SecSIP Technical Documentation

Alexandre Boeglin, Abdelkader Lahmadi, Olivier Festor

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

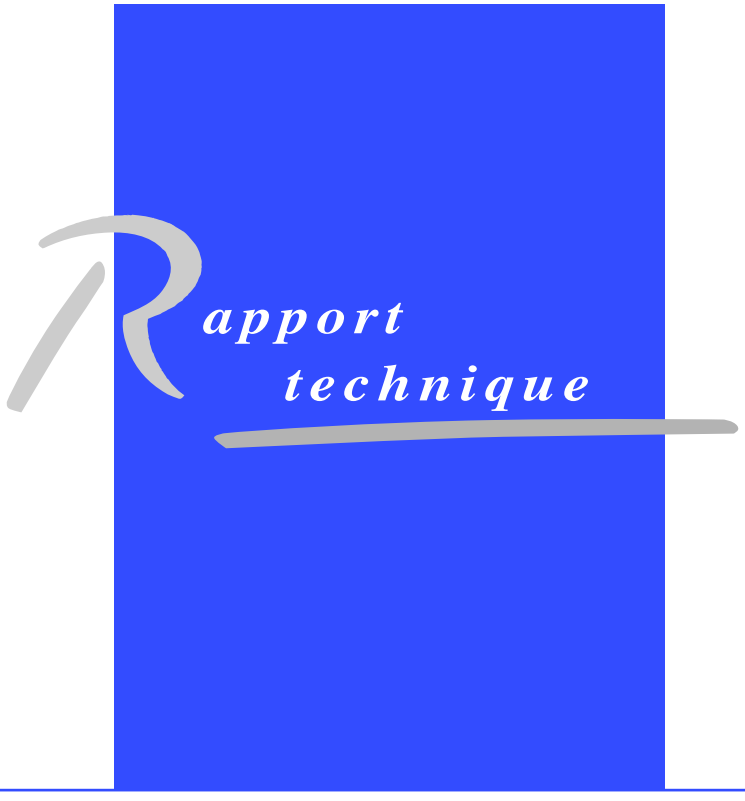
SecSip Technical Documentation

Alexandre Boeglin — Abdelkader Lahmadi — Olivier Festor

N° 0393

July 2010

Networks and Telecommunications



ISSN 0249-0803 ISRN INRIA/RT--0393--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

SecSip Technical Documentation

Alexandre Boeglin , Abdelkader Lahmadi , Olivier Festor

Theme : Networks and Telecommunications
Équipe-Projet Madynes

Rapport technique n° 0393 — July 2010 — 24 pages

Abstract: This document provides information about the architecture of SecSip, a stateful SIP firewall. It first details how the rules are handled by the engine, how they are organized internally, and how they can be imported and exported. It then describes how packets are filtered by the engine, which uses the rules as a basis of its decision-making process. Finally, it goes over the architecture of the manager interface, describing its internals.

Key-words: VoIP, SIP, firewall, SecSip, internals, architecture

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Documentation technique de SecSip

Résumé : Ce document fournit des informations sur l'architecture de SecSip, un pare-feu SIP à conservation d'état. Il détaille dans un premier temps la gestion des règles par le moteur, leur organisation interne, et la manière dont elle peuvent être importées et exportées. Il décrit ensuite le filtrage des paquets par le moteur, qui se base sur les règles pour prendre ses décisions. Enfin, il expose l'architecture et le fonctionnement de l'interface de gestion.

Mots-clés : VoIP, SIP, pare-feu, SecSip, fonctionnement, architecture

Contents

Chapter 1

SecSip Architecture

1.1 Functional Model

The functional architecture of SecSip [2] is presented in figure 1.1.

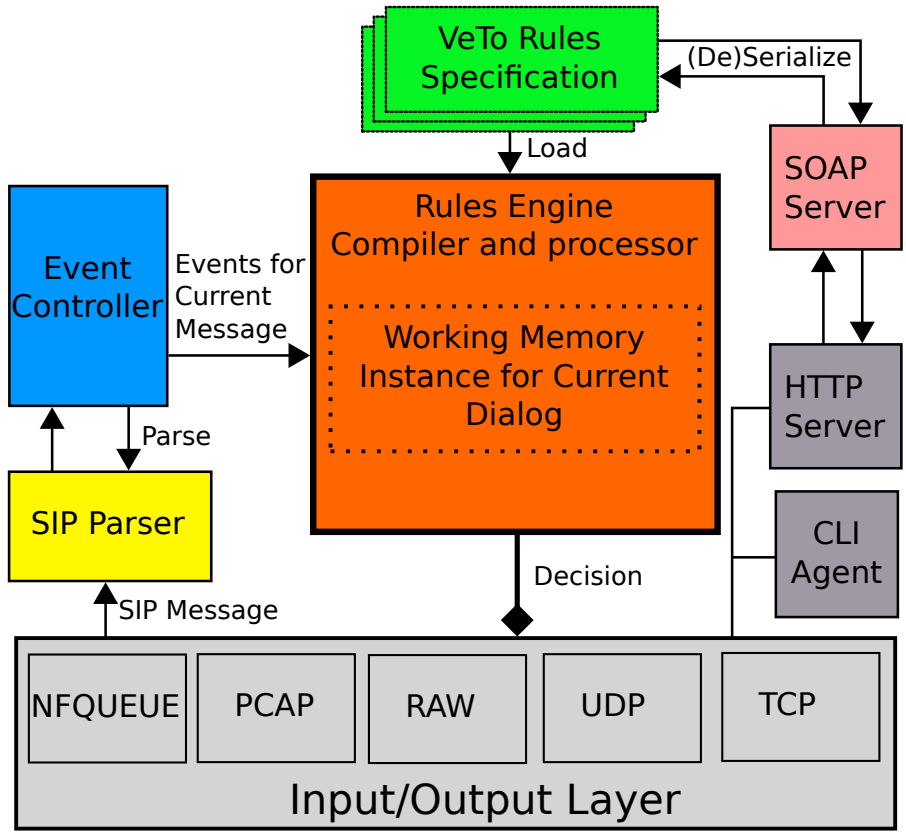


Figure 1.1: Functional Architecture of SecSip

The purpose of each block is detailled below:

- the role of the **Input/Output Layer** is to capture packets from the network, and to pass them to the SIP Parser. And once the processing of the packet by the SecSip Rules Engine is finished, the Input/Output Layer is instructed to either release the packet into the network, or to discard it,
- the **SIP Parser** analyses the packet, and extracts the value of interresting SIP fields from it, and then hands over the extracted data to the EventController, in the form of a SIP Tree,
- the **Event Controller** uses event definitions from the loaded Rules Specification, and generates events in accordance with the presence, or value of certain elements of the SIP Tree,
- the **VeTo Rules Specifications** are data structures that represent sets of filtering rules, in an uncompiled format, which allows them to be easily exported and imported to SecSip. When a Rules Specification gets loaded by the Rules Engine, it is first compiled, in order to offer optimized execution,
- the **Rules Engine** is the main component of SecSip. Its role is to track the state of the SIP dialogs that go through the firewall, and to block them as soon as an attack pattern is detected. To do so, it uses a graph compiled from a VeTo Rules Specification, and events generated by the SIP Parser. The state of each active dialog is stored internally by the engine,
- the **HTTP Server and CLI Agent** provide a way for administrators to interact with SecSip while it's running, either monitoring its status, or changing the filtering rules. The CLI Agent provides a shell-based interface to telnet clients, whereas the HTTP Server provides a web-based interface to web browser, which can communicate with SecSip through SOAP requests,
- the **SOAP Server** handles requests from SOAP clients, converting rule specifications from and to XML trees.

1.2 Source Code Organization

The SecSip source code lies in the `src/` directory of the SecSip distribution. It is organized as follows:

- the `easyc/` directory contains utility functions used by the rest of the software, like memory management, generic field types, and container structures,
- the `api/sipcap/` directory contains the a support library used by the I/O layer implementations,
- the `io/` directory contains implementations of the I/O Layer, that handle capturing packets from the network, and releasing or dropping them after a decision has been made. Implementations exist for pcap, raw sockets, and nfqueue,
- the `easysip/` directory contains the SIP parser and SIP tree implementations, the parser analyses a packet, then fills a tree with the acquired info,

- the `sipf/core/` directory contains the core of SecSip : the main loop, the user agent and soap server implementations, and the dialog data structure,
- the `sipf/checker/` directory contains the rules engine, all the logic and structures that take part in the decision-making process,
- the `manager/` directory contains the html and javasctipt code used by the manager interface.

1.3 VeTo Rules

VeTo [3] rules are used as the basis of the filtering in SecSip, they are used to describe protections against attacks on SIP devices. VeTo rules are organized in blocks.

1.3.1 Block Types

There are three types of VeTo blocks: context blocks, definition blocks, and VeTo blocks.

- **Context Blocks** contain metadata about an attack: which devices are vulnerable, the lifetime of the vulnerability, and its importance,
- **Definition Blocks** are used to define the variables and collections required to track the exploitation of attacks, as well as event detection rules,
- **VeTo Blocks** contain event patterns, associated to actions that have to be taken when the patterns are matched. Those are actually the "rules" that prevent attacks from being exploited.

The VeTo language grammar is extensively described in the **VeTo Reference Manual**.

1.3.2 Configurations

SecSip provides three configuration storages, in order to emulate a NETCONF-like behaviour.

- The **Startup** configuration is the non-volatile storage that will be active at SecSip startup,
- The **Running** configuration is the one currently active in SecSip,
- The **Candidate** configuration is used to draft new configurations, or make changes to existing ones, without impacting SecSip. The candidate configuration can then be "pushed" over the running or startup one.

1.3.3 Rulespecs

Rulespec is the internal structure used by SecSip to store VeTo rules. It is the central point of storage of rules, and thus has to store every bit of information contained in rules, in a way that it can easily be retrieved.

It contains informations that are used by all kind of rules (the targets, the operator and its parameters, and the list of actions), as well as informations specific to each kind of rule (for instance, the event pattern of a VeTo block).

The rulespec structure is used to build the rule objects which are then used by the SecSip engine to process packets.

Text and XML exports are built by serializing this structure, and when importing configuration from text or XML, they are converted to rulespecs.

SecSip internally maintains three sets of rulespecs, as the configuration storages, and each of these storage is composed of three lists of rulespecs, one for each block type.

1.3.4 Configuration Handling

When SecSip starts up, the config file is parsed into a set of rulespecs, which is then stored as the "Startup" configuration. This configuration is then copied over the "Running" configuration and processed into rules by the SecSip engine.

The other serializations and deserializations will take place when some of the the SOAP binding's actions are invoked:

- when a **getConfig** request is received, the target configuration is transformed into an XML tree, and sent as the response,
- and when a **setConfig** request is received, the XML tree included in the request is deserialized into a configuration (a set of rulespecs), and if this configuration is the "Startup" configuration, it will also be used to compose a VeTo rules file, which will be saved as the SecSip config.

Chapter 2

Packet Filtering

When a packet reaches the SecSip firewall, it first has to be "intercepted" so that it can be processed. The packet is then passed to a SIP parser, which gramatically decomposes it in SIP and SDP fields. These fields can then be matched against regular expressions defined in VeTo rules, and events are then generated.

The generated events are then fed into the event graph as tokens. Token are propagated through the graph until an action is reached, or until the graph state does not allow them to be propagated any further.

Actions that have been reached are executed, and they can cause the packet to be dropped.

Finally, the packets that were not dropped are released, and routed into the network.

2.1 Packet Interception and Release

Packet interception and release is implemented in `src/io` and is handled by either NFQueue, or PCAP and raw sockets.

In the case of NFQueue, the packet is held in a queue by the kernel packet filter, until a decision has been made by SecSip, and it is then either released or dropped.

In the case of PCAP and raw sockets, the packet is captured by SecSip, which actually duplicates it, so the original one has to be dropped in any case. Then, if SecSip decides that the packet has to be forwarded, the copy is reinjected into the network using a raw socket, and if the packet has to be dropped, nothing happens, as the original has already been dropped.

2.2 Context Filtering

Context blocks can then be used to filter packets, so that a dialog directed to a specific IP address will only be tested for attacks that the device that uses this address is vulnerable to.

2.3 Definition Blocks and Event Generation

The first actual step of SIP packet filtering is the evaluation of definition blocks. These blocks allow to declare the variables that are used to track the state of a dialog and the different known attack schemes.

By default, these variables are tied to a specific dialog, but their scope can span over all registered dialogs in SecSip, when declared with the "global" keyword.

Variables can be of several types, and they can be static or dynamic. Static variables take their value from SIP and SDP fields or are explicitely defined in context blocks, while the value of dynamic variables is set by logic (inherent to the type of the variable, or set by actions contained in VeTo blocks).

A Variable has one of the following types:

- **collection**: a collection variable is declared with the name of a SIP field as a parameter. Each time the variable is updated, the value of the field for the current packet is added to the collection. Is is then possible to check if a specific value is contained in the collection,
- **counter**: a counter variable starts with a value of 0, and is increased each time the variable is updated. A counter variable also has a "cooldown" facility: the counter can be periodically decreased by a user-defined amount,
- **timer**: a timer variable has a user defined value, and when updated, will start to count down until it reaches zero,
- **state**: a state variable can be used to implement a user defined state machine, and to take actions based on the current state,
- **event**: an event variable is a flag that can be raised conditionnaly. It is used by SecSip to match temporal patterns defined in VeTo rules.

Events are usually declared with a condition (generally a regular expression on one of the SIP fields of the packet), so that when a packet is captured, the event is generated only if the condition is true.

2.4 The Event Graph

In order to track the temporal event patterns, we decided to use an event graph, inspired by active databases [1]. In this section, we detail the structure of the graph, then explain how it is built, and finally go through the execution of the graph.

The event graph is implemented in the `rules_eventgraph.c` file of the `src/sipf/checker` directory of the sources.

2.4.1 Structure of the Graph

Building one single graph for the complete set of rules allows SecSip to factorize as much as possible of the temporal event patterns, and reduce the number of necessary tests, as all events or subpatterns that are common to several event patterns only have to be tracked once.

This way, instead of having to maintain one state machine per event pattern per SIP dialog, we only maintain one global graph state per dialog, which also reduces memory usage.

```
(ev1, [ev2, 3]) -> drop;
(ev1(~ev2)) -> drop;
(ev2{5}) -> drop;
```

Figure 2.1: Example Set Of VeTo Rules

The event graph is modeled using three main structures: graph nodes, links between nodes, and tokens (which represent the flow of events through graphs).

In the event graph, each node represents an event, or a composition of events. Single events are the roots of the graph, and each node to one of the leaves adds a level of composition. At each leaf node of the graph, we have one of the patterns that the graph represents, and the list of conditions and actions associated with the pattern.

Nodes are connected by links, which also carry a meaning, so two nodes can be connected by multiple links with different meaning, depending on the destination node type.

Each node also carries a list of tokens, which represent events or compositions of events as they travel through the graph. Tokens originate from the roots of the graph each time an event is "raised"; and each time a token reaches one of the leaf nodes, the conditions and actions references in this node are processed.

Figure 2.1 shows an example set of rules, and figure 2.2 is the resulting event graph.

- root nodes at the bottom of the graph represent simple events. "*" is the "any" event, which always generates a token, for each received packet, and " ev2" is a negative event, which generates a token when event "ev2" was not triggered by the current packet,

- action nodes are leaves of the graph. each one contains a reference to one of the Veto rules (in this example the three rules are only composed of the "drop" action),

- links between nodes are of different colors, depending on the type of the link (red is left, blue is right, black is start, and grey is trigger),

- the "^" node represents the embedded event "(ev1(ev2))" which means that "ev1" and " ev2" must happen at the same time.

- the "[5]" node is a repeat node, meaning that the left token (ev2) must be repeated exactly 5 times for it to be propagated. The "any" event is used so that this node is triggered at each incoming packet,

- the "(,3)" node is a timer node, meaning that its left token (ev2) must happen within three seconds after the preceding token (ev1). The "any" event is used as a trigger in the cases we negate the timer, and we don't want the left token top be received within the specified time,

- the "," node is a composite node, meaning that its left token (ev1) must happen right before its right token ((ev2, 3)).

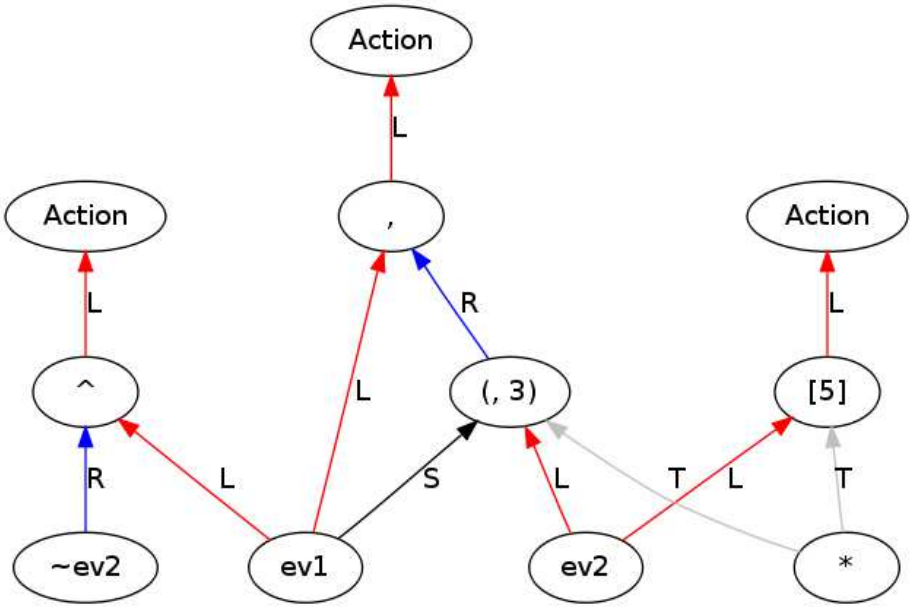


Figure 2.2: Event Graph Produced Using Set Of Rules Defined In Figure 2.1

Tokens

A token carries three pieces of information:

- the **start packet number**, which is the sequence number of the earliest packet for which the token is valid (inclusive),
- the **end packet number**, which is the sequence number of the latest packet for which the token is valid, plus one (exclusive),
- a **reference counter**, which is increased each time a token is pushed through a link, and decreased each time the token is discarded. When it reaches zero after discarding a token, the memory used by the token has to be freed.

The sequence number of a packet is determined by a per-dialog counter. Each time a packet is captured, the value of the counter is associated with the packet, and the counter is incremented.

The idea behind this is that tokens should represent intervals for which events (or sequences of events) are valid. So, when a token hits an action node, its interval will span over the complete attack pattern that was described in the configuration.

For instance, if event ev1 was raised by packet 0, and event ev2 was raised by packet 1, two tokens would have been created: one with interval `[0, 1[` for ev1 and one with interval `[1, 2[` for ev2. If those two tokens were to be pushed to a composite node expecting ev1 to happen right before ev2, it would discard those two tokens, and create a new one, with interval `[0, 2[` and push it further through its links to other nodes of the graph.

Links

Each node may contain a list of links. Links themselves contain a reference to the node they connect to, as well as the type of the link.

Link types are as follows:

- **left** and **right link** are "normal" links, through which tokens usually travel. Their meaning is not necessarily different, for instance, links to an embedded node (in which tokens have to be simultaneous to be propagated), which token came from left or right does not really matter, whereas in composite links, the left token has to happen immediately before the right token,
- the **tick link** is used by nodes that are not only triggered by the tokens they control. In the case of a timer node, the node might have to time out if no token was pushed to it, in which case a left or right link is not an option,
- the **start link** is used to inform the timer node when the timer has to start, and this cannot be done by a left or right link, as the timer has to start before the token it controls is first received.

Nodes

Nodes of the graph represent single events (as the root nodes), or composition of events that form the event patterns that the graph has to match. Each leaf node of the graph represents one of these complete patterns that trigger rule actions.

Each time a token is pushed to a node, it is queued for execution. Nodes are ordered by path length before execution, so that one node is executed only after all its parents have been.

Each type of node has its own execute function, that determines how and when tokens should be propagated.

The following types of nodes are used:

The Root Node A root node produces a token each time the referenced event is detected in a packet(or absent from it, if the negative flag of the node is set to "true"). They are the roots of the graph.

Action Node An action node executes the rules that it references, each time a token is passed to them. They are the leaves of the graph. A rule consists of a condition (optional) and of one or more actions. If there is no condition, or the condition is true, the actions are executed sequentially.

An action node takes its input from a left link.

Composite Node A composite node takes input from one left link and one right link. If a token from the left link immediately precedes a token from the right link, they are merged in a new token (union of the two previous tokens), which is then propagated to children of the current nodes.

Embedded Node An embedded node takes input from one left link and one right link. If a token from the left link overlaps with a token from the right link (meaning that the events happen at the same time, at least partially), they are merged in a new token (intersection of the two previous tokens), which is then propagated to children of the current nodes.

Repeat Node A repeat node has a minimum, and an optional maximum value. It takes its input from a left link, and a tick node is used to trigger the execute function each time a packet is received. When the count of packets from left link is within the min and max range, a new token is propagated, which spans over the matched repeat sequence.

Timer Node A timer node controls wether a token arrives within a certain period of time, defined in seconds. When it receives a token on its start link, it registers the time at which it arrived, and it will forward the first token it receives on its left link, if it is received within the defined period. It can also be set to negative, in which case it will create a new token and propagate it if no token was received on the left link during the defined period of time. In this case, it uses a tick link to be triggered each time a packet is received.

2.4.2 Building the Graph

Rules are added sequentially to the graph. Each time a node insertion is required, SecSip first checks if a similar node already exists in the graph. If such a node exists (same parameters, same parents), it is simply reused.

Once that each node has been added to the graph, their path length is calculated. It is set to zero for action nodes, and increased by one for each traversed node. For a node that has multiple paths to action nodes, the highest value is used as the path length.

Then, a list of root nodes, sorted by decreasing path length is built.

2.4.3 Event Graph Execution

When a packet is received, SecSip first updates the events defined in the definition blocks, as seen previously. Then, the execute function of each root node is called.

If the execution of a node causes a token to be produced, it is propagated to other nodes, using the list of links held by the executed node. The propagation is done by inserting the destination node into a triggered node list, sorted by decreasing path length.

Then, each node in the triggered node list is executed, which can cause new nodes to be added to the list, until the list is empty.

2.5 Actions Execution

Action nodes are leaves, or terminal nodes of the graph, they do not produce tokens, but instead they hold a list of rules, each of which can contain a condition, and one or more actions.

When an action node is executed, it will process its rules sequentially. First, it will check if a condition is present and if it is true, then it will execute each action defined in the rule.

The following action types exist:

- the **store** action is used to add a value to a collection, using the value of a SIP field of the current packet,
- the **assign** action is used to set the value of a singleton, state, counter, or timer variable, using either the value of a SIP field of the current packet, or a used defined value,
- the **apply** action is used to update the value of a counter or timer variable, using the logic inherent to the counter or timer types,
- the **drop** action is used to discard a packet, to prevent it from reaching its target.

Chapter 3

The Manager

The manager is a javascript web application, served by SecSip using the nanohttp library (which is part of the csoap[1] library), that runs in an internet browser. It allows to view, edit and save SecSip configurations.

It is built using the JavaScriptMVC[2] framework, which facilitates object oriented programming, and provides an infrastructure that uses the model-view-controller design pattern.

It relies on a XML Schema to express and verify the grammar used to create and edit configurations.

The manager communicates with SecSip using SOAP, for operations on the configurations.

These aspects of the manager will be detailed un the following sections.

3.1 Design of the Manager

The manager is built on the model-view-controller design pattern.

The following models have been built to represent the information that the manager works with:

- the **Config** model handles communications with SecSip, and deals with configurations as a whole. It has methods to get and send a configuration to SecSip, and copy a configuration onto another one,
- the **Rule Element** model represents a part of a configuration. In the manager, the representation of a configuration is similar to an XML tree: it has a Rule Element at its root, and this Rule Element can contain attributes, a text value, and other Rule Elements. The Rule Element model also provides methods to serialize a tree of elements to XML, and to deserialize an XML Document to a tree of Rule Elements,
- finally, the **Constraint** model represents all constraints that have been expressed in the XML Schema. It checks that mandatory children and attributes of Rule Elements are there and in the correct order, that no invalid elements are present,

[1] `http://csoap.sourceforge.net/`
[2] `http://javascriptmvc.com/`

and it can also test reference and uniqueness constraints that can be present in XML Schemas.

To render those models to the user, a couple of views have been created, and for each view, a controller has been added, that interacts with the models:

- the **Config** view and controller consist of the menu bar at the top of the manager window, and link each button in the menu to one of the methods defined in the Config model (new congig, get, send, copy config),

- the **Root Element** view and controllers are used to display configuration inconsistencies and errors detected by constraint that are defined at the root of the XML Schema, or that span over multiple VeTo blocks,

- the **Tree** view and controller are used to render and interact with the list ot VeTo blocks that appears at the left of the manager window. The main function of this list is to allow the user to select individual blocks to view and edit them, and it also allows to create new blocks, and to delete and reorder existing blocks,

- and finally, the **Element** view and controller are used to display and interact with Rule Elements, which represent the actual content of the configuration. They allow to view and edit the attributes and values of an element and its tree of children, to add and delete child elements, and they also show unmatched constraints and other errors, at each node's level. An XML view is also available, which displays and allows to edit a tree of Rule Elements as an XML Document.

3.2 XML Schema and Constraints

The manager is actually a graphical XML editor, and it uses an XML Schema that dictates what a configuration should look like.

When adding, editing or deserializing configuration elements, the constraints expressed in the schema are checked, and if they don't match (for instance: the structure of the configuration does not follow the schema, some unknown elements are contained in the configuration, or some references point to non-existing block labels), error messages will be displayed to the user.

It also allows to make the manager interface actually usable: when adding a block, an element or an attribute, only those allowed by the schema are proposed to the user.

3.3 The SOAP Binding

The manager communicates with SecSip through SOAP requests, using the web browser ability to send HTTP requests.

The Config model contains methods that will form valid SOAP requests, and that will extract configuration representations from SOAP responses.

These methods will then call methods of the Rule Element model that will handle the serialization and deserialization of these data.

3.4 Supported Browsers

The SecSip manager has been successfully tested with Chrome 5, Firefox 3.6, Internet Explorer 8, Opera 10, and Safari 4.

"Pretty" reindentation of the XML view is known not to work in Internet Explorer 8 and Opera 10.

Bibliography

[1] Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: interval-based event specification and detection for active databases. *Data Knowl. Eng.*, 59(1):139–165, 2006.

[2] Abdelkader Lahmadi and Olivier Festor. Secsip: a stateful firewall for sip-based networks. In *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, pages 172–179, Piscataway, NJ, USA, 2009. IEEE Press.

[3] Abdelkader Lahmadi and Olivier Festor. Veto: An exploit prevention language from known vulnerabilities in sip services. In *IEEE/IFIP Network Operations and Management Symposium, NOMS 2010, 19-23 April 2010, Osaka, Japan*, pages 216–223, 2010.

Appendix A

TODOs

Implement Context Blocks Context blocks are currently not implemented in SecSip. There are two things to do: add a key-value based storage in rulespec to store context information, and filter definition blocks based on the ip addresses or ranges stored in the context blocks, so that only events that matter to the contexts that apply to a specific SIP dialog are triggered for this dialog.

Interface with Netconf and Yang To be able to control and monitor SecSip using Netconf, the SecSip Schema will need to be slightly adapted, so that it can also be expressed in Yang.

This means that the XML serializers and deserializers in SecSip will also need to be modified.

Then, Netconf interfaces must be implemented into SecSip (the SSH transport being mandatory, as per RFC 4741).

Secure the Access to the Manager Access to the manager could be secured using HTTPS and HTTP authentication, which are built in the nanohttp library, but are not currently used.

Fix the UA Commands The currently available UA commands are a bit too simple for everyday administration tasks. Existing commands will have to be improved, and new commands might need to be added, for the UA to be usable on its own.

Improve the Manager The manager currently only allows to modify the configuration. It could be enhanced with the ability to view real-time statistics of the firewall, or to allow per-dialog actions.

Fix the Configure Script The configure script could be made a bit cleaner, in terms of dependancy handling. Flags "–with-pcap", "–with-nfqueue" and "–with-csoap" would need to be added, or reworked, so that it becomes possible to build SecSip with or without them.

Fix Memory Leaks Currently, there are still memory leaks in SecSip, mainly because memory allocated for packet filtering is never freed, and because old dialogs are never flushed either.

Better Error handling when parsing rules When changing the configuration using the SOAP interface while SecSip is running, it is possible to submit a faulty configuration, and the engine converting rulespecs to rules might hit an error, and stop in the middle of the process, leaving "holes" in the firewall.

We would need to add a temporary rulespec repository, to which the running config is copied before the new one is imported, and that SecSip can revert to if an error is found.

Implement List and Bag Types Currently, "set" is the only implemented collection type. "list" and "bag" need to be implemented.

Implement Singleton Type The "singleton" type is currently not clearly defined, and not implemented at all.

Review Timer Variables Although there is a "timer" implementation in SecSip, the documentation might be a bit too vague. We would need to add examples of timer usage, and clearly define how to declare a timer, with the meaning of its parameters, and how it should behave.

Add logic to assertion Currently, a rule condition consists only of one assertion. We could add multiple condition support per rule, and logic between conditions (and, or, not).

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803