**Introduction**

GOOSE is a communication protocol defined in the IEC61850 standard. It is used by Intelligent Electronic Devices (IEDs) in electrical substations to facilitate information exchange between devices. A GOOSE parser has been developed to enable detailed analysis of the transmitted data and allow rule-based identification of anomalies related to cybersecurity attacks. It is compatible with an older instance of Zeek Network Security Monitor (v2.6).

In this guide, the installation and usage of the parser will be described.

**System Requirements**

In general, the GOOSE parser can run on any system that supports Zeek. In our setup, it was tested successfully in a virtual machine environment with the following configuration.

|  |  |
| --- | --- |
| **Component** | **Setting** |
| Operating System | Ubuntu 18.04 |
| RAM | 4 GB |
| Processor | 3.5 GHz |
| Disk Space | 20 GB |

**List of New/Modified Files and Folders**

|  |  |  |
| --- | --- | --- |
| **S/N** | **Description** | **Path** |
| 1 | Main parser files | src/analyzer/protocol/goose/ |
| 2 | Definition of GOOSE record (goose\_records::gcp) | scripts/base/init-bare.bro |
|  | Sample Script A – Print & log packets | scripts/base/protocols/goose/Sample\_Script\_A.bro |
| 4 | Sample Script B - Check if stNum value rolls over correctly | scripts/base/protocols/goose/Sample\_Script\_B.bro |
| 5 | Sample Script C - Check if stNum/sqNum are set accordingly when dataset is updated | scripts/base/protocols/goose/Sample\_Script\_C.bro |
| 6 | Btest files (for baseline test) | testing/btest/Baseline/scripts.base.protocols.goose.goose\_test\_script/  testing/btest/scripts/base/protocols/goose/  testing/scripts/goose-test.bro |
| 7 | Modified Zeek Core Files to support Layer 2 protocol parsing | src/Sessions.cc  src/Sessions.h  src/analyzer/protocol/CMakeLists.txt  src/iosource/Packet.cc  src/iosource/Packet.h  src/types.bif |

**Installation**

The GOOSE parser is built upon the framework provided by the Zeek Network Security Monitor. Formerly known as Bro, Zeek is an open source IDS which allows comprehensive network analysis. The GOOSE parser, downloadable as a patch, has to be applied to a compatible version of Zeek. The installation steps are shown below.

**Zeek Installation**

1. Install the required dependencies for Zeek listed in the official website:

<https://docs.zeek.org/en/current/install/install.html#required-dependencies>

1. Clone the Zeek repository with the following command:

*# git clone --recursive* [*https://github.com/zeek/zeek*](https://github.com/zeek/zeek)

**Patch Installation**

1. Switch to a snapshot of the Zeek repository that is compatible with the GOOSE parser.

*# cd <zeek\_dir>/*

*# git checkout aff3f4*

*# git submodule update --init --recursive*

1. Apply the GOOSE parser as a patch. The whitespace warnings may be ignored.

*# git apply --reject --whitespace=fix goose\_parser.patch*

1. Build and install from source. Some commands may require root privileges.

*# ./configure*

*# make*

*# make install*

**Usage**

The ADSC Github repository contains sample GOOSE trace files. These traces were generated as part of a research project in GOOSE communication within a typical substation and are available for download from this link: <https://github.com/smartgridadsc/IEC61850SecurityDataset>

A trace file can be analysed in Zeek from the terminal with the following commands:

*# cd <project\_directory>*

*# sudo ./build/src/bro –r <trace\_file> ./scripts/base/protocols/goose/Sample\_Script\_A.bro*

A subset of the GOOSE data fields relevant for cybersecurity analysis will be printed to the console and logged into ‘goose.log’ upon execution.

**Rule-Based Anomaly Detection**

The underlying framework of Zeek consists of 2 major abstraction layers; Event Engine and Script Interpreter (<https://docs.zeek.org/en/current/intro/>). The event engine performs the core packet processing and raises a series of *events.* The handling of these *events* can be prescribed in the script layer in the form of Zeek scripts that may implement actions to be taken based on received data from the event engine.

The GOOSE parser residing in the event engine layer extracts the fields in a GOOSE packet according to Section ‘GOOSE Data Fields and Supported Datatypes’. On successful parsing of a packet, an event is raised with the following signature, accessible from user-defined scripts:

*event goose\_packet\_event(p: goose\_records::gcp)*

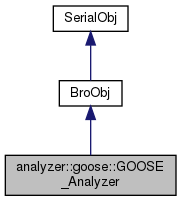
The input parameter *p*, of type, goose\_records::gcp is a Zeek record (similar to a struct in C) and defines the following fields for each GOOSE packet:

|  |  |  |
| --- | --- | --- |
| **Description** | **Field Name** | **Zeek Datatype** |
| Packet timestamp | packet\_ts | double |
| Source MAC Address | src\_mac | string |
| Destination MAC Address | dest\_mac | string |
| Status Number | stNum | count (equivalent to unsigned int) |
| Sequence Number | sqNum | count (equivalent to unsigned int) |
| gocbRef | packet\_type | string |
| \*Numeric data entries | data\_values | vector of double |
| String data entries | string\_values | vector of string |

\*All data entries of type int, bool and float will be converted to double when passed from the event engine to the scripting layer.

**GOOSE Parser Design**

The parser has one main analyser class *GOOSE\_Analyzer* defined in the *analyzer* namespace of Zeek in C++ files. This class inherits from *BroObj* directly to provide support for lower level protocols as GOOSE is transported on layer 2.

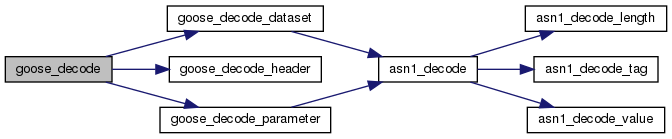


*GOOSE\_Analyzer class* interacts with the lower layers of Zeek to receive and consume live packets from the network or packets from input trace files. The I/O source is determined from the command line flags when starting the program and monitored for incoming data. Sanity checks are done by Zeek and packets are dispatched to the respective analysers. When a new packet is passed to the GOOSE analyser, it allocates a local buffer and begins the decoding process.



The decoding and storage of packet data is performed in a set of supporting C files and structures.

The bytes of the packet are interpreted according to the datatype and size with appropriate byte-order conversions. The data is then stored in separate C structures representing contents of the header, Protocol Data Unit (PDU) and dataset.



On successful completion of the decode process, the data selected to be passed to the scripting engine is converted from C objects to Zeek objects. Each numeric dataset entry is transformed from its original type to double and inserted into a vector. Each string based dataset entry is inserted into a second vector. An event is then raised with the processed data as input parameters and added to the event queue for script level handling. Failure to decode will be logged in *weird.log* and no events will be generated.

**Script Files**

Zeek offers a simplified scripting language with common programming constructs and datatypes. For a full tutorial, please refer to the official Zeek documentation (<https://docs.zeek.org/en/current/examples/scripting/>).

Users may define rules in new script files following the scripting syntax. A script is activated by appending its filename to the command when running the program (see Section ‘Testing’).

The following sample scripts included in the installation serve to show simple implementations of rule-based handling of events.

**Sample Script B: Check if stNum value rolls over correctly**

# Define a module name

module Sample\_B;

# Constant variable declarations

const stNum\_MIN = 1;

const stNum\_MAX = 4294967295;

# Container to store IED identifier (e.g. gocbRef) where stNum has hit MAX value

global ieds: set[string];

# Entry point. This event will be generated by the event engine when a new packet arrives.

event goose\_packet\_event(p:goose\_records::gcp)

{

# stNum value fails to rollover correctly

if(p$packet\_type in ieds && p$stNum!=stNum\_MIN)

print "Error: stNum failed to rollover", p;

# stNum value correctly rolls over from MAX to MIN

else if(p$packet\_type in ieds && p$stNum==stNum\_MIN)

delete ieds[p$packet\_type];

# stNum reached the MAX

else if(p$stNum==stNum\_MAX)

add ieds[p$packet\_type];

}

**Sample Script C: Check if stNum/sqNum are set accordingly when dataset is updated**

# Define a module name

module Sample\_C;

# Forward Declaration

global dataset\_has\_changed: function(packet\_type: string, current\_data\_values: vector of double): bool;

# Define a record to store previous packet data

type previous\_record: record {

stNum: count;

sqNum: count;

data\_values: double\_vec;

};

# Define a container of previous packet data, indexed by gocbRef of type string

global prev: table [string] of previous\_record;

# Entry point. This event will be generated by the event engine when a new packet arrives.

event goose\_packet\_event(p: goose\_records::gcp)

{

if(dataset\_has\_changed(p$packet\_type, p$data\_values))

{

# When dataset changes, check that stNum increments by 1 and sqNum is reset

if(p$stNum != prev[p$packet\_type]$stNum+1 || p$sqNum!=0)

{

print "Error in stNum/sqNum sequence", p;

print "Previous data", prev[p$packet\_type];

}

}

# Store the packet data

prev[p$packet\_type] = [$stNum = p$stNum, $sqNum = p$sqNum, $data\_values = p$data\_values];

}

function dataset\_has\_changed(packet\_type: string, current\_data\_values: double\_vec): bool

{

if([packet\_type] in prev==F)

return F;

for (i in current\_data\_values)

if (prev[packet\_type]$data\_values[i]!=current\_data\_values[i])

return T;

return F;

}

**GOOSE Data Fields and Supported Datatypes**

The following tables list the GOOSE fields that are processed and the datatypes currently supported.

|  |  |
| --- | --- |
| **Field** | **Length (Bytes)** |
| Destination MAC Address | 6 |
| Source MAC Address | 6 |
| VLAN Tag TPID [Optional] | 2 |
| VLAN Tag TCI [Optional] | 2 |
| EtherType = 0x88b8 | 2 |
| APPID | 2 |
| Length | 2 |
| Reserved 1 | 2 |
| Reserved 2 | 2 |
| GOOSE APDU | - |
| Frame check sequence | 0 (not parsed) |

|  |  |
| --- | --- |
| **GOOSE APDU Field** | **Length (Bytes)** |
| gocbRef | Defined in \*TLV |
| timeAllowedToLive | Defined in TLV |
| datSet | Defined in TLV |
| goID | Defined in TLV |
| t | Defined in TLV |
| stNum | Defined in TLV |
| sqNum | Defined in TLV |
| test | Defined in TLV |
| confRev | Defined in TLV |
| ndsCom | Defined in TLV |
| numDatSetEntries | Defined in TLV |
| allData | Defined in TLV per entry |

\* TLV (Tag-Length-Value) is a generic network encoding scheme utilized in IEC61850 to describe the value type, length and data.

The ‘allData’ field in the GOOSE APDU contains a sequence of entries defined by the user. Each entry has its own datatype. The GOOSE parser currently supports the following datatypes in ASN.1 BER Encoding:

|  |  |
| --- | --- |
| **Datatype** | **Limits** |
| Boolean |  |
| Integer | 32-bit |
| Unsigned Integer | 32-bit |
| Floating point | 32-bit |
| BitString | Primitive encoding |
| Octet String |  |
| Visible String | Single-byte character set |

**Contact Information**

If you would like to get in touch regarding our project, please contact any of the contributors via email:

1. Chen Binbin

Email: [binbin.chen@adsc-create.edu.sg](mailto:binbin.chen@adsc-create.edu.sg)

1. Tan Heng Chuan

Email: [hc.tan@adsc-create.edu.sg](mailto:hc.tan@adsc-create.edu.sg)

1. Vyshnavi

Email: [vyshnavi.m@adsc-create.edu.sg](mailto:vyshnavi.m@adsc-create.edu.sg)

For more information on our organization’s research activities, please refer to our main website at <https://adsc.illinois.edu>.