Suricata - Documentation #4662
Add documentation section covering Suricata rule grammar
09/03/2021 02:20 PM - Juliana Fajardini Reichow

Description
As asked in our forum (https://forum.suricata.io/t/rule-grammar-specification/1664). Suggestion to add EBNF or PEG notation.

... If you want to work on this ticket, please ask to claim it in the comments.

Related issues:
Related to Documentation #4706: Guide for rulewriting

History
#1 - 09/24/2021 08:37 AM - Juliana Fajardini Reichow
- Related to Documentation #4706: Guide for rulewriting added

#2 - 02/22/2022 12:59 PM - Kirill Krotov
Hello, I want to discuss defining grammar for Suricata rules.
Currently I'm writing a grammar for Suricata rules by reversing its parser. Unfortunately Suricata rules parser is very loose about syntax of rules I will provide example to show what I mean.

You first expectation about rule's syntax is probably can be summarized in following (incomplete and maybe wrong I just want to concentrate on what parser does):

File = { (Rule, WS, Newline) | (Rule, WS, Comment) | Comment } ;
Comment = NewLine | ( '#' , { ASCII - NewLine }, NewLine ) ;
Rule = Action, WS, Header, WS, Options ;
Action = 'alert' | 'drop' | ... ;
Header = Protocol, WS, IPList, WS, PortList, WS, Direction, WS, IPList, WS, PortList ;
Protocol = 'tcp' | 'udp' | ... ;
Options = '(', { Option }, ')' ;
Option = OptionName | OptionName, ':', OptionValue, ';' ;
WS = { ' ' | '	' } ;
Newline = '
' | EOF ;

So something like this: alert tcp any any -> any any (sid: 123;) is totally correct rule, but let's discus it in more details

Comments
Actually comment can appear only at the beginning of line and line entirely considered to be commented if first symbol is: '!','',';','\n'.

So the next rule is just incorrect:

alert tcp any any -> any any (sid:123;) # comment

Next will not emit any error, but will not be parsed

alert tcp any any -> any any (sid:123;)

07/16/2022
Line breaks

Suricata rules has useful feature: it is able to continue rule on next line.

I think it's intended to work like this:

```plaintext
alert tcp any any -> any any (\n  sid: 123; \n)
```

But it's actually too powerful. It is working only on rule lines by removing whitespaces (isspace) at the end of line to '\'. Next line is not altered in any way, so it is possible to break rule at any place!

```plaintext
a\n1\ne\nr\nt tcp any any -\n> any any (sid:123;)
```

Also there is related problem with whitespaces.

Whitespaces

Whitespaces is something like ' ', \t newline and also \v, \r. They are handled by parser very inconsistent.

For example the first symbol after Action can be: \t, \t, \r, but first symbol after Protocol is only (but after it can also be \t).

Some example with escaped characters:

```plaintext
# valid
alert tcp any any -> any any (sid:123;)

# valid
alert\ttcp any any -> any any (sid:123;)

# invalid
alert tcp\tan y any -> any any (sid:123;)

# invalid
alert tcp\t any any -> any any (sid:123;)

# valid
alert tcp \tan y any -> any any (sid:123;)
```

Also imagine you used line break after Protocol and started next line with tab:

```plaintext
alert tcp\n\tan y any -> any any (sid:123;)
```

This will be considered incorrect rule.

The more strange situation with white spaces before Options: first should be ' ' but then can be varieties of spaces: \w, \r, \t, \f.

Options

I will not elaborate how options are parsed, because there are many different options and each can have some not obvious behavior, I just point that round brackets around options is totally optional:

```plaintext
# valid rule
alert tcp any any -> any any sid:123;
```

IP/Port lists

There a different elements that can be discussed separately.

Negation

Negation can be placed anywhere and can be duplicated. Also it can be placed after comma and the end of list. Valid rule that feature negation peculiarities:

```plaintext
# valid
alert tcp any !!10!! -> any any (sid:123;)
```
# valid (negated variable "HELLO")
alert tcp any $HELLO!O --> any any (sid:123;)

List brackets

Square brackets can be used to hide subtle bugs:

# valid (10 and 11 accepted correctly)
alert tcp any [10]11 --> any any (sid:123;)

# invalid (because how end of 11 if found)

# valid (11 ignored)
alert tcp any 11[10] --> any any (sid:123;)

# valid
alert tcp any 11[10]! --> any any (sid:123;)

# valid (try to guess what does it mean)
alert tcp any 11![999] --> any any (sid:123;)

Comma (separator)

Minor issue with comma is that can be leaved at the end of sequence. It's not a big deal, but can possibly be combined with other quirks of parser.
Examples:

# valid
alert tcp any 10, --> any any (sid:123;)

# valid
alert tcp any 10,! --> any any (sid:123;)

# valid
alert tcp any 10,![] --> any any (sid:123;)

Acceptable character ranges

There is possibility to abuse list parsing to allow variable include characters that are not allowed in non list context.

# invalid
alert tcp any $HELLO WORLD --> any any (sid:123;)

# valid (variable "HELLO WORLD" recognized)
alert tcp any [$HELLO WORLD] --> any any (sid:123;)

Conclusion

There are many weird way to write rules. But I think that we can ask three questions:

1. Can we write formal grammar for current rules?
2. Is there something wrong with current rule format?
3. What about backward compatibility if something will be changed (format will be restricted)?

I think that current loose parser implementation is not something very desirable, because user can not be sure what they wrote (see examples with list brackets as separators) and they can write rules that very difficult to read.

Also writing grammar for current parser is not easy task, it's certainly should be possible (except line breaks, they can be implemented as part of tokenizer, but should be more restricted to be incorporated in grammar), but grammar will be bloated with special cases and not very useful in my opinion.

But if Suricata rules will be changed, that can break old user rules. It's certainly possible, but this can be alleviated. For example we can write tool that rewrite rules in more strict format.