

Lightweight Small Footprint Security Module (SEM) - programmer's and user's manual

version 1.00.00 - June 30, 2007



<http://www.positif.org>

Contents

1	Introduction	3
2	Installing Scratchbox	3
2.1	Getting the packages	3
2.2	Starting Scratchbox	3
2.3	Creating cross-compilation target for ARM	3
3	Installing Snort in Scratchbox	4
4	Connection between iPAQ and Desktop PC	4
5	Matlab	5
5.1	Installation of SOM Toolbox	5
5.2	Matlab scripts	5
5.2.1	snortsoms	5
5.2.2	importtrainhttps	6
5.2.3	importtesthttps	6
5.3	Matlab functions	6
5.3.1	createsoms hitc(data all payload, train all payload, dataobject, train, count)	7
5.3.2	classifybmus(data, map)	7
5.3.3	classifyhits(data, map, bmuclass)	7
5.3.4	mergedata(data all, data, results, errorthreshold))	7
5.3.5	exportsoms(dataobject, levelinfo, ports, count)	8
6	SNORT	8
6.1	Introduction to the source tree	8
6.2	Minimizing Snort and plugin registration	8
6.3	Snort SOM plugins	9
6.4	New files	9
6.5	SOM plugin library	9
6.5.1	Constants	9
6.5.2	Constants	10
6.5.3	Functions	10
6.5.4	The trainings plugin	11
6.5.5	The testing plugin	11
6.6	Starting the plugins	11

1 Introduction

This should be a short guide to port Snort to an ARM architecture. In our case we used Snort 2.4.2, Scratchbox 1.0.2. Our aim architecture was an iPAQ 3600. Most of our information we took from the following Internet resources:

- <http://www.scratchbox.org/documentation/user/scratchbox-1.0/html/installdoc.html> "A little Documentation about installing Scratchbox."
- <http://handhelds.org/moin/moin.cgi/UsbNet> "Setting up an USB network to the iPAQ."

2 Installing Scratchbox

2.1 Getting the packages

Under Debian/Linux the installation of the packages works the following way: You will need root privileges for this part of the Scratchbox installation. Add this line to the `/etc/apt/sources.list` file:

```
deb http://scratchbox.org/debian ./
```

Update the package list with command:

```
# apt-get update
```

Install packages:

```
# apt-get install <packages>
```

After downloading Scratchbox will be unpacked to `/scratchbox` directory and the installation procedure will ask you some questions about the group and user accounts. Default group to Scratchbox users is `sbox`. Group can be renamed but default should be fine unless you have LDAP or similar network authentication scheme. If network authentication is used using an existing group is recommended.

Users who will be using Scratchbox should be added using command:

```
# sb-adduser <username>
```

It will automatically include users to the Scratchbox group, create user directories under `/scratchbox/users` directory and mount several directories (`/dev`, `/proc`, `/tmp`) under user directory.

2.2 Starting Scratchbox

If you were logged into the Scratchbox machine before you were added as a Scratchbox user, you may need to re-login to your machine, so that you get `sbox` group privileges needed for running Scratchbox. You can check this by running the following command:

```
$ groups
```

If it prints out the `sbox` group name, you're ready to start Scratchbox. Start Scratchbox with command:

```
$ /scratchbox/login
```

When you login for the first time, Scratchbox copies the `terminfo` terminal capability database to your Scratchbox home directory (`/scratchbox/users/username/home/username/`) and creates the default `HOST` target.

2.3 Creating cross-compilation target for ARM

When you're logged in into the Scratchbox start the `sb-menu`: `[sbox-HOST:] > sb-menu` Here you have to configure the setup options:

- create a new target and specify a name
- next select the tool chain for creating ARM binaries (`arm-gcc-3.3.4-glibc-2.3.2`)

- select none of the development tools

Select the qemu-arm emulator. Now the setup is done, but sb-menu asks if we want to extract a rootstrap and/or install system files on the target. Answer no to the rootstrap question but choose to install files.

Select the qemu-arm emulator. Now the setup is done, but sb-menu asks if we want to extract a rootstrap and/or install system files on the target. Answer no to the rootstrap question but choose to install files.

You can install the C-library (and related binaries) provided by the toolchain, some standard config files in /etc, config files required by the selected devkits (we have not selected any) and some target-specific binaries.

The default selection is fine, so you can just press enter to install them. Everything is now ready and we can activate the target by answering yes to the last question.

Scratchbox is now ready for cross-compilation for ARM with the help of the QEMU emulator.

3 Installing Snort in Scratchbox

First we need two addition packages: libpcap and pcre. Here we take the same versions which are installed on the iPAQ. In our case libpcap-0.8.3 and pcre-4.4. Those packages have to be compiled and installed in the Scratchbox. Normally with:

```
$ ./configure
$ ./make
$ make install
```

After installation of these packages snort can also be compiled and installed in the Scratchbox without any problem.

```
$ ./configure
$ ./make
$ make install
```

Last thing you have to do is to put the binary snort executable on the iPAQ and start it there. For instance copy the snort binary with ssh through the USB net. A little instruction for setting up the USB connection follows in the next section.

4 Connection between iPAQ and Desktop PC

First you have to enable the USB Net in kernel (USB Net is already enabled in most of the distributions). For the 2.6 kernel you need the following settings:

```
Device Drivers ---> USB support ---> USB Network Adapters --->
<M> Multi-purpose USB Networking Framework
<M> Simple USB Network Links (CDC Ethernet subset)
<M> [*] Embedded ARM Linux links (iPAQ, ...)
```

Recompile the kernel and install the new modules as usual.

Today most Linux distribution are hot pluggable so when you connect the iPAQ via USB the necessary kernel modules and network device get loaded automatically.

But if you want to do this by hand her a little instruction:

Load the kernel module:

```
$ modprobe usbnet
```

Connect the iPAQ via USB to your computer. After that look for an USB network device:

```
$ iwconfig usb0
```

If no device will be found, something is wrong with your connection. Maybe `dmesg` helps.

Normally the network device on the iPAQ will start automatically. At last setup the IP addresses for your USB network devices at your iPAQ and your computer. The default IP address at the iPAQ is 192.168.0.201. Maybe your LAN works with the same range of IP addresses, so we suggest to use 192.168.129.201 for the iPAQ and for your computer 192.168.129.101.

If the network devices are running with the right IP addresses you should be able to ping and connect via ssh to the iPAQ.

The next sections of this document should give a short overview of implemented and used Matlab functions for the SOM training plugin. Afterwards the source code for the new Snort plugins will be explained.

5 Matlab

For creating our Self-Organizing-Maps (SOM) we used Matlab 7.1 and the freely available SOM Toolbox 2.0 from the Laboratory of Computer and Information Science, Helsinki University of Technology (<http://www.cis.hut.fi/projects/somtoolbox/>).

This toolbox provides all needed functions for creating SOMs for our Snort plugin.

5.1 Installation of SOM Toolbox

To install the SOM Toolbox, just extract the downloaded archive from the homepage (see link above) into Matlabs toolbox directory. This directory can be found in the Matlab 7.1 main directory. After extraction, the paths to the SOM Toolbox directory have to be set additionally. Open Matlab, go to File/Set Path... and add the folder SOM Toolbox to the paths. Save settings. Now the SOM Toolbox is integrated into Matlab and can be used. To ensure that everything went right, try to start one of the four available SOM Toolbox demos: `som demo1`, `som demo2`, `som demo3` or `som demo4`.

5.2 Matlab scripts

This section describes all Matlab scripts which were written to make the handling of our data easier.

5.2.1 snortsoms

This script generates all files which are needed for our Snort SOM plugin. To get data into Matlab files it uses `importtrainhttps`.

Further, some objects get initialized which are going to store results of functions called in a later step. Now all data which was imported before is loaded into the workspace of Matlab so that it is possible to work with it. This loaded data contains the whole training data set in one matrix, the training data set per connection gathered in cells and the list of ports. Next step is to call the function `createsoms_hitc` with the arguments:

data_all_payload contains all training data in one matrix where one row is the byte histogram of one payload packet.

train_all_payload contains all training data in cells. Each cell represents a port and is a matrix containing all payload packet histograms for this specific port.

dataobject will contain the whole SOM tree.

train is a flag which indicates that a SOM shall be created.

count is a counter which counts through all recursive calls of the function `createsoms` hitc.

This function returns now the complete SOM tree as `dataobject`.

To export the information retrieved in the last step the function `exportsoms` is executed. Therefore the following arguments are used:

dataobject now containing the whole SOM tree.

levelinfo will afterwards contain a string representation of the structure of the SOM tree.

ports contains all ports to classify.

count is a counter which counts through all recursive calls of the function `exportsoms` and serves as unique identification number for each SOM layer.

This function now returns the string representation of the SOM tree as `levelinfo`. Last part of the script is to extract the information out of `levelinfo` and write it into the file `somconfig`.

5.2.2 `importtrainhttps`

This script imports all training data which was collected before by the Snort training plugin. The data has to be situated in the subdirectory `data/train`. All files in this directory get loaded separately and get stored in `data` all payload and in `train` all payload.

In `data` all payload data of all files is concatenated to one big matrix, in `train` all payload data of each file is stored separately in one cell object as matrix. The data doesn't have to get normalized because this is done by the Snort training plugin. After all also the names of the ports of the data get stored in the object `ports`.

Finally, the script writes all imported data to files. `importtrainhttps` creates the following files in the subdirectory `data`: `data_all_payload.mat`, `train_all_payload.mat` and `ports.mat`.

5.2.3 `importtesthttps`

This script imports all testing data which was collected before by the Snort training plugin.

The data has to be situated in the subdirectory `data/test`. Testing data and training data are the same. The difference is the representation. Testing data is stored in files per session. Training data is stored in files per port. The testing data is only used for testing purpose in Matlab and is not necessary for creating the SOM files for Snort. The script stores data per session as a single matrix and combines all sessions as cells in one object called `testing_all_payload`.

Further, the index of the corresponding port for each session gets stored in the object `conn_labels_testing`. The filename of each session gets stored in `conn_names_testing`.

Finally, the script writes all imported data to files. `importtesthttps` creates the following files in the subdirectory `data`: `testing_all_payload.mat`, `conn_labels_testing.mat` and `conn_names_testing.mat`.

5.3 Matlab functions

This section describes all the Matlab functions which are needed to create SOMs and to export information for our Snort SOM testing plugin. Most of the functions were written by Peter Teufl [2] [1] during his master thesis. Thanks to him, because we were able to adopt some parts.

5.3.1 `createsoms hitc(data all payload, train all payload, dataobject, train, count)`

This is the main function of the whole program. It constructs a `dataobject` which contains all information. The first time `createsoms hitc` is executed the argument `dataobject` is empty. It is filled in each recursive execution of this function. First step of `createsoms hitc` is to create a SOM with `data all payload`. This is done by executing `som make` which is provided by SOM Toolkit. Next step is the creation of a `bmuclass` payload by applying the the function `classifybmus`. Last important step of the first phase is calling `classifyhits` which returns the classification results for `train all payload`. Now the data is merged to a new set to reduce the error rate until it is low than error. With this new data sets the function `createsoms hitc` is executed again. All gathered information in each call of `createsoms hitc` is stored in one and the same `dataobject`. The following listing shows where data is stored in `dataobject`:

`dataobject{6}` contains the map

`dataobject{7}` contains the `bmulist` for this map

`dataobject{10}` contains lists of classes which can be distinguished by using this SOM. Each sublist, if it has more than one number in it, defines the different classes which can not be separated in this SOM. For this non-separable classes another SOM exists which can be found in **`dataobject{11}`**.

`dataobject{11}` contains `dataobjects`, too. For each list from above which has more than one entry there exists a corresponding `dataobject`. These `dataobjects` have the same structure.

Finally, `dataobject` is returned by the function and contains all information of the whole SOM tree. This function was written by Peter Teufl [2] [1].

5.3.2 `classifybmus(data, map)`

This function takes as arguments an already trained SOM and a set of data. It creates a matrix with n rows, where n is the number of best matching units (bmu), and m columns, where m is the number of different classes of this SOM. This matrix provides information about how often data is classified by each bmu. So it is possible to find out which bmu is a classifier for which class. The function just goes through all data and uses the function `som_hits` which is provided by the SOM Toolkit. `som_hits` takes one subset of `data` and `map`. It returns a vector with the length n for each subset. Each element of the vector belongs to one bmu and holds the number of hits the `data` subset has caused. After the for-loop the values of the matrix containing the hits is normalized. In our implementation data is `train_all_payload`. This function was written by Peter Teufl [2] [1].

5.3.3 `classifyhits(data, map, bmuclass)`

Besides the set of data and the SOM this function takes also `bmuclass`, which was computed before by the function `classifybmus`. The main purpose of `classifyhits` is to get for each data packet the corresponding class label. It goes through all data packets and tries to find out which bmu it hits. This is done by using `som_bmus` which is part of the SOM Toolkit. Last important step of this function is to find out to which port this bmu belongs. These values are collected and returned as result payload. This function was written by Peter Teufl [2] [1].

5.3.4 `mergedata(data all, data, results, errorthreshold)`

This function calculates an error out of the results of the function `classifyhits`. If this error bigger than `errorthreshold` data is merged to a new set `datanew` is created. This `datanew` represents a set of data which was not able to be separated with this SOM. Afterwards this `datanew` will again be used for executing `createsoms hitc`. `mergedata` returns a list of non-separable data, a matrix `datanew_all` where the whole merged data is stored and `datanew` where the merged data is stored in cells per port. This function was written by Peter Teufl [2] [1].

5.3.5 exportsoms(dataobject, levelinfo, ports, count)

This function exports all the information which is gathered in *dataobject*. First, it selects the map out of the *dataobject*.

This *map* is nothing else than the first SOM. Further, *bmulist* and *list* are extracted. The *count* argument gets the new *id* of this SOM. In the next few lines, strings of filenames are created which will be used later for representation of extracted information in string format and for output file creation. After all this is done the for-loop goes through all sublists out of *list* and writes all elements of each sublist into *tempstr* which is nothing else than a string representation for our somconfig file. The numbers written in each sublist correspond to the specific ports of the argument *ports*. This is the reason why *new_ports* is created. It holds only those corresponding ports. Each sublist out of *list* stands for one class of the actual SOM.

If one of the sublists has more than one number in it than these numbers of ports are not separable in this SOM. In this case there exists another SOM which is able to distinguish between ports of this sublist. For this sublist *exportsoms* is recursively executed with the corresponding subdataobject which contains all necessary information for this sublist.

By executing *exportsoms* recursively the program goes through the whole SOM tree and extracts all information into *levelinfo*. *count* is incremented by one and will be the new *id* for the next SOM. After the for-loop has finished all information of one SOM is gathered as one string line in *tempstr*.

Such a line can look as follows: `map1 bmulist1 1 143:2 22:2 23:2 5900:0 88:0. map1` and `bmulist1` specify the filenames for the corresponding map and *bmulist* with the *id* 1. 1 is the *id* 143:2 stands for the port 143 which can not be classified in this map but in the map with the *id* 2.

All ports which have a 0 can be classified with the actual map.

tempstr is finally stored in *levelinfo* at the position specified by its *id*. Further, all somvectors of this SOM are extracted out of map. Last step of *exportsoms* is to create files containing all *somvectors* and the *bmulist* of the actual *map*. All files are created in the subdirectory `export`.

6 SNORT

6.1 Introduction to the source tree

Snort is very modular with many different plugins for intrusion detection. That's why snort has a core for the main functionality and a special interface for plugins registration. For our project we decided to use current version 2.4.2. The main source files for the core can be found in the directory `./src/`. The following directories `./preprocessors/`, `./detection-plugins/`, and `./output-plugins/` are commonly used for writing plugins.

6.2 Minimizing Snort and plugin registration

For using Snort with SOM plugins it is not necessary to build up the application with the whole range of detection-, preprocessor-, and output-plugins. The source structure of Snort is modular and so it is possible to disable most of the plugins. So the size of the Snort binary can be reduced drastically.

The file `plugbase.h` contains the main plugin registration. For registration, two steps are necessary:

- Every plugin needs its own header file and must be included at the top of the file. For instance:

```
#include "preprocessors/spp_somt.h"
```

- As mentioned above there are three different types of plugins. Each type has its own initializing function: `InitPlugIns()` for detection-plugins, `InitPreprocessors()` for preprocessor-plugins, and `InitOutputPlugins()` for output-plugins. In one of these three functions the plugin setup function must be called for final registration. For the SOM trainings plugin the function `SetupSomT()`; will be called in function `InitPreprocessors()`.

To disable one plugin it is only necessary to uncomment the line with `#include` and the plugin setup function. Its important, that not every plugin can be disabled. There are some basic plugins that Snort needs for running: `sp_flowbits.h`, `spo_log_tcpdump.h`, and `spo_alert_full.h`. After disabling the plugins the Snort binary is approximately 1.5 MB smaller than with all plugins enabled. Naturally the file size of the binary depends on the platform Snort is compiled for.

6.3 Snort SOM plugins

The aims of the two new plugins are to collect data for training SOM maps with Matlab and afterwards testing network traffic with the trained SOM.

6.4 New files

- `./preprocessors/som_lib.h` This file defines structures (SomMap and SomVector), constants, and functions which are used in both SOM plugins.
- `./preprocessors/som_lib.c` Implementation of the basic operation on the structures SomMap and SomVector.
- `./preprocessors/spp_somc.h` Header definitions for the training plugin.
- `./preprocessors/spp_somc.c` Implementation of the training plugin.
- `./preprocessors/spp_somt.h` Header definitions for the testing plugin.
- `./preprocessors/spp_somt.c` Implementation of the testing plugin.

6.5 SOM plugin library

The library consists of the files `som_lib.h` and `som_lib.c` and provides basic functionality for the both plugins. Firstly, the header file description:

6.5.1 Constants

- `#define SOM_BUFFER_SIZE` The temporary buffer size for string operations.
- `#define SOM_VECTOR_SIZE` The vector size of one best matching unit. It has the typical size of the byte spectrum vector.
- `#define SOM_MAX_MAPS` How many SOMs could be loaded. Only used in the SOM testing plugin.
- `#define SOM_CLASS_SIZE` The maximum count of SOM classes. Necessary for the structures SomVector and SomMap. See later.
- `#define SOM_FILE_DELIMITER` Delimiters for string tokenizer. Used in both plugins for file parsing.
- `SOM_CONFIG_FILENAME` The main configuration file for loading SOMs. Only used in the SOM testing plugin.
- `SOM_CONFIGFILE_UNCOMMENT` Char for uncommenting lines for the SOM configuration file.
- `SOM_NOGROUP` Initial value for the variable bestgroup in structure SomVector.
- `SOMC_MIN_DISTANCE_THRESHOLD` Value for the minimal euclidean distance threshold. See next constant.
- `SOMC_MODE_ALERT` The SOM testing plugin can be run in two modes. If this constant is set to 0 the best matching unit will be outputted. Otherwise if the constant is set to 1 an alert will occur when the calculated minimal distance of this packet is greater than the `SOMC_MIN_DISTANCE_THRESHOLD`.

- **SOM_PAYLOAD_TRAIN_DIRECTORY** In this directory the SOM training plugin saves all necessary files for training SOMs with Matlab.
- **SOM_PAYLOAD_TEST_DIRECTORY** In this directory the SOM training plugin saves all necessary files for testing SOMs with Matlab.

6.5.2 Constants

The structure `SomVector_` defines all necessary data of a SOM unit. A SOM consists of an array of `SomVector_s`. More details later.

```
struct SomVector_ {
float vector[SOM_VECTOR_SIZE];
struct SomVector_ *next;
float hits[SOM_CLASS_SIZE];
int16_t bestgroup;
};
```

The variable `vector` of type `float` contains the vector for one SOM unit. In that case the size of the array is 256. To build up a whole SOM the pointer `next` represents the memory address to the next vector. This is an implementation of simple linked lists. The variable `hits` stores the information how often this SOM unit is hit by a group of classes. In `bestgroup` the group with the best hit is saved.

The structure `SomMap_` stores all data for one SOM and allows to build up a SOM tree. The array `class_som` saves the pointer to the next SOM level. So each group of classes can have its more detailed SOMs.

```
struct SomMap_ {
struct SomVector_ *somvector;
u_int16_t id;
char *class_label[SOM_CLASS_SIZE];
u_int16_t class_som_ids[SOM_CLASS_SIZE];
struct SomMap_ *class_som[SOM_CLASS_SIZE];
struct SomMap_ *all_maps[SOM_MAX_MAPS];
};
```

In the variable `somvector` the first element of the linked list is stored. `id` contains the id of the current SOM. `class_label` and `class_som_ids` stores the ids and labels of all classes in the SOM. In the root node of the SOM tree a pointer `all_maps` to all used SOMs is stored. It helps to access the maps more easily.

6.5.3 Functions

- `SomVector *newSomVector ()` creates the structure for a new SOM vector.
- `SomVector *loadSomVector (char* filename_som, char* filename_bmulist)` loads the data for a whole SOM. The function needs two files. One for the SOM and one for the hits to a SOM unit. Each row of both files represents a SOM unit.
- `void deleteSomVector (SomVector *somvector)` deletes the structure for a new SOM vector.
- `SomMap *newSomMap ()` creates the structure for a new SOM.
- `SomMap *loadSomMap (char* filename)` loads the complete SOM tree. As parameter the filename to the main SOM configuration will be passed.

- `void deleteSomMap (SomMap *somap)` deletes the structure for a new SOM.
- `void packetSomVector (Packet *p, SomVector *somvector)` calculates the byte spectrum for one TCP packet.

6.5.4 The trainings plugin

This plugin collects TCP data for training the Self-Organizing Maps. The collected data will be outputted to two types of files. One file type saves all data for one TCP port, which later will be used to train SOMs with Matlab. The other type stores data for every TCP connection which later will be used to test SOMs with Matlab. The structures of both files are the same: every line is a list of float numbers separated by spaces, which are a representation of the byte stream vector. The directories for saving these files are stored in the two constants `SOM_PAYLOAD_TRAIN_DIRECTORY` and `SOM_PAYLOAD_TEST_DIRECTORY` in `som_lib.h`. The following list describes the main functions of this plugin:

- `void SomTInit (u_char * args)` This function will be called from the Snort plugin registration function and registers the preprocessor-plugin, which is named SomC. As argument the command line parameters will be passed.
- `void SomTProcess (Packet *p, void *context)` calculates the byte spectrum from the TCP packet `p` and saves this data to two files. One for training SOMs and one for testing.

6.5.5 The testing plugin

The SOM testing plugin can be run in two modes. If this constant `SOMC_MODE_ALERT` is set to 0 the best matching unit will be outputted. Otherwise if the constant is set to 1 an alert will occur when the calculated minimal distance of this packet is greater than the `SOMC_MIN_DISTANCE_THRESHOLD`. A description about the file format for all import files can be found in the section 2.3.5. The following list describes the main functions of this plugin:

- `void SomCInit (u_char * args)` This function will be called from the Snort plugin registration function and register the preprocessor-plugin, which is named SomC. As argument the command line parameters will be passed.
- `void SomCProcess (Packet *p, void *context)` is the main function for processing TCP data and calculating the best matching SOM unit. Firstly, the function determines the byte spectrum from the TCP packet. Secondly, the function calls `min_distance()`.
- `void SomCCleanExitFunction (int signal, void *foo)` frees the used memory space, when exiting Snort. In this case the loaded SOM is freed. At this point the parameters are just dummies.
- `float min_distance(SomVector *somvector_p, SomMap *somap, u_int16_t depth)` is the most interesting function. It searches the best matching unit in `somap` for `somvector`. If the best matching unit belongs to a group of classes the function calls itself for an other recursion. As parameter the SOM which belongs to the group of classes will be passed. The recursion ends when a best matching unit with no group assignment will be found.
- `float euclidean_distance (SomVector *v1, SomVector *v2)` calculates the euclidean distance between two vectors.

6.6 Starting the plugins

The two new SOM plugins are preprocessor plugins and must be started with special rules. Thats why we created two different rule files for each of the plugins. The rule file `somt.rules` for the training plugin contains the following line:

```
preprocessor somt
```

For starting the testing plugin the file `somc.rules` must have the following line:

```
preprocessor somc
```

For instance to start Snort with one of the plugins execute this command:

```
./snort -c <somt.rules|somc.rules> -i eth0
```

In this example Snort is listening at the network interface `eth0`. For other Snort arguments read the manual.

Important, dont forget that the SOM training plugin needs special directories for creating the testing data. See section 3.5.5. Even the SOM testing plugin needs some files for running. See section 2.3.5 for more information.

References

- [1] Peter Teufl Udo Payer, Stefan Kraxberger. Traffic classification using self-organizing maps. *Proceedings INC 2005*, 05 - 07 July 2005.
- [2] Stefan Kraxberger Udo Payer. Som-based lightweight policy verification. *Proceedings TERENA 2006*, 15 - 18 May 2006.