses the analyser tree.

Writing this type of plugin consists of four main steps: (1) adding the parser as a Bro component[1], (2) hooking the component into Bro's analyser tree, (3) attaching the parser to the tree, and (4) implementing the analyser. Adding the parser as a Bro component can be done by calling the function

```
AddComponent(new ::analyzer::Component(string <name>, factory_callback <factory>));
```

[1]Bro defines a component as a specific piece of functionality that a plugin provides.

Algorithm 1 Example of feature extraction plugin for Ping Tunnel

```
#include "Plugin.h"
#include "Event.h" // mgr.QueueEvent()
#include "featureextraction.bif.h"
namespace plugin { namespace PTunnel_FeatureExtraction { Plugin plugin; } }
using namespace plugin::PTunnel_FeatureExtraction;
plugin::Configuration Plugin::Configure()
{
  plugin::Configuration config;
  config.name = "PTunnel::FeatureExtraction";
  config.description = "<Insert description>";
  config.version.major = 0;
  config.version.minor = 1;
  return config;
}
void plugin::PTunnel_FeatureExtraction::ExtractFeature(StringVal* UID, Val* conn_ID,
  Val* direction, StringVal* payload, unsigned int position, unsigned int len)
{
  if (payload->Len() < position + len) return; // if there's not enough bytes then skip

  const u_char* bytes = payload->Bytes();
  val_list* vl = new val_list;
  unsigned int i;

  // pass around Bro unique UID
  BroString* newStr = new BroString(*(UID->AsString()));
  vl->append(new StringVal(newStr)); // Bro unique UID

  // pass around the connection ID (tuple)
  RecordVal* conn_r = conn_ID->AsRecordVal();
  RecordVal* conn_ID_copy = new RecordVal(conn_id);
  conn_ID_copy->Assign(0, new AddrVal(conn_r->Lookup(0)->AsAddr()));
  PortVal* src_port = conn_r->Lookup(1)->AsPortVal();
  conn_ID_copy->Assign(1, new PortVal(src_port->Port(), src_port->PortType()));
  conn_ID_copy->Assign(2, new AddrVal(conn_r->Lookup(2)->AsAddr()));
  PortVal* dst_port = conn_r->Lookup(3)->AsPortVal();
  conn_ID_copy->Assign(3, new PortVal(dst_port->Port(), dst_port->PortType()));
  vl->append(conn_ID_copy); // 4 tuple network ID
  vl->append(new Val(direction->AsEnum(), TYPE_ENUM));// direction
  vl->append(new VectorVal(internal_type("FeatureAnalysis::feature_vector")->AsVectorType()));
  for (i = 0; i < len; i++) (*vl)[3]->AsVectorVal()->Assign(i, new Val((double)(*(bytes+position+i)), TYPE_DOUBLE));
  mgr.QueueEvent(FeatureExtraction::PTunnel_feature_event, vl);
}
```

Algorithm 2 Plugin for URG flag and URG parser

```
#include "Plugin.h"
#include "Parser.h"
namespace plugin { namespace FeatureExtraction_UrgentPointer { Plugin plugin; } }
using namespace plugin::FeatureExtraction_UrgentPointer;
plugin::Configuration Plugin::Configure()
{
   AddComponent(new ::analyzer::Component("URG_parser",
      ::plugin::FeatureExtraction_UrgentPointer::URG_parser::Instantiate));
   plugin::Configuration config;
   config.name = "Header::Parser";
   config.description = "<Insert description>";
   config.version.major = 0;
   config.version.minor = 2;
   EnableHook(HOOK_SETUP_ANALYZER_TREE, 0);
   return config;
}
void Plugin::HookSetupAnalyzerTree(Connection *conn) {
   if ( conn->ConnTransport() != TRANSPORT_TCP ) return;
   analyzer::TransportLayerAnalyzer* root = conn->GetRootAnalyzer();
   URG_parser* urg_parser = new URG_parser(conn);
   ((analyzer::tcp::TCP_Analyzer *) root)->AddChildPacketAnalyzer(urg_parser);
   urg_parser->Init();
}
```

where <name> is the name of the component, and <factory> is the handle to instantiate the parser. Bro defines this factory as “A factory function to instantiate instances of the analyzer’s class, which must be derived directly or indirectly from analyzer::Analyzer. This is typically a static *Instatiate()* method inside the class that just allocates and returns a new instance.”. In our particular example, the factory is “::plugin::FeatureExtraction_UrgentPointer::URG_parser::Instantiate”. In order to hook into the Bro analyser tree, we first need to add the hook to the function in Bro which attaches the root analyser to the flow. This can be done by

EnableHook(HOOK_SETUP_ANALYZER_TREE, <priority>);

in the Configure() function of the plugin and by overriding the function

void Plugin::HookSetupAnalyzerTree(Connection *conn)

Here <priority> is an integer denoting the priority of the plugin in the case there are other plugins that implement the hook as well. The smaller the number, the higher the priority. Except for adding the EnableHook() call, the HookSetupAnalyzerTree() function and the code to attach the parser to the Bro analyser tree (denoted in bold in Algorithm 2), everything else is automatically generated when the plugin skeleton is created.

Implementing the analyser can be done by inheriting the base analyzer class, and then overriding the constructor, destructor, the Init() and Done() methods, the DeliverPacket() method, and the IsReuse() method. The main function that we need to implement is the DeliverPacket() method which will be called for each incoming packet that Bro sees. Note that in this example we are only interested in the implementation of DeliverPacket(), since this is the function Bro calls to parse an incoming packet (denoted in bold in Algorithm 3). The code takes the TCP header (which is included in the IP payload) and then extracts the source and destination ports, which will be used to create the 5-tuple. The parser then extracts the urgent flag and urgent pointer from the TCP header, and bundles the values along with the connection UID, the 5-tuple and the direction as event arguments. The event is then added to the event queue.

Algorithm 3 Protocol parser for URG flag and URG ptr

```
#include "analyzer/protocol/tcp/TCP.h"
#include "parser.bif.h"
#include "Event.h"
namespace plugin { namespace Header_Parser {
class Parser : public analyzer::TransportLayerAnalyzer {
public:
   Parser(Connection* conn);
   virtual ~Parser();
   virtual void Init();
   virtual void Done();
   static analyzer::Analyzer* Instantiate(Connection* conn)
   {
      return new Parser(conn);
   }
protected:
   virtual void DeliverPacket(int len, const u_char* data, bool is_orig,
      uint64 seq, const IP_Hdr* ip, int caplen);
   virtual bool IsReuse(double t, const u_char* pkt);
};
} }
using namespace plugin::Header_Parser;
Parser::Parser(Connection* conn) : TransportLayerAnalyzer ("Parser", conn) { }
Parser::~Parser() { }
void Parser::Init() { Analyzer::Init(); }
void Parser::Done() { Analyzer::Done(); }
bool Parser::IsReuse(double t, const u_char* pkt) { return 0; }
void Parser::DeliverPacket(int len, const u_char* data, bool is_orig,
      uint64 seq, const IP_Hdr* ip, int caplen) {
   if ((data == NULL) || (ip == NULL)) return; udp port
   Analyzer::DeliverPacket(len, data, is_orig, seq, ip, caplen);
   const struct tcphdr* tp = (const struct tcphdr*) ip->Payload();
   RecordVal* id_val = new RecordVal(conn_id); // build the conn_id
   id_val->Assign(0, new AddrVal(_conn->OrigAddr()));
   id_val->Assign(1, new PortVal(ntohs(_conn->OrigPort()), TRANSPORT_TCP));
   id_val->Assign(2, new AddrVal(_conn->RespAddr()));
   id_val->Assign(3, new PortVal(ntohs(_conn->RespPort()), TRANSPORT_TCP));
   val_list *vl = new val_list;
   vl->append(new StringVal((_conn->GetUID()).Base62("C"))); // pass the UID
   vl->append(id_val); // conn_ID
   vl->append(new Val((is_orig) ? 0 : 1, TYPE_ENUM)); // Direction
   vl->append(new Val(tp->urg, TYPE_COUNT)); // URG_flag
   vl->append(new Val(ntohs(tp->urg_ptr), TYPE_COUNT)); // URG_ptr
   mgr.QueueEvent(FeatureExtraction::URG_feature_event, vl);
}
```

Algorithm 4 Example .bif file

```
module PTunnel;

%%{
   #include "Plugin.h"
%%}

ExtractHeaderFeature%(UID : string, id : conn_id, payload: string, position:count, len:count%): bool
%{
   plugin::PTunnel_FeatureExtraction::ExtractFeature(UID, id, payload, position, len);
   return new Val(1, TYPE_BOOL);
%}

event feature_event%(value:int%);
```

2.3 Add Built-in Functions and Events Declaration (BiF file)

Here, we provide a quick introduction into writing a .bif file. A .bif file is an important part of writing Bro plugins, because it provides a simplified syntax (C++-like syntax) to write code, events, and function definitions that Bro scripts may use. It also enables the C++ code to see the types defined in the Bro script. Effectively a .bif file acts as an interface between Bro script and C++ code.

A .bif file has its own syntax, which for the most part is a mix of Bro script and C++. To be more precise, anything that is between "%{" and "%}" is treated as pure C++. As far as we understand, the parameters of a function or an event have the same syntax as parameters in Bro script (the .bif file can see the type defined in Bro script), and they are enclosed in "%(" and "%)". As part of the Bro documentation there is a tutorial on on how to write a .bif file, including the detailed data type documentation [1].

Algorithm 4 is an example of our .bif file. There are some things that we want to highlight:

- The "module" keyword defines the name of the module which contains the functions and event definitions. We need to know the module name in order to call the functions or handle the events from the script layer. For example, in this particular scenario the script has to refer to the event as "PTunnel::feature_event" and the function as "PTunnel::ExtractHeaderFeature".
- We need to include a reference to the plugin core that we implemented as otherwise there is no access to the plugin's contents. In particular, we need to include "Plugin.h" where "Plugin.h" is the header file of the plugin.
- In C++ "::" is used to access the content of the namespace. Recall that our plugin is defined under the namespace of "PTunnel_FeatureExtraction", and this is further located under the namespace of "plugin". So to access the ExtractFeature() function we need to use the full name.

2.4 Write the Script

The Bro script (usually dubbed "Scriptland" by the Bro community) is where the events are handled. In order to make use of any events, whether provided by Bro or a plugin, we have to create an event handler in the script. Note that there is a tutorial in the Bro documentation on how to write a script [4]. The main interaction between the script and the plugin is through the .bif file, so we only need to know the interface of the plugin defined in the .bif file. Algorithm 5 is an example of a script to handle ICMP echo requests and replies which are required by the Ping Tunnel detection plugin.

Note that we have to be careful when we write the names of events that we want to handle, for example, in this case "icmp_echo_request" or "icmp_echo_reply". Bro will not complain if the event name is mistyped, but it will render the event handler useless. This is particularly important since then the event handler pointer in the plugin will be NULL, which can cause undesirable effects (see Section 2.2.1 for an example of such an effect).

By default, Bro only loads two scripts from the plugin:

<plugin_root_directory>/scripts/__load__.bro

Algorithm 5 Example Bro Script

```
event icmp_echo_reply(c: connection, icmp: icmp_conn, id: count, seq: count, payload: string)
{
  PTunnel::ExtractHeaderFeature(payload);
}
event icmp_echo_request(c: connection, icmp: icmp_conn, id: count, seq: count, payload: string)
{
  PTunnel::ExtractHeaderFeature(payload);
}
```

```
<plugin_root_directory>/scripts/__preload__.bro
```

where <plugin-root-directory> is the plugin folder. Therefore, any additional script has to be called or referenced from “<plugin_root_directory>/scripts/__load__.bro”. The suggestion is to save our custom script as

```
<plugin_root_directory>/scripts/<plugin-namespace>/<plugin-name>/__load__.bro
```

and then add a line in “<plugin_root_directory>/scripts/__load__.bro” to load the script, e.g.

```
@load <plugin-namespace>/<plugin-name>/__load__.bro
```

2.5 Build the Plugin and Bro Setup

In order to compile the plugin, just “configure” and “make” the plugin from the plugin root directory. The prerequisites to build the plugin is to have the Bro source code and all Bro prerequisites installed. In the plugin root directory, type

```
./configure –bro-dist=<bro-dist>
```

to configure the plugin. After configuring the plugin, we can build the plugin by typing

```
make
```

The result of this is a directory called “build” which contains all of the required files for Bro to use the plugin. The next step is then to make Bro actually recognise the plugin. There are two ways to do this, either we point Bro to the directory containing the build files, or we can copy the build files to a directory where Bro will search for available plugins. With <bro> being the location of the installed Bro, the default location where Bro searches for plugins is

```
<bro>/lib/bro/plugins
```

So to install the plugin, from the plugin root directory we can type

```
cp -r build/. <bro>/lib/bro/plugins/<plugin-directory-name>
```

where <plugin-directory-name> is an arbitrary directory name used to distinguish the plugin. Alternatively we can set the environment variable BRO_PLUGIN_PATH to the new plugin’s build directory as follows

```
export BRO_PLUGIN_PATH=<plugin-root-directory>/build
```

3 Extending the Analysis Plugin

Extending the analysis plugin can be done by extending the classes FeatureAnalyzer and Data_Container, which are defined in Analysis.h and Data_Container.h respectively. Then there are several other places that need attention:

analysis.bif: We need to add an entry in an enumeration to the analysis metric so that it is visible from the script level.

Analysis.h: In the case where we implement a new data container, we need to add an entry in an enumeration here.

Flow.h: Since the enumeration of the Analysis ID is located in analysis.bif, we also need to add a reference to the new analysis metric to the flow configuration file.

Plugin.cc: Here we need to match the analysis metric enumeration and the reference contained in the flow configuration, and also we need to map the analysis metric to the data container used.

Flow.cc: This is the core of adding the feature value to the data container and metric calculation when it is triggered. We need to add the case handling for the new analysis metric (and corresponding data container).

For example, suppose we have an analysis where we store the summation of the feature values and calculate the average value as the metric value. Below are the data container (Figure 6) and the analyzer class of such analysis (Figure 7).

Algorithm 6 Example of Data Container

```
#include "Data_Container.h"

namespace CCD {

class Sum_Avg_Data : public Data_Container {
public: Sum_Avg_Data() _sum(0), _count(0) {}
    virtual ~Regularity_Data() {}
    virtual void add_feature(double feature) {_sum+=feature; _count++;}
    double get_sum() {return _sum;}
    unsigned long get_count() {return _count;}

private:
    double _sum;
    unsigned long _count;
};

}
```

Algorithm 7 Example of Analyzer

```
#include "Analysis.h" // superclass
#include "Regularity_Data_Container.h" // Regularity_Data

namespace CCD {

class Sum_Avg : public FeatureAnalyzer {
public: Sum_Avg(std::shared_ptr<Sum_Avg_Data> data) : FeatureAnalyzer(data) {}
    virtual ~Sum_Avg() {};
    virtual double calculate_metric() {return _data->get_sum() / (double) _data->get_count();};
};

}
```

Since we add a data container, we need to update an enumeration in the Data_container.h (Figure 8)

Algorithm 8 Data_Container.h

```
...
const int DATA_CONTAINER_TYPE_COUNT = 6;
enum Data_Container_Enum {
   RAW_DATA = 0,
   HISTOGRAM_DATA = 1,
   PATTERN_DATA = 2,
   REGULARITY_DATA = 3,
   NULL_DATA = 4,
   SUM_AVG_DATA = 5
};
...
```

We also need to add an entry to the enumeration in the analysis.bif file (see Figure 9) and the file Flow.h (see Figure 10).

Algorithm 9 analysis.bif

```
...
enum Analysis_ID %{
   KS_ANALYSIS = 0,
   ENTROPY_ANALYSIS = 1,
   CCE_ANALYSIS = 2,
   NULL_ANALYSIS = 3,
   MULTIMODAL_ANALYSIS = 4,
   AUTOCORRELATION_ANALYSIS = 5,
   REGULARITY_ANALYSIS = 6,
   SUM_AVG_ANALYSIS = 7,
%}
...
```

Algorithm 10 Flow.h

```
...
class FlowConfig { public:
...
   unsigned int KS_analysis; // analysis ID for KS test
   unsigned int Entropy_analysis; // analysis ID for Entropy
   unsigned int CCE_analysis; // analysis ID for CCE
   unsigned int Regularity_analysis; // analysis ID for Regularity
   unsigned int Autocorrelation_analysis; // analysis ID for Autocorrelation
   unsigned int MultiModal_analysis; // analysis ID for Multi-Modality
   unsigned int Null_analysis; // analysis ID for null analysis
   unsigned int Sum_Avg_analysis;
...
};
...
```

Next, we create a mapping between the definitions in analysis.bif and FlowConfig (see Figure 11). We also add a mapping between the analysis and its data container (SUM_AVG_ANALYSIS ~ SUM_AVG_DATA).

Algorithm 11 Plugin.cc

```
...
plugin::Configuration Plugin::Configure() {
...
   // get the enumeration from analysis.bif
   _flow_config->KS_analysis = KS_ANALYSIS;
   _flow_config->Entropy_analysis = ENTROPY_ANALYSIS;
   _flow_config->CCE_analysis = CCE_ANALYSIS;
   _flow_config->Autocorrelation_analysis = AUTOCORRELATION_ANALYSIS;
   _flow_config->MultiModal_analysis = MULTIMODAL_ANALYSIS;
   _flow_config->Regularity_analysis = REGULARITY_ANALYSIS;
   _flow_config->Null_analysis = NULL_ANALYSIS;
   _flow_config->Sum_Avg_analysis = SUM_AVG_ANALYSIS;

   // mapping from analysis to its data container
   _flow_config->map_analysis_data.resize(NUMBER_OF_ANALYSIS);
   _flow_config->map_analysis_data[KS_ANALYSIS] = RAW_DATA;
   _flow_config->map_analysis_data[ENTROPY_ANALYSIS] = HISTOGRAM_DATA;
   _flow_config->map_analysis_data[CCE_ANALYSIS] = PATTERN_DATA;
   _flow_config->map_analysis_data[NULL_ANALYSIS] = NULL_DATA;
   _flow_config->map_analysis_data[MULTIMODAL_ANALYSIS] = HISTOGRAM_DATA;
   _flow_config->map_analysis_data[AUTOCORRELATION_ANALYSIS] = RAW_DATA;
   _flow_config->map_analysis_data[REGULARITY_ANALYSIS] = REGULARITY_DATA;
   _flow_config->map_analysis_data[SUM_AVG_ANALYSIS] = SUM_AVG_DATA;
...
}
...
```

Finally, we add the function to add the feature into the data container, and the call to the analysis object when the analysis is triggered (see Figure 12)

Algorithm 12 Flow.cc

```
...
void Flow::add_feature(unsigned int tag, std::vector<unsigned int> aid, double feature) {
   ...
   switch (type) {
      ...
      case REGULARITY_DATA : {
         _data[_current_set_ID][tag][type] = std::shared_ptr<Sum_Avg_Data> (new Sum_Avg_Data ()); break; }
      ...
      }
   ...
}
void Flow::add_analysis(unsigned int tag, unsigned int aid) {
   ...
   if (_analysis[_current_set_ID][tag][aid].get() == NULL) {
      ...
      else if (aid == _config->Sum_Avg_analysis)
         _analysis[_current_set_ID][tag][aid] = std::shared_ptr<Sum_Avg>
            (new Sum_Avg(std::static_pointer_cast<Sum_Avg_Data> (_data [_current_set_ID][tag][type])));
      ...
   }
   ...
}
...
```

4 Conclusions

In this report we explained how to write plugins for BroCCaDe using two examples: a feature extraction plugin based on a working example of creating a plugin to detect Ping Tunnel, and a feature extraction plugin with protocol parser to extract the TCP URG flag and URG pointer. We also discussed how to extend BroCCaDe's analysis plugin with a new analysis metric.

Acknowledgements

This work was supported by a grant from the Comcast Innovation Fund.

References

[1] Extending bro with builtin functions (bif). `https://www.bro.org/development/howtos/bif-doc/`, Accessed: 11 April 2017.

[2] Ping tunnel - for those times when everything else is blocked. `http://www.cs.uit.no/~daniels/PingTunnel/`, Accessed: 6 April 2017.

[3] Writing bro plugins. `https://www.bro.org/sphinx-git/devel/plugins.html`, Accessed: 27 March 2017.

[4] Writing bro scripts. `https://www.bro.org/sphinx/scripting/index.html`, Accessed: 27 March 2017.

[5] H. Gunadi and S. Zander. Bro Covert Channel Detection (BroCCaDe) Framework: Design and Implementation. Technical Report 20171117B, School of Engineering and IT, Murdoch University, 2017.

[6] S. Wendzel, S. Zander, B. Fechner, and C. Herdin. Pattern-based survey and categorization of network covert channel techniques. *ACM Computing Surveys (CSUR)*, 47(3):50, 2015.