Suricata Developer Guide

Release 6.0.7

OISF

Sep 27, 2022

CONTENTS

CHAPTER

ONE

WORKING WITH THE CODEBASE

1.1 Contributing

1.1.1 Code Submission Process

Commits

1. Commits need to be logically separated. Don't fix unrelated things in one commit.
2. Don't add unnecessary commits, if commit 2 fixes commit 1 merge them together (squash)
3. Commits need to have proper messages, explaining anything that is non-trivial
4. Commits should not at the same time change, rename and/or move code. Use separate commits for each of this, e.g, a commit to rename files, then a commit to change the code.
5. Documentation updates should be in their own commit (not mixed with code commits)
6. **Commit messages need to be properly formatted:**
    - Meaningful and short (50 chars max) subject line followed by an empty line
    - Naming convention: prefix message with sub-system ("rule parsing: fixing foobar"). If you're not sure what to use, look at past commits to the file(s) in your PR.
    - Description, wrapped at ~72 characters
7. Commits should be individually compilable, starting with the oldest commit. Make sure that each commit can be built if it and the preceding commits in the PR are used.

Information that needs to be part of a commit (if applicable):

1. Ticket it fixes. E.g. "Fixes Bug #123."
2. Compiler warnings addressed.
3. Coverity Scan issues addressed.
4. Static analyzer error it fixes (cppcheck/scan-build/etc)

Pull Requests

A github pull request is actually just a pointer to a branch in your tree. Github provides a review interface that we use.

1. A branch can only be used in for an individual PR.
2. A branch should not be updated after the pull request

3. A pull request always needs a good description (link to issue tracker if related to a ticket).
4. Incremental pull requests need to link to the prior iteration
5. Incremental pull requests need to describe changes since the last PR
6. Link to the ticket(s) that are addressed to it.
7. When fixing an issue, update the issue status to `In Review` after submitting the PR.
8. Pull requests are automatically tested using github actions (https://github.com/OISF/suricata/blob/master/.github/workflows/builds.yml). Failing builds won't be considered and should be closed immediately.
9. Pull requests that change, or add a feature should include a documentation update commit

Tests and QA

As much as possible, new functionality should be easy to QA.

1. Add `suricata-verify` tests for verification. See https://github.com/OISF/suricata-verify
2. Add unittests if a `suricata-verify` test isn't possible.
3. Provide pcaps that reproduce the problem. Try to trim as much as possible to the pcap includes the minimal set of packets that demonstrate the problem.
4. Provide example rules if the code added new keywords or new options to existing keywords

1.2 Coding Style

Suricata uses a fairly strict coding style. This document describes it.

1.2.1 Formatting

clang-format

`clang-format` is configured to help you with formatting C code.

> **Note**
> The `.clang-format` file requires clang 9 or newer.

Format your Changes

Before opening a pull request, please also try to ensure it is formatted properly. We use `clang-format` for this, which has git integration through the `git-clang-format` script to only format your changes. On some systems, it may already be installed (or be installable via your package manager). If so, you can simply run it.

It is recommended to format each commit as you go. However, you can always reformat your whole branch after the fact.

Note
Depending on your installation, you might have to use the version-specific `git clang-format` in the commands below, e.g. `git clang-format-9`, and possibly even provide the `clang-format` binary with `--binary clang-format-9`.
As an alternative, you can use the provided `scripts/clang-format.sh` that isolates you from the different versions.

Formatting the most recent commit only

The following command will format only the code changed in the most recent commit:

```
$ git clang-format HEAD^
# Or with script:
$ scripts/clang-format.sh commit
```

Note that this modifies the files, but doesn't commit them – you'll likely want to run

```
$ git commit --amend -a
```

in order to update the last commit with all pending changes.

Formatting code in staging

The following command will format the changes in staging, i.e. files you `git add`-ed:

```
$ git clang-format
# Or with script:
$ scripts/clang-format.sh cached
```

If you also want to change the unstaged changes, do:

```
$ git clang-format --force
# Or with script:
$ scripts/clang-format.sh cached --force
```

Formatting your branch' commits

In case you have multiple commits on your branch already and forgot to format them you can fix that up as well.

The following command will format every commit in your branch off master and rewrite history using the existing commit metadata.

Tip: Create a new version of your branch first and run this off the new version.

```
# In a new version of your pull request:
$ scripts/clang-format.sh rewrite-branch
```

You could also add the formatting as an additional commit "at the end". However, this is frowned upon. It's preferred to use `rewrite-branch` instead.

```
# It's preferred to use rewrite-branch instead of this:
$ git clang-format first_commit_on_your_branch^
# Or with script:
$ scripts/clang-format.sh branch
```

Note the usage of `first_commit_on_your_branch^`, not `master`, to avoid picking up new commits on master in case you've updated master since you've branched.

Check formatting

Check if your branch changes' formatting is correct with:

```
$ scripts/clang-format.sh check-branch
```

Add the `--diffstat` parameter if you want to see the files needing formatting. Add the `--diff` parameter if you want to see the actual diff of the formatting change.

Formatting a whole file

> **Note**
> Do not reformat whole files by default, i.e. do not use `clang-format` proper in general.

If you were ever to do so, formatting changes of existing code with clang-format shall be a different commit and must not be mixed with actual code changes.

```
$ clang-format -i {file}
```

Disabling clang-format

There might be times, where the clang-format's formatting might not please. This might mostly happen with macros, arrays (single or multi-dimensional ones), struct initialization, or where one manually formatted code.

You can always disable clang-format.

```
/* clang-format off */
#define APP_LAYER_INCOMPLETE(c, n) (AppLayerResult){1, (c), (n)}
/* clang-format on */
```

Installing clang-format and git-clang-format

clang-format 9 or newer is required.

On ubuntu 18.04:

- It is sufficient to only install clang-format, e.g.

  ```
  $ sudo apt-get install clang-format-9
  ```

- See http://apt.llvm.org for other releases in case the clang-format version is not found in the default repos.

On fedora:

- Install the `clang` and `git-clang-format` packages with

```
$ sudo dnf install clang git-clang-format
```

Line length

Limit line lengths to 100 characters.

When wrapping lines that are too long, they should be indented at least 8 spaces from previous line. You should attempt to wrap the minimal portion of the line to meet the 100 character limit.

clang-format:

- ColumnLimit: 100
- ContinuationIndentWidth: 8
- ReflowComments: true

Indent

We use 4 space indentation.

```
int DecodeEthernet(ThreadVars *tv, DecodeThreadVars *dtv, Packet *p,
        uint8_t *pkt, uint16_t len, PacketQueue *pq)
{
    SCPerfCounterIncr(dtv->counter_eth, tv->sc_perf_pca);

    if (unlikely(len < ETHERNET_HEADER_LEN)) {
        ENGINE_SET_INVALID_EVENT(p, ETHERNET_PKT_TOO_SMALL);
        return TM_ECODE_FAILED;
    }

    ...

    DecodeNetworkLayer(tv, dtv, SCNtohs(p->ethh->eth_type), p,
            pkt + ETHERNET_HEADER_LEN, len - ETHERNET_HEADER_LEN);

    return TM_ECODE_OK;
}
```

Use 8 space indentation when wrapping function parameters, loops and if statements.

Use 4 space indentation when wrapping variable definitions.

```
const SCPlugin PluginSpec = {
    .name = OUTPUT_NAME,
    .author = "Some Developer",
    .license = "GPLv2",
    .Init = TemplateInit,
};
```

clang-format:

- AlignAfterOpenBracket: DontAlign
- Cpp11BracedListStyle: false
- IndentWidth: 4

- TabWidth: 8 [llvm]
- UseTab: Never [llvm]

Braces

Functions should have the opening brace on a newline:

```
int SomeFunction(void)
{
    DoSomething();
}
```

Note: this is a fairly new requirement, so you'll encounter a lot of non-compliant code.

Control and loop statements should have the opening brace on the same line:

```
if (unlikely(len < ETHERNET_HEADER_LEN)) {
    ENGINE_SET_INVALID_EVENT(p, ETHERNET_PKT_TOO_SMALL);
    return TM_ECODE_FAILED;
}

for (ascii_code = 0; ascii_code < 256; ascii_code++) {
    ctx->goto_table[ctx->state_count][ascii_code] = SC_AC_FAIL;
}

while (funcs != NULL) {
    temp = funcs;
    funcs = funcs->next;
    SCFree(temp);
}
```

Opening and closing braces go on the same line as as the _else_ (also known as a "cuddled else").

```
if (this) {
    DoThis();
} else {
    DoThat();
}
```

Structs, unions and enums should have the opening brace on the same line:

```
union {
    TCPVars tcpvars;
    ICMPV4Vars icmpv4vars;
    ICMPV6Vars icmpv6vars;
} l4vars;

struct {
    uint8_t type;
    uint8_t code;
} icmp_s;

enum {
    DETECT_TAG_TYPE_SESSION,
    DETECT_TAG_TYPE_HOST,
    DETECT_TAG_TYPE_MAX
};
```

clang-format:

- BreakBeforeBraces: Custom [breakbeforebraces]
- BraceWrapping:
  - AfterClass: true
  - AfterControlStatement: false
  - AfterEnum: false
  - AfterFunction: true
  - AfterStruct: false
  - AfterUnion: false
  - AfterExternBlock: true
  - BeforeElse: false
  - IndentBraces: false

1.2.2 Flow

Don't use conditions and statements on the same line. E.g.

```
if (a) b = a; // <- wrong

if (a)
    b = a; // <- right

for (int i = 0; i < 32; ++i) f(i); // <- wrong

for (int i = 0; i < 32; ++i)
    f(i); // <- right
```

Don't put short or empty functions and structs on one line.

```
void empty_function(void)
{
}

int short_function(void)
{
    return 1;
}
```

Don't use unnecessary branching. E.g.:

```
if (error) {
    goto error;
} else {
    a = b;
}
```

Can be written as:

```
if (error) {
    goto error;
}
a = b;
```

clang-format:

- AllowShortBlocksOnASingleLine: false [llvm]
- AllowShortBlocksOnASingleLine: Never [llvm] (breaking change in clang 10!) [clang10]
- AllowShortEnumsOnASingleLine: false [clang11]
- AllowShortFunctionsOnASingleLine: None
- AllowShortIfStatementsOnASingleLine: Never [llvm]
- AllowShortLoopsOnASingleLine: false [llvm]
- BreakBeforeBraces: Custom [breakbeforebraces]
- BraceWrapping:
  - SplitEmptyFunction: true
  - SplitEmptyRecord: true

1.2.3 Alignment

Pointers

Pointers shall be right aligned.

```
void *ptr;
void f(int *a, const char *b);
void (*foo)(int *);
```

clang-format:

- PointerAlignment: Right
- DerivePointerAlignment: false

Declarations and Comments

Trailing comments should be aligned for consecutive lines.

```
struct bla {
    int a;        /* comment */
    unsigned bb; /* comment */
    int *ccc;     /* comment */
};

void alignment()
{
    // multiple consecutive vars
    int a = 13;           /* comment */
    int32_t abc = 1312;   /* comment */
```

(continues on next page)

(continued from previous page)

```
    int abcdefghikl = 13; /* comment */
}
```

clang-format:

- AlignConsecutiveAssignments: false
- AlignConsecutiveDeclarations: false
- AlignTrailingComments: true

1.2.4 Functions

parameter names

TODO

Function names

Function names are NamedLikeThis().

```
static ConfNode *ConfGetNodeOrCreate(char *name, int final)
```

static vs non-static

Functions should be declared static whenever possible.

inline

The inlining of functions should be used only in critical paths.

1.2.5 Variables

Names

A variable is `named_like_this` in all lowercase.

```
ConfNode *parent_node = root;
```

Generally, use descriptive variable names.

In loop vars, make sure `i` is a signed int type.

Scope

TODO

1.2.6 Macros

Macro names are ALL_CAPS_WITH_UNDERSCORES. Enclose parameters in parens on each usage inside the macro.

Align macro values on consecutive lines.

```
#define ACTION_ALERT       0x01
#define ACTION_DROP        0x02
#define ACTION_REJECT      0x04
#define ACTION_REJECT_DST  0x08
#define ACTION_REJECT_BOTH 0x10
#define ACTION_PASS        0x20
```

Align escape for multi-line macros right-most at ColumnLimit.

```
#define MULTILINE_DEF(a, b)                                                    \
    if ((a) > 2) {                                                             \
        auto temp = (b) / 2;                                                   \
        (b) += 10;                                                             \
        someFunctionCall((a), (b));                                            \
    }
```

clang-format:

- AlignConsecutiveMacros: true [clang9]
- AlignEscapedNewlines: Right

1.2.7 Comments

TODO

Function comments

We use Doxygen, functions are documented using Doxygen notation:

```
/**
 * \brief Helper function to get a node, creating it if it does not
 * exist.
 *
 * This function exits on memory failure as creating configuration
 * nodes is usually part of application initialization.
 *
 * \param name The name of the configuration node to get.
 * \param final Flag to set created nodes as final or not.
 *
 * \retval The existing configuration node if it exists, or a newly
 * created node for the provided name. On error, NULL will be returned.
 */
static ConfNode *ConfGetNodeOrCreate(char *name, int final)
```

General comments

We use `/* foobar */` style and try to avoid `//` style.

1.2.8 File names

File names are all lowercase and have a .c. .h or .rs (Rust) extension.

Most files have a _subsystem_ prefix, e.g. `detect-dsize.c, util-ip.c`

Some cases have a multi-layer prefix, e.g. `util-mpm-ac.c`

1.2.9 Enums

Use a common prefix for all enum values. Value names are ALL_CAPS_WITH_UNDERSCORES.

Put each enum values on a separate line. Tip: Add a trailing comma to the last element to force "one-value-per-line" formatting in clang-format.

```
enum { VALUE_ONE, VALUE_TWO };  // <- wrong

// right
enum {
    VALUE_ONE,
    VALUE_TWO, // <- force one-value-per-line
};
```

clang-format:

- AllowShortEnumsOnASingleLine: false [clang11]

1.2.10 Structures and typedefs

TODO

1.2.11 switch statements

Switch statements are indented like in the following example, so the 'case' is indented from the switch:

```
switch (ntohs(p->ethh->eth_type)) {
    case ETHERNET_TYPE_IP:
        DecodeIPV4(tv, dtv, p, pkt + ETHERNET_HEADER_LEN,
                   len - ETHERNET_HEADER_LEN, pq);
        break;
```

Fall through cases will be commented with `/* fall through */`. E.g.:

```
switch (suri->run_mode) {
    case RUNMODE_PCAP_DEV:
    case RUNMODE_AFP_DEV:
    case RUNMODE_PFRING:
        /* find payload for interface and use it */
        default_packet_size = GetIfaceMaxPacketSize(suri->pcap_dev);
        if (default_packet_size)
            break;
        /* fall through */
    default:
        default_packet_size = DEFAULT_PACKET_SIZE;
```

Do not put short case labels on one line. Put opening brace on same line as case statement.

```
switch (a) {
    case 13: {
        int a = bla();
        break;
    }
    case 15:
        blu();
        break;
    default:
        gugus();
}
```

clang-format:

- IndentCaseLabels: true
- IndentCaseBlocks: false [clang11]
- AllowShortCaseLabelsOnASingleLine: false [llvm]
- BreakBeforeBraces: Custom [breakbeforebraces]
- BraceWrapping:
  - AfterCaseLabel: false (default)

1.2.12 const

TODO

1.2.13 goto

Goto statements should be used with care. Generally, we use it primarily for error handling. E.g.:

```
static DetectFileextData *DetectFileextParse (char *str)
{
    DetectFileextData *fileext = NULL;

    fileext = SCMalloc(sizeof(DetectFileextData));
    if (unlikely(fileext == NULL))
        goto error;

    memset(fileext, 0x00, sizeof(DetectFileextData));

    if (DetectContentDataParse("fileext", str, &fileext->ext, &fileext->len, &fileext-
↪>flags) == -1) {
        goto error;
    }

    return fileext;

error:
    if (fileext != NULL)
        DetectFileextFree(fileext);
    return NULL;
}
```

Put goto labels at brace level.

```
int goto_style_nested()
{
    if (foo()) {
    label1:
        bar();
    }

label2:
    return 1;
}
```

clang-format:

- IndentGotoLabels: true (default) [clang10]

1.2.14 Includes

TODO

A .c file shall include it's own header first.

clang-format:

- SortIncludes: false

1.2.15 Unittests

When writing unittests that use a data array containing a protocol message, please put an explanatory comment that contain the readable content of the message

So instead of:

```
int SMTPProcessDataChunkTest02(void)
{
    char mimemsg[] = {0x4D, 0x49, 0x4D, 0x45, 0x2D, 0x56, 0x65, 0x72,
```

you should have something like:

```
int SMTPParserTest14(void)
{
    /* 220 mx.google.com ESMTP d15sm986283wfl.6<CR><LF> */
    static uint8_t welcome_reply[] = { 0x32, 0x32, 0x30, 0x20,
```

1.2.16 Banned functions

function	replacement	reason
strok	strtok_r	
sprintf	snprintf	unsafe
strcat	strlcat	unsafe
strcpy	strlcpy	unsafe
strncpy	strlcat	
strncat	strlcpy	
strndup		OS specific
strchrnul		
rand		
rand_r		
index		
rindex		
bzero	memset	

Also, check the existing code. If yours is wildly different, it's wrong. Example: https://github.com/oisf/suricata/blob/master/src/decode-ethernet.c

1.3 Unittests

Unittests are a great way to create tests that can check the internal state of parsers, structures and other objects.

Tests should:

- use FAIL/PASS macros
- be deterministic
- not leak memory on PASS
- not use conditions

1.4 Fuzz Testing

To enable fuzz targets compilation, add *–enable-fuzztargets* to configure.

Note: This changes various parts of Suricata making the *suricata* binary unsafe for production use.

The targets can be used with libFuzzer, AFL and other fuzz platforms.

1.4.1 Running the Fuzzers

TODO. For now see src/tests/fuzz/README

Reproducing issues

1.4.2 Extending Coverage

1.4.3 Adding Fuzz Targets

1.4.4 Oss-Fuzz

Suricata is continuesly fuzz tested in Oss-Fuzz. See https://github.com/google/oss-fuzz/tree/master/projects/suricata

CHAPTER

TWO

SURICATA INTERNALS

2.1 Packet Pipeline

2.2 Threading

2.3 Important Data Structures

2.3.1 Introduction

This section explains the most important Suricata Data structures.

For a complete overview, see the doxygen: https://doxygen.openinfosecfoundation.org

2.4 Engines

2.4.1 Flow

2.4.2 Stream

2.4.3 Defrag

CHAPTER

THREE

EXTENDING SURICATA

3.1 Packet Capture

3.2 Packet Decoder

3.3 App-Layer

3.3.1 Parsers

Callbacks

The API calls callbacks that are registered at the start of the program.

The function prototype is:

```
typedef AppLayerResult (*AppLayerParserFPtr)(Flow *f, void *protocol_state,
        AppLayerParserState *pstate,
        const uint8_t *buf, uint32_t buf_len,
        void *local_storage, const uint8_t flags);
```

Examples

A C example:

```
static AppLayerResult HTPHandleRequestData(Flow *f, void *htp_state,
        AppLayerParserState *pstate,
        const uint8_t *input, uint32_t input_len,
        void *local_data, const uint8_t flags);
```

In Rust, the callbacks are similar.

```
#[no_mangle]
pub extern "C" fn rs_dns_parse_response_tcp(_flow: *const core::Flow,
        state: *mut std::os::raw::c_void,
        _pstate: *mut std::os::raw::c_void,
        input: *const u8,
        input_len: u32,
        _data: *const std::os::raw::c_void,
        _flags: u8)
-> AppLayerResult
```

Return Types

Parsers return the type *AppLayerResult*.

There are 3 possible results:

- *APP_LAYER_OK* - parser consumed the data successfully
- *APP_LAYER_ERROR* - parser encountered a unrecoverable error
- *APP_LAYER_INCOMPLETE(c,n)* - parser consumed *c* bytes, and needs *n* more before being called again

Rust parsers follow the same logic, but can return

- *AppLayerResult::ok()*
- *AppLayerResult::err()*
- *AppLayerResult::incomplete(c,n)*

For *i32* and *bool*, Rust parsers can also use *.into()*.

APP_LAYER_OK / AppLayerResult::ok()

When a parser returns "OK", it signals to the API that all data has been consumed. The parser will be called again when more data is available.

APP_LAYER_ERROR / AppLayerResult::err()

Returning "ERROR" from the parser indicates to the API that the parser encountered an unrecoverable error and the processing of the protocol should stop for the rest of this flow.

Note: This should not be used for recoverable errors. For those events should be set.

APP_LAYER_INCOMPLETE / AppLayerResult::incomplete()

Using "INCOMPLETE" a parser can indicate how much more data is needed. Many protocols use records that have the size as one of the first parameters. When the parser receives a partial record, it can read this value and then tell the API to only call the parser again when enough data is available.

consumed is used how much of the current data has been processed *needed* is the number of bytes that the parser needs on top of what was consumed.

Example:

```
[ 32 record 1 ][ 32 record 2 ][ 32 r.. ]
 0           31  32          63  64    72
                               ^   ^
consumed: 64 ---------------/    |
needed:   32 --------------------/
```

Note: "INCOMPLETE" is only supported for TCP

The parser will be called again when the *needed* data is available OR when the stream ends. In the latter case the data will be incomplete. It's up to the parser to decide what to do with it in this case.

Supporting incomplete data

In some cases it may be preferable to actually support dealing with incomplete records. For example protocols like SMB and NFS can use very large records during file transfers. Completely queuing these before processing could be a waste of resources. In such cases the "INCOMPLETE" logic could be used for just the record header, while the record data is streamed into the parser.

3.4 Detection

3.5 Output

3.5.1 Introduction

Extending Suricata's alert and event output.

BIBLIOGRAPHY

[llvm] Default LLVM clang-format Style

[clang9] Requires clang 9

[clang10] Requires clang 10

[clang11] Requires clang 11

[breakbeforebraces] BreakBeforeBraces: Mozilla is closest, but does not split empty functions/structs