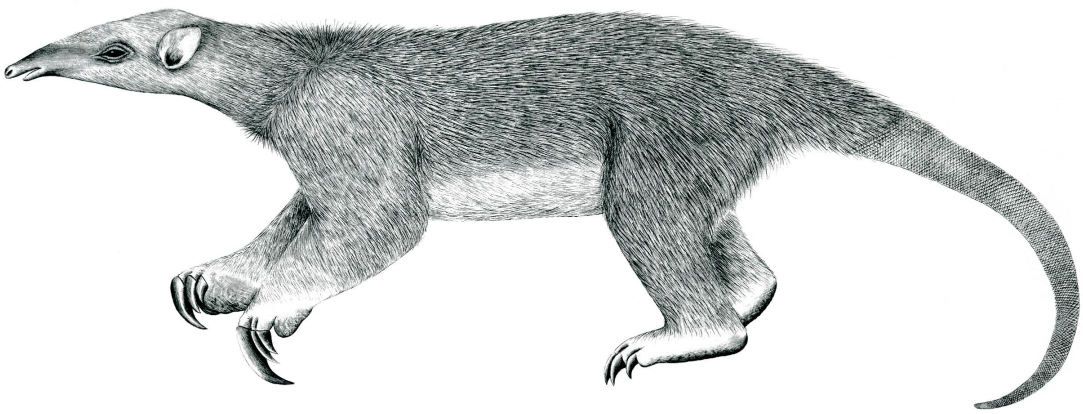
Tranalyzer2

Creating a Custom Plugin

Developer Guide



Tranalyzer Development Team

Contents

Copyright © 2008–2024 by Tranalyzer Development Team

1 Creating a Custom Plugin

A plugin is a shared library file comprising of special functionality. Tranalyzer2 dynamically loads these shared libraries at runtime from the *~/.tranalyzer/plugins* directory in the user's home folder. Therefore Tranalyzer2 is available for users if being installed in the */usr/local/bin* directory while the plugins are user dependent. To develop a plugin it is strongly recommended that the user utilizes our special "t2plugin" script. This script uses the plugin skeleton "t2PSkel" to create a new custom named plugin. It is available in the *scripts/* folder. The script copies only the required files. Therefore it is recommended to upload the newly created folder to a SVN/GIT repository before running `./autogen.sh` (alternatively, `./autogen.sh -c` can be used to clean up automatically generated files that should not be committed). The skeleton contains a header and a source file comprising of all mandatory and optional functions as well as a small `HOWTO` file and a script to build and move a shared library to the plugins folder.

1.1 Plugin Name

Plugin names should be kept short, start with a lowercase letter and only contain characters in the following ranges: a–z, A–Z, 0–9, _. In addition, each "word" should start with an uppercase letter, e.g., `pluginName`.

1.2 Plugin Number

The plugin number (or order) influences when a plugin is to be loaded (useful if a plugin depends on another one). This number should consist of three digits and be unique. The plugin orders used in your Tranalyzer installation can be listed with `$T2HOME/doc/list_plugin_numbers.sh` or `t2plugin -l`. As a rule of thumb, numbers greater than 900 should be kept for sink (output) plugins and numbers smaller than 10 for global plugins.

plugin range	description
000 - 099	Global
100 - 199	Basic L2/3/4 plugins
200 - 299	Service and routing
300 - 699	L7 protocols
700 - 799	Math and statistics
800 - 899	Classifier and AI
900 - 999	Output (sink)

1.3 Plugin Creation

To create a new plugin named *pluginName* with plugin order 123, run the following command from Tranalyzer's root folder:

```
./scripts/t2plugin -c pluginName -n 123
```

If no plugin number is provided, then the script will choose a random one that is not used by any other plugin.

A C++ plugin can be created by passing the additional '–cpp' option:

```
./scripts/t2plugin -c pluginName -n 123 -cpp
```


Copyright © 2008–2024 by Tranalyzer Development Team

1.3.1 autogen.sh

The `autogen.sh` script provides the `EXTRAFILES` variable, which is used to list extra files, such as lists of subnets, protocols, services, databases or blacklists, that the plugin needs in order to run. The files listed in this variable are automatically copied into the Tranalyzer plugin folder.

```
EXTRAFILES=(dependency1 dependency2)
```

The `CFLAGS` variable in `autogen.sh` can be used if a plugin requires specific libraries, compilation or linking flags, e.g., `CFLAGS="-lzip"`. In such a case, the `DEPS` variable can be used to list the dependencies, e.g., `DEPS="libzip"`.

1.4 Compilation

The plugin can then be compiled by typing `./autogen.sh`. For a complete list of options, run `./autogen.sh -h`

1.5 Plugin Structure

All plugins have the same global structures, namely, a comment describing the license of the plugin, e.g., GPLv2+, some includes, followed by the declaration of variables and functions. This section discusses the Tranalyzer callbacks which follows the elements already mentioned. Note that all the callbacks are optional, but a plugin **MUST** call one of the initialization macros.

First, a plugin MUST have one the following declarations:

- `T2_PLUGIN_INIT(name, version, t2_v_major, t2_v_minor)`
- `T2_PLUGIN_INIT_WITH_DEPS(name, version, t2_v_major, t2_v_minor, deps)`

For example, to initialize `myPlugin`:

```
T2_PLUGIN_INIT_WITH_DEPS("myPlugin", "0.9.3", 0, 9, "tcpFlags,basicStats")
```

The available callbacks are:

- `void t2Init()`
- `binary_value_t *t2PrintHeader()`
- `void t2OnNewFlow(packet_t *packet, unsigned long flowIndex)`
- `void t2OnLayer2(packet_t *packet, unsigned long flowIndex)`
- `void t2OnLayer4(packet_t *packet, unsigned long flowIndex)`
- `void t2OnFlowTerminate(unsigned long flowIndex, outputBuffer_t *buf)`
- `void t2PluginReport(FILE *stream)`
- `void t2Finalize()`
- `void t2BufferToSink(outputBuffer_t *buffer, binary_value_t *bv)` [Sink (output) plugins only]

Copyright © 2008–2024 by Tranalyzer Development Team

The following callbacks offer more advanced capabilities:

- `void t2BusCallback(uint32_t status)` [Not implemented]
- `void t2Monitoring(FILE *stream, uint8_t state)`
- `void t2SaveState(FILE *stream)`
- `void t2RestoreState(char *str)`

1.5.1 `void t2Init()`

This function is called before processing any packet.

1.5.2 `binary_value_t *t2PrintHeader()`

This function is used to describe the columns output by the plugin. Refer to Section 1.8 and the `BV_APPEND` macros.

1.5.3 `void t2OnNewFlow(packet_t *packet, unsigned long flowIndex)`

This function is called every time a new flow is created.

1.5.4 `void t2OnLayer2(packet_t *packet, unsigned long flowIndex)`

This function is called for every packet with a layer 2. If `flowIndex` is `HASHTABLE_ENTRY_NOT_FOUND`, this means the packet also has a layer 4 and thus a call to `t2OnLayer4()` will follow.

1.5.5 `void t2OnLayer4(packet_t *packet, unsigned long flowIndex)`

This function is called for every packet with a layer 4.

1.5.6 `void t2OnFlowTerminate(unsigned long flowIndex, outputBuffer_t *buf)`

This function is called once a flow is terminated. Output all the statistics for the flow here. Refer to Section 1.8 and the `OUTBUF_APPEND` macros.

1.5.7 `void t2BusCallback(uint32_t status)`

Currently not implemented.

1.5.8 `void t2Monitoring(FILE *stream, uint8_t state)`

This function is used to report information regarding the plugin at regular interval or when a `USR1` signal is received. `state` can be one of the following:

- `T2_MON_PRI_HDR`: a header (value names) must be printed
- `T2_MON_PRI_VAL`: the actual data must be printed
- `T2_MON_PRI_REPORT`: a report (similar to the plugin report) must be printed

Copyright © 2008–2024 by Tranalyzer Development Team

1.5.9 `void t2PluginReport(FILE *stream)`

This function is used to report information regarding the plugin. This will appear in the final report.

1.5.10 `void t2Finalize()`

This function is called once all the packets have been processed. Cleanup all used memory here.

1.5.11 `void t2SaveState(FILE *stream)`

This function is used to save the state of the plugin. Tranalyzer can then restore the state in a future execution.

1.5.12 `void t2RestoreState(char *str)`

This function is used to restore the state of the plugin. `str` represents the line written in t2SaveState().

1.5.13 `void t2BufferToSink(outputBuffer_t *buffer, binary_value_t *bv)`

This callback is only required for sink (output) plugins.

1.6 Error, warning, and informational messages

Tranalyzer2 provides several macros to report errors, warnings, information or simple messages:

`T2_PLOG()`	print a normal message (standard terminal colors)	`pluginName: message`
`T2_PINF()`	print an information message (blue)	`[INF] pluginName: message`
`T2_POK()`	print an okay message (green)	`[OK] pluginName: message`
`T2_PWRN()`	print a warning message (yellow)	`[WRN] pluginName: message`
`T2_PERR()`	print an error message (red)	`[ERR] pluginName: message`

Note that `T2_PERR` always prints to `stderr`, while the other macros print to `stdout` or `PREFIX_log.txt` if Tranalyzer `-l` option was used.

Their usage is straightforward:

```
T2_PLOG("pluginName", "message %d", 42);
```

Note that a trailing newline is automatically added.

1.7 Naming Conventions

In order to avoid conflicts with other plugins, prefix all your macros and publicly available functions and variables with the plugin name or a few letters uniquely identifying the plugin.

Copyright © 2008–2024 by Tranalyzer Development Team

1.8 Generating Output

The following macros can be used to declare and append new columns to the output buffer. The `BV_APPEND_*` macros are used to declare a new column with a given `name`, description `desc` and type. The `OUTBUF_APPEND_*` macros are used to append a value `val` of the given type to the buffer `buf`.

`BV` **Macro**	**Type**	**Corresponding** `OUBUF` **Macro**
Unsigned values		
`BV_APPEND_U8(bv, name, desc)`	`bt_uint_8`	`OUTBUF_APPEND_U8(buf, val)`
`BV_APPEND_U16(bv, name, desc)`	`bt_uint_16`	`OUTBUF_APPEND_U16(buf, val)`
`BV_APPEND_U32(bv, name, desc)`	`bt_uint_32`	`OUTBUF_APPEND_U32(buf, val)`
`BV_APPEND_U64(bv, name, desc)`	`bt_uint_64`	`OUTBUF_APPEND_U64(buf, val)`
`BV_APPEND_H8(bv, name, desc)`	`bt_hex_8`	`OUTBUF_APPEND_H8(buf, val)`
`BV_APPEND_H16(bv, name, desc)`	`bt_hex_16`	`OUTBUF_APPEND_H16(buf, val)`
`BV_APPEND_H32(bv, name, desc)`	`bt_hex_32`	`OUTBUF_APPEND_H32(buf, val)`
`BV_APPEND_H64(bv, name, desc)`	`bt_hex_64`	`OUTBUF_APPEND_H64(buf, val)`
Signed values		
`BV_APPEND_I8(bv, name, desc)`	`bt_int_8`	`OUTBUF_APPEND_I8(buf, val)`
`BV_APPEND_I16(bv, name, desc)`	`bt_int_16`	`OUTBUF_APPEND_I16(buf, val)`
`BV_APPEND_I32(bv, name, desc)`	`bt_int_32`	`OUTBUF_APPEND_I32(buf, val)`
`BV_APPEND_I64(bv, name, desc)`	`bt_int_64`	`OUTBUF_APPEND_I64(buf, val)`
Floating points values		
`BV_APPEND_FLT(bv, name, desc)`	`bt_float`	`OUTBUF_APPEND_FLT(buf, val)`
`BV_APPEND_DBL(bv, name, desc)`	`bt_double`	`OUTBUF_APPEND_DBL(buf, val)`
String values		
`BV_APPEND_STR(bv, name, desc)`	`bt_string`	`OUTBUF_APPEND_STR(buf, val)`
`BV_APPEND_STRC(bv, name, desc)`	`bt_string_class`	`OUTBUF_APPEND_STR(buf, val)`
Time values (timestamp and duration)[1]		
`BV_APPEND_TIMESTAMP(bv, name, desc)`	`bt_timestamp`	`OUTBUF_APPEND_TIME(buf, sec, usec)`
`BV_APPEND_DURATION(bv, name, desc)`	`bt_duration`	`OUTBUF_APPEND_TIME(buf, sec, usec)`
IP values (network order)		
`BV_APPEND_IP4(bv, name, desc)`	`bt_ip4_addr`	`OUTBUF_APPEND_IP4(buf, val)`
`BV_APPEND_IP6(bv, name, desc)`	`bt_ip6_addr`	`OUTBUF_APPEND_IP6(buf, val)`

[1] Time values use an `uint64` for the seconds and an `uint32` for the micro-seconds

Copyright © 2008–2024 by Tranalyzer Development Team

`BV_APPEND_IPX(bv, name, desc)`	`bt_ipx_addr`	`OUTBUF_APPEND_IPX(buf, version, val)`[2]

If more flexibility is required the following macros can be used:

- `BV_APPEND(bv, name, desc, num_val, type1, type2, ...)`
- `OUTBUF_APPEND(buf, val, size)`

1.8.1 Repetitive Values

A repetitive value consists of a `uint32` representing the number of repetitions, followed by the actual repetitions.

All the `BV_APPEND` macros introduced in the previous section can be suffixed with `_R` to represent a repetitive value:

`BV_APPEND_U8(bv, name, desc)` (non-repetitive) ⇒ `BV_APPEND_U8_R(bv, name, desc)` (repetitive).

In addition, the following `OUTBUF` macros are available for repetitive values:

`OUTBUF` **Macro**	**Description**	**Type**
`OUTBUF_APPEND_OPTSTR(buf, val)`	If `val` is `NULL` or empty, appends 0 (`uint32`) else appends 1 (`uint32`) and the string	`bt_string`, `bt_string_class`
`OUTBUF_APPEND_NUMREP(buf, reps)`	Appends the number of repetitions (`uint32`)[3]	

1.8.2 Column Names

Column names should be kept short and only contain characters in the following ranges: `_, a-z, A-Z, 0-9`. In addition, each "word" should start with an uppercase letter, e.g., `myCol2`. The `'_'` character should be used to name compound values, e.g., `field1_field2`. A good practice is to prefix each column name with the short name of the plugin, e.g., `ftpDecode` → `ftpStat, ftpCNum`.

1.8.3 More Complex Output

Refer to Section 1.8.

1.9 Accessible structures

Due to practical reasons all plugins are able to access every structure of the main program and the other plugins. This is indeed a security risk, but since Tranalyzer2 is a tool for practitioners and scientists in access limited environments the maximum possible freedom of the programmer is more important for us.

[2] Appends the IP `version` (`uint8`), followed by the IP. If `version` is 6, then calls `OUTBUF_APPEND_IP6(buf, val.IPv6.s6_addr[0]` else calls `OUTBUF_APPEND_IP4(buf, val.IPv4.s_addr`

[3] The correct number of values **MUST** then be appended.

Copyright © 2008–2024 by Tranalyzer Development Team

1.10 Important structures

A predominant structure in the main program is the flow table *flow* where the six tuple for the flow lookup timing information is stored as well as a pointer to a possible opposite flow. A plugin can access this structure by including the `packetCapture.h` header. For more information please refer to the header file.
Another important structure is the main output buffer `mainOutputBuffer`. This structure holds all standard output of activated plugins whenever a flow is terminated. The main output buffer is accessible if the plugin includes the header file `main.h`.

1.11 Generating output (advanced)

As mentioned in Section 1.8 there are two ways to generate output. The first is the case where a plugin just writes its arbitrary output into its own file, the second is writing flow-based information to a standard output file. We are now discussing the later case.

The standard output file generated by the `txtSink` plugin consists of a header, a delimiter and values. The header is generated using header information provided by each plugin, that writes output into the standard output file. During the initialization phase of the sniffing process, the core calls the `t2PrintHeader()` functions of these plugins. These functions return a single structure or a list of structures of type `binary_value_t`. Each structure represents a statistic. To provide a mechanism for hierarchical ordering, the statistic itself may contain one ore more values and one or more substructures.

The structure contains the following fields:

Name	Type	Description
`num_values`	`uint32_t`	Amount of values in the statistic
`subval`	`binary_subvalue_t*`	Type definition of the values
`name`	`char[128]`	Name of the statistic
`desc`	`char[1024]`	Description of the statistic
`is_repeating`	`uint32_t`	One, if the statistic is repeating, zero otherwise
`next`	`binary_value_t*`	Used if the plugin provides more than one statistics

The substructure `binary_subvalue_t` is used to describe the values of the statistic. For each value, one substructure is required. For example, if `num_values` is two, two substructures have to be allocated. The substructures must be implemented as a continuous array consisting of the following fields:

Name	Type	Description
`value_type`	`uint32_t`	Type of the value
`num_values`	`uint32_t`	Amount of values in the statistic
`subval`	`binary_subvalue_t*`	Definition of the values
`is_repeating`	`uint32_t`	one, statistic is repeating, zero otherwise

Compared to the `binary_value_t` representation two strings are omitted in the statistic's short and long description and the `*next` pointer but it contains a new field, the value type. Possible values for this new field are described in the enumeration `binary_types` defined in the header file *binaryValue.h*. If the field contains a value greater than zero the fields `num_values` and `subval` are ignored. They are needed if a subval contains itself subvalues. To indicate additional

Copyright © 2008–2024 by Tranalyzer Development Team

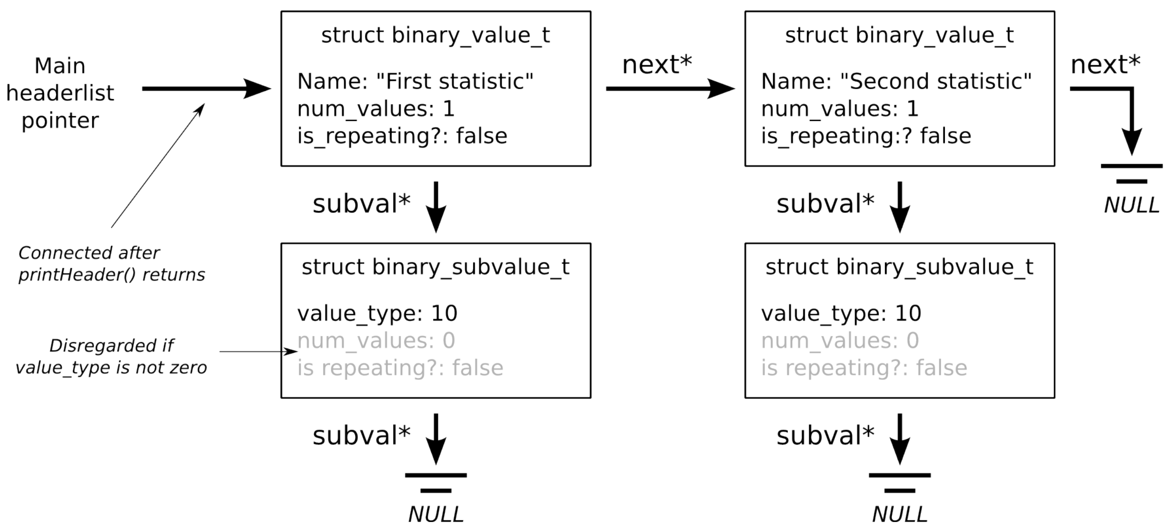
subvalues, the field `value_type` need to be set to zero. The mechanism is the same as for the `binary_value_t`.

The field `is_repeating` should be used if the number of values inside a statistic is variable; e.g. a statistic of a vector with variable length.

1.11.1 Examples

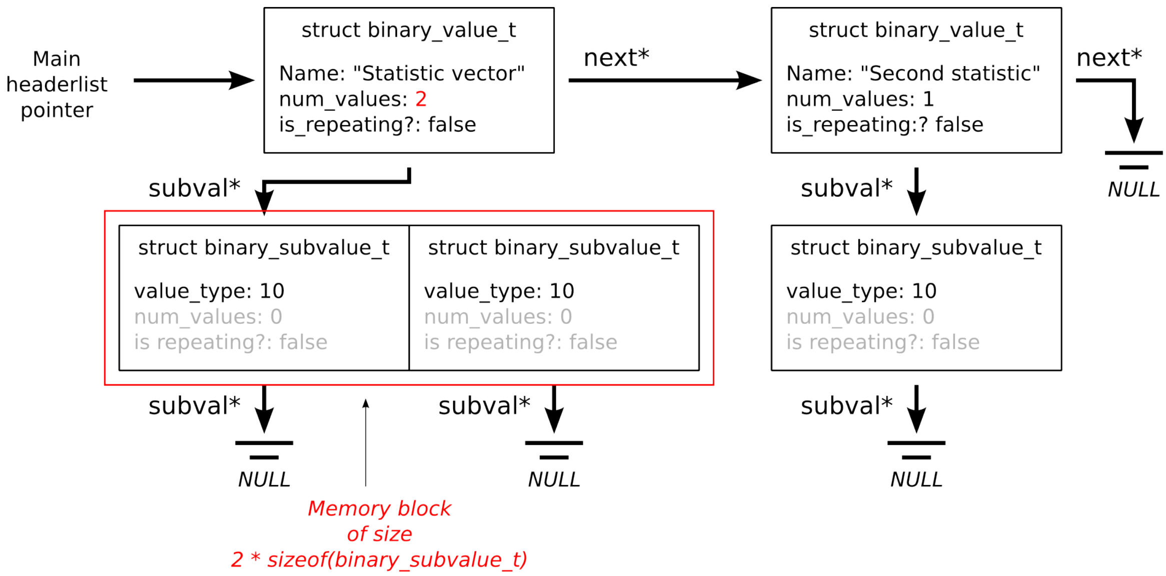
The following examples illustrate the usage of the said two structures:

Example 1: Two Statistics each containing a single value If a plugin's output is consisting of two statistics each having a single value it needs to pass a list containing two structures of type `binary_value_t`. Both structures contain a substructure with the type of the single values. The following diagram shows the relationships between all four structures:

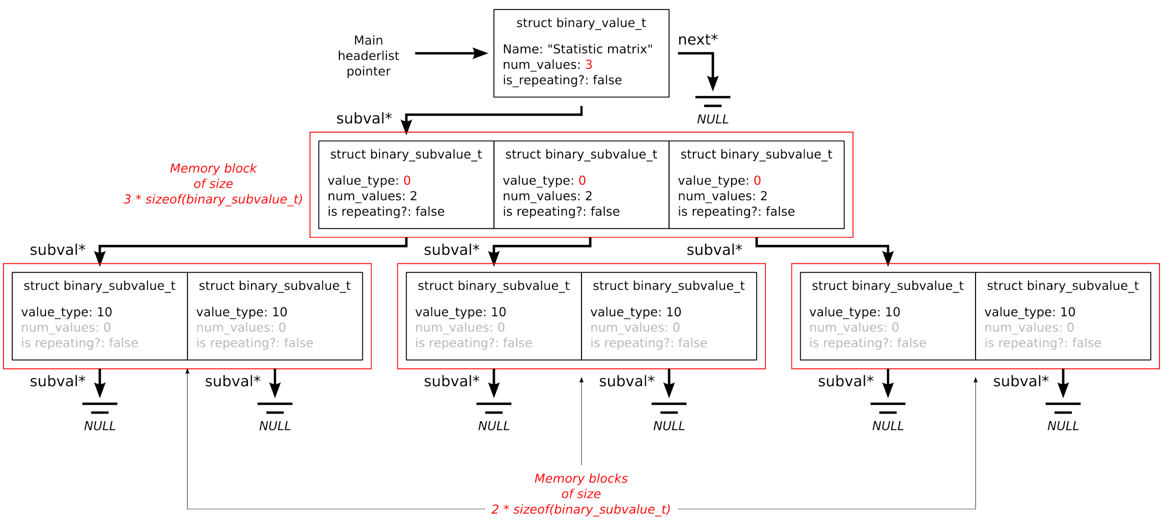


Example 2: A statistic composed of two values Now the output of the plugin is again two statistics, but the first statistic consists of two values; e.g. to describe a position on a grid. Therefore `num_values` is two and `subval*` points to a memory field of size two-times struct `binary_subvalue_t`. The subvalues themselves contain again the type of the statistic's values. Note: These values do not need to be identical.

Copyright © 2008–2024 by Tranalyzer Development Team



Example 3: A statistic containing a complete matrix With the ability to define subvalues in subvalues it is possible to store multidimensional structures such as matrices. The following example illustrates the definition of a matrix of size three times two:



1.11.2 Helper functions

In order to avoid filling the structures by hand a small API is located in the header file *binaryValue.h* doing all the nitty-gritty work for the programmer. The most important functions are described below.

```
binary_value_t* bv_append_bv(binary_value_t* dest, binary_value_t* new)
```

Append a `binary_value_t` struct at the end of a list of `binary_value_t` structures.

Copyright © 2008–2024 by Tranalyzer Development Team

Return a pointer to the start of the list.

Arguments:

Type	Name	Description
`binary_value_t*`	`dest`	pointer to the start of the list
`binary_value_t*`	`new`	pointer to the new `binary_value_t` structure

```
binary_value_t* bv_new_bv (const char *name, const char *desc, uint32_t is_repeating,
                           uint32_t num_values...)
```

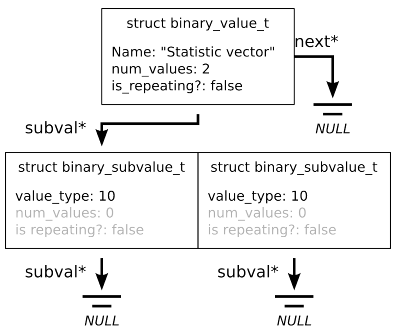
Generate a new structure of type `binary_value_t` and returns a pointer to it.

Arguments:

Type	Name	Description
`char*`	`name`	name for the statistic
`char*`	`desc`	description for the statistic
`uint32_t`	`is_repeating`	one, if the statistic is repeating, zero otherwise
`uint32_t`	`num_values`	number of values for the statistic
`int`	`...`	types of the statistical values, repeated `num_values`-times

The function creates a `binary_value_t` structure and sets the values. In addition, it creates an array field with `num_values` `binary_subvalue_t` structures and fills the value types provided in the variable argument list.

Example: The call `bv_new_bv("stat_vec", "Statistic vector", 2, 0, bt_uint_64, bt_uint_64)` creates the following structures:




```
binary_value_t* bv_add_sv_to_bv (binary_value_t* dest, uint32_t pos,
                                 uint32_t is_repeating, uint32_t num_values, ...)
```


Copyright © 2008–2024 by Tranalyzer Development Team

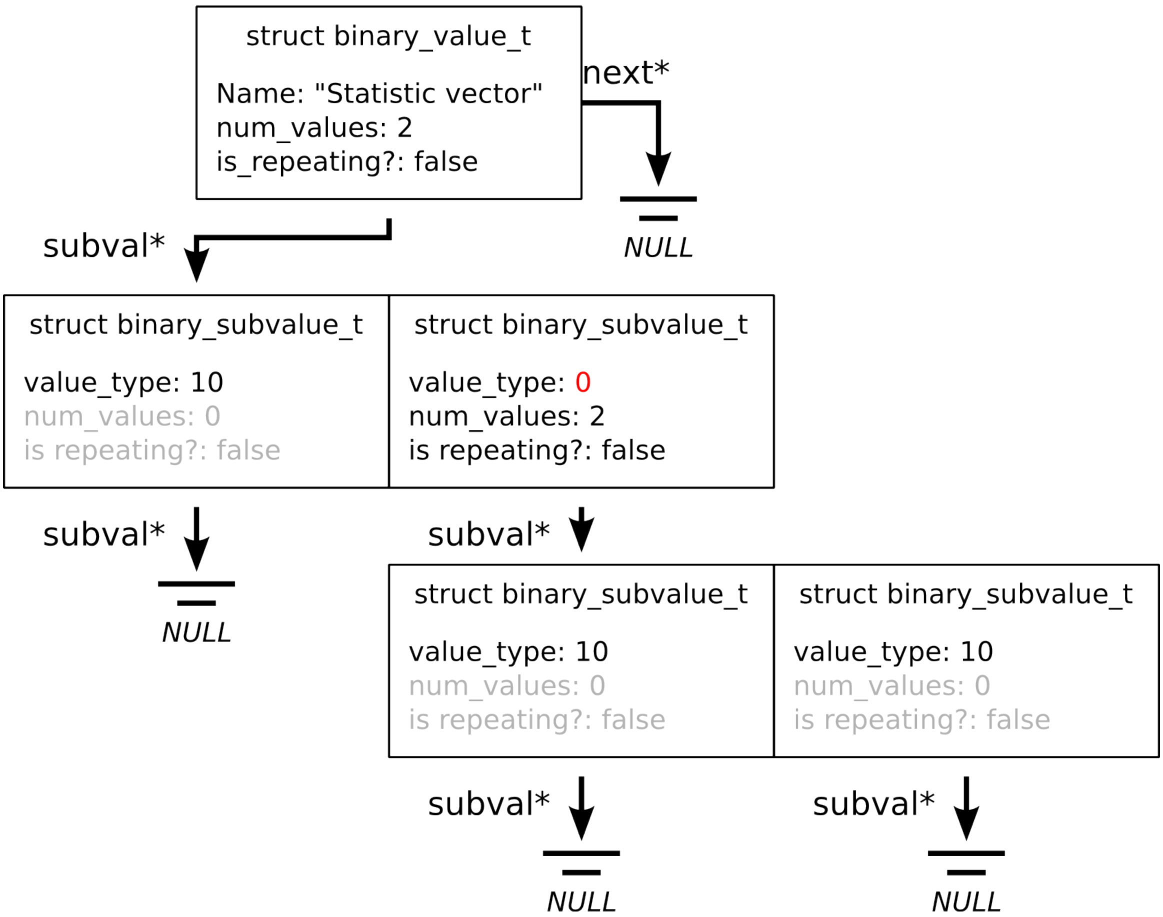
Replace a subvalue in a `binary_value_t` structure with a new substructure that contains additional substructures and returns a pointer to the parent binary value.

Arguments:

Type	Name	Description
`binary_value_t*`	`dest`	pointer to the parent binary value
`uint32_t`	`pos`	position of the substructure to be replaced, starting at 0
`uint32_t`	`is_repeating`	one, if the subvalue is repeating, zero otherwise
`uint32_t`	`num_values`	number of values in the subvalue
`int`	`...`	types of the statistical values, repeated `num_values`-times

This function is only valid if `dest` is already a complete statistic containing all necessary structures.

Example: Let *dest* be a pointer to the `binary_value_t` structure from the example above. A call to the function `bv_add_sv_to_bv(dest, 1, 0, 2, bt_uint_64, bt_uint_64)` replaces the second substructure with a new substructure containing two more substructures:



Copyright © 2008–2024 by Tranalyzer Development Team

```
binary_value_t* bv_add_sv_to_sv (binary_subvalue_t* dest, uint32_t pos,
                                 uint32_t is_repeating, uint32_t num_values, ...)
```

Replace a subvalue in a `binary_subvalue_t` structure with a new substructure that contains additional substructures and returns a pointer to the parent binary subvalue.

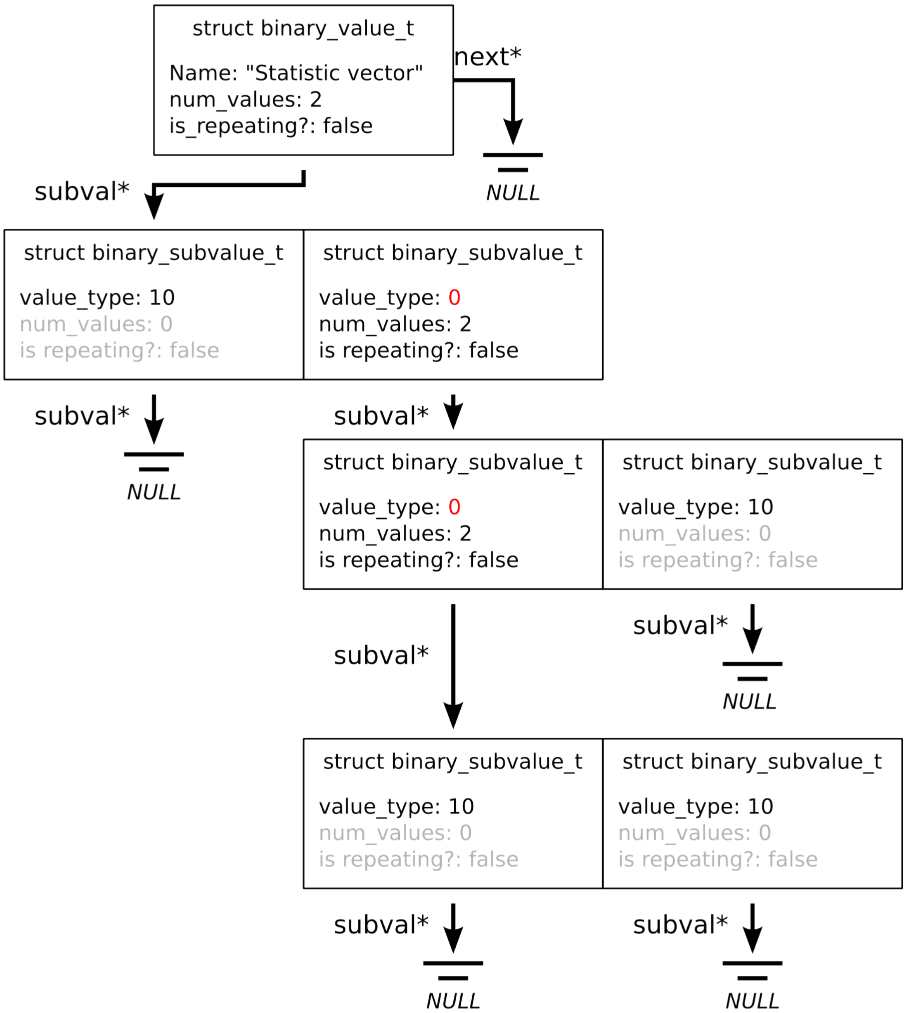
Arguments:

Type	Name	Description
`binary_subvalue_t*`	`dest`	Pointer to the parent binary subvalue
`uint32_t`	`pos`	Position of the substructure to be replaced, starting at 0
`uint32_t`	`is_repeating`	one, if the subvalue is repeating, zero otherwise
`uint32_t`	`num_values`	Number of values in the subvalue
`int`	`...`	Types of the statistical values, repeated `num_values`-times

For all hierarchical deeper located structures than above the function described above is required.

Example: Let *dest* be a pointer to the subvalue structure being replaced in the example above. A call to the function `bv_add_sv_to_sv(dest, 0, 0, 2, bt_uint_64, bt_uint_64)` replaces *dest's* first the substructure with a new substructure containing two more substructures:

Copyright © 2008–2024 by Tranalyzer Development Team



1.11.3 Writing into the standard output

Standard output is generated using a buffer structure. Upon the event `t2OnFlowTerminate` (see 1.15.6) Plugins write all output into this buffer. It is strongly recommended using the function `outputBuffer_append(outputBuffer_t* buffer, char* output, size_t size_of_output)`.
Arguments:

Type	Name	Description
`outputBuffer_t*`	`buffer`	pointer to the standard output buffer structure, for standard output, this is `main_output_buffer`
`char*`	`output`	pointer to the output, currently of type `char`
`size_t`	`size_of_output`	the length of field `output` in single bytes

The output buffer is send to the *output sinks* after all plugins have stored their information.

Copyright © 2008–2024 by Tranalyzer Development Team

Example: If a plugin wants to write two statistics each with a single value of type `uint64_t` it first has to commit its `binary_value_t` structure(s) (see section above). During the call of its `t2OnFlowTerminate()` function the plugin writes both statistical values using the append function:

```
outputBuffer_append(main_output_buffer, (char*) &value1, 4);
outputBuffer_append(main_output_buffer, (char*) &value2, 4);
```

Where `value1` and `value2` are two pointers to the statistical values.

1.12 Writing repeated output

If a statistic could be repeated (field `is_repeating` is one) the plugin has first to store the number of values as `uint32_t` value into the buffer. Afterwards, it appends the values.

Example: A plugin's output is a vector of variable length, the values are of type `uint16_t`. For the current flow, that is terminated in the function `t2OnFlowTerminate()`, there are three values to write. The plugin first writes a field of type `uint32_t` with value three into the buffer, using the append function:

```
outputbuffer_append(main_output_buffer, (char*) &numOfValues, sizeof(uint32_t));
```

Afterwards, it writes the tree values.

1.13 Important notes

- IP addresses (`bt_ip4_addr`, `bt_ip6_addr` and `bt_ipx_addr`) or MAC addresses (`bt_mac_addr`) are stored in network order.
- Strings are of variable length and need to be stored with a trailing zero byte (`'\0'`).

1.14 Administrative functions

Every plugin has to provide five administrative functions. The first four are mandatory while the last one is optional. For convenience, the following two macros can be used instead:

- `T2_PLUGIN_INIT(name, version, t2_v_major, t2_v_minor)`
- `T2_PLUGIN_INIT_WITH_DEPS(name, version, t2_v_major, t2_v_minor, deps)`

For example, to initialize `myPlugin`:

```
T2_PLUGIN_INIT_WITH_DEPS("myPlugin", "0.9.3", 0, 9, "tcpFlags,basicStats")
```

Function	Description
`const char *t2PluginName()`	Name of the plugin, e.g., `"myPlugin"`.
`const char *t2PluginVersion()`	Version of the plugin, e.g., `"0.9.0"`
`unsigned int t2SupportedT2Major()`	Minimum major version of the core supported by the plugin, e.g., 0
`unsigned int t2SupportedT2Minor()`	Minimum minor version of the core supported by the plugin, e.g., 9
`const char *t2Dependencies()`	Comma separated list of plugin names required by the plugin, e.g.,

Copyright © 2008–2024 by Tranalyzer Development Team

Function	Description
	"tcpFlags,tcpStates"

The existence of these functions is checked during the plugin initialization phase one and two, as highlighted in Figure 1.

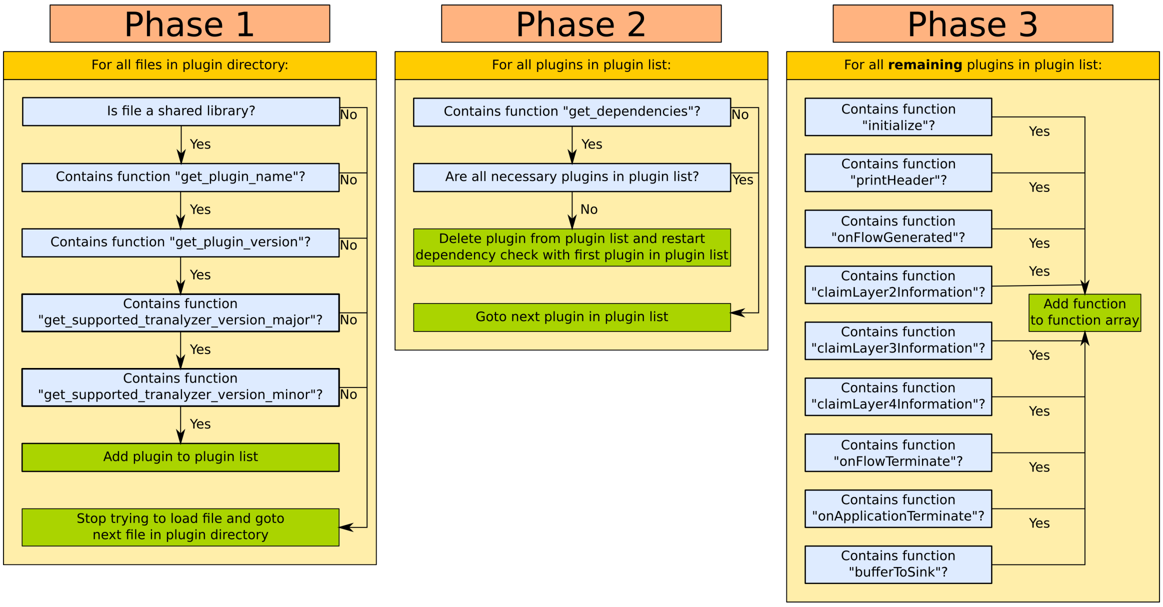


Figure 1: *Processing of the plugin loading mechanism*

1.15 Processing functions

During flow analysis Tranalyzer2 generates several *events* based on the status of the program, the inspected OSI layer of the current packet or the status of the current flow. These events consist of specific function calls provided by the plugins. The implementation of the event functions is dependent on the required action of a plugin to be carried out upon a certain event.

1.15.1 void t2Init()

The `t2Init` event is generated before the program activates the packet capturing phase. After Tranalyzer2 has initialized its internal structures it grants the same phase to the plugins. Therefore temporary values should be allocated during that event by using a C `malloc`.

1.15.2 binary_value_t *t2PrintHeader()

This event is also generated during the initialization phase. With this event the plugin providing data to the standard output file signals the core what type of output they want to write (see 1.8). The function returns a pointer to the generated `binary_value_t` structure or to the start pointer of a list of generated `binary_value_t` structures.

Copyright © 2008–2024 by Tranalyzer Development Team

1.15.3 void t2OnNewFlow(packet_t *packet, unsigned long flowIndex)

This event is generated every time Tranalyzer2 recognizes a new flow not present in the flow table. The first parameter is the currently processed packet, the second denotes the new generated flow index. As long as the flow is not terminated the flow index is valid. After flow termination the flow number is reintegrated into a list for later reuse.

1.15.4 void t2OnLayer2(packet_t *packet, unsigned long flowIndex)

This event is generated for every new packet comprising of a valid and supported layer two header, e.g. Ethernet as default. This is the first event generated after libpcap dispatches a packet and before a lookup in the flow table happened. At this very point in time no tests are conducted for higher layer headers. If a plugin tries to access higher layer structures it has to test itself if they are present or not. Otherwise, at non-presence of higher layers an unchecked access can result in a `NULL` pointer access and therefore in a possible segmentation fault! We recommend using the subsequent two events to access higher layers.

1.15.5 void t2OnLayer4(packet_t *packet, unsigned long flowIndex)

This event is generated for every new packet containing a valid and supported layer four header. The current supported layer four headers are TCP, UDP and ICMP. This event is called after Tranalyzer2 performs a lookup in its flow table and eventually generates an `t2OnNewFlow` event. Implementation of other protocols such as IPsec or OSPF are planned.

1.15.6 void t2OnFlowTerminate(unsigned long flowIndex, outputBuffer_t *buf)

This event is generated every time Tranalyzer2 removes a flow from its active status either due to timeout or protocol normal or abnormal termination. Only during this event, the plugins write output to the standard output.

1.15.7 void t2Finalize()

This event is generated shortly before the program is terminated. At this time no more packets or flows are processed. This event enables the plugins to do memory housekeeping, stream buffer flushing or printing of final statistics.

1.15.8 void t2BufferToSink(outputBuffer_t *buffer, binary_value_t *bv)

The Tranalyzer core generates this event immediately after the `t2OnFlowTerminate` event with the main output buffer as parameter. A plugin listening to this event is able to write this buffer to a data sink. For example the `binSink` plugin pushes the output into the `PREFIX_flows.bin` file.

1.16 Timeout handlers

A flow is terminated after a certain timeout being defined by so called *timeout handlers*. The default timeout value for a flow is 182 seconds. The plugins are able to access and change this value. For example, the `tcpStates` plugin changes the value according to different connection states of a TCP flow.

1.16.1 Registering a new timeout handler

To register a new timeout handler, a plugin has to call the `timeout_handler_add(float timeout_in_sec)` function. The argument is the new timeout value in seconds. Now the plugin is authorized by the core to change the timeout of a flow to the registered timeout value. Without registering a timeout handler the test is unreliable.

Copyright © 2008–2024 by Tranalyzer Development Team

1.16.2 Programming convention and hints

- A call to `timeout_handler_add` should only happen during the initialization function of the plugin.
- Registering the same timeout value twice is no factor.
- Registering timeout values in fractions of seconds is possible, see `tcpStates` plugin.

Copyright © 2008–2024 by Tranalyzer Development Team

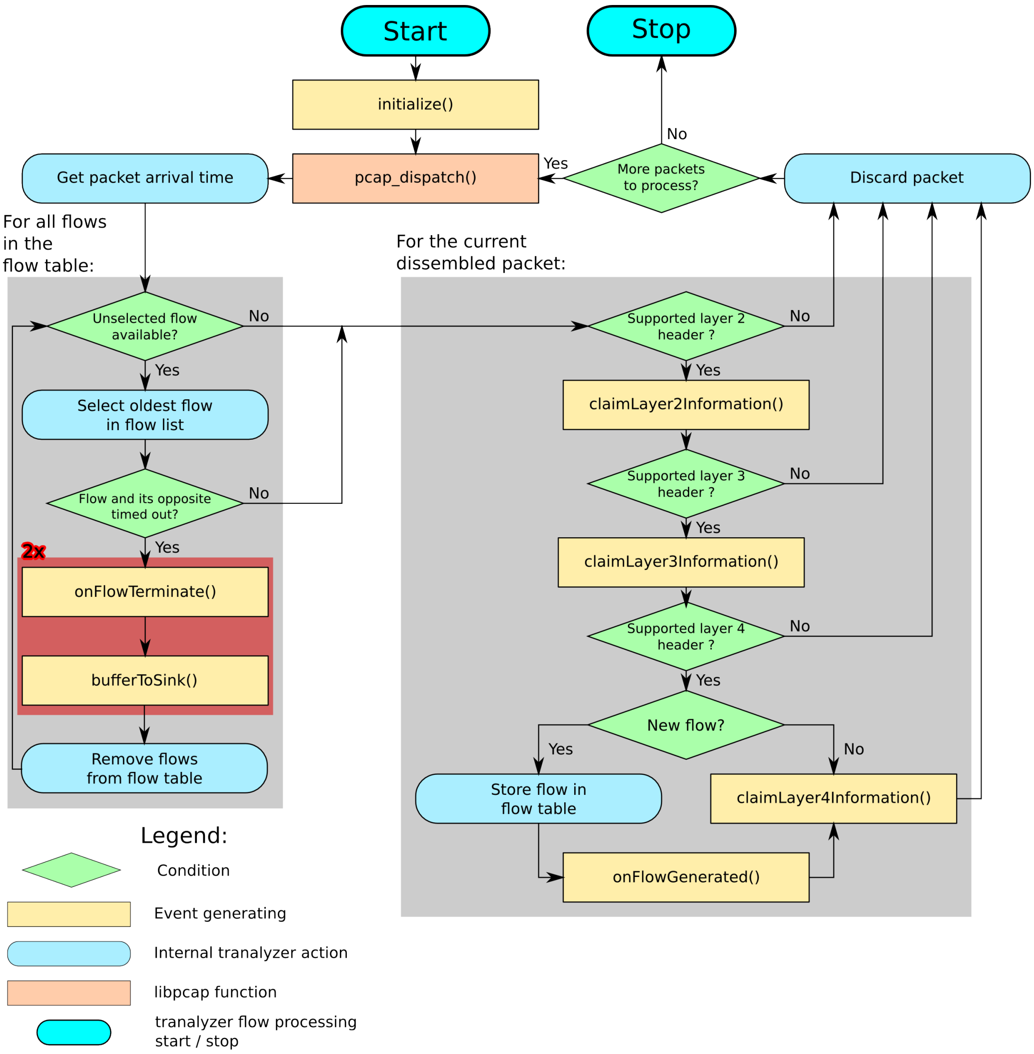


Figure 2: *Tranalyzer packet processing and event generation.*

Copyright © 2008–2024 by Tranalyzer Development Team