03-02-2022-24064301-HauptPost-0631

3.4.8 metadata

The `metadata` tag allows a rule writer to embed additional information about the rule, typically in a key-value format. Certain metadata keys and values have meaning to Snort and are listed in Table 3.3. Keys other than those listed in the table are effectively ignored by Snort and can be free-form, with a key and a value. Multiple keys are separated by a comma, while keys and values are separated by a space.

Table 3.3: Snort Metadata Keys

Key	Description	Value Format
`engine`	Indicate a Shared Library Rule	"shared"
`soid`	Shared Library Rule Generator and SID	gid|sid
`service`	Target-Based Service Identifier	"http"

NOTE

The `service` Metadata Key is only meaningful when a Host Atttribute Table is provided. When the value exactly matches the service ID as specified in the table, the rule is applied to that packet, otherwise, the rule is not applied (even if the ports specified in the rule match). See Section 2.7 for details on the Host Attribute Table.

1644709564601258531 9

Format

The examples below show an stub rule from a shared library rule. The first uses multiple metadata keywords, the second a single metadata keyword, with keys separated by commas.

```
metadata:key1 value1;
metadata:key1 value1, key2 value2;
```

Examples

```
alert tcp any any -> any 80 (msg:"Shared Library Rule Example"; \
    metadata:engine shared; metadata:soid 3|12345;)

alert tcp any any -> any 80 (msg:"Shared Library Rule Example"; \
    metadata:engine shared, soid 3|12345;)

alert tcp any any -> any 80 (msg:"HTTP Service Rule Example"; \
    metadata:service http;)
```

3.4.9 General Rule Quick Reference

Table 3.4: General rule option keywords

Keyword	Description
`msg`	The msg keyword tells the logging and alerting engine the message to print with the packet dump or alert.
`reference`	The reference keyword allows rules to include references to external attack identification systems.
`gid`	The gid keyword (generator id) is used to identify what part of Snort generates the event when a particular rule fires.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 145

sid	The sid keyword is used to uniquely identify Snort rules.
rev	The rev keyword is used to uniquely identify revisions of Snort rules.
classtype	The classtype keyword is used to categorize a rule as detecting an attack that is part of a more general type of attack class.
priority	The priority keyword assigns a severity level to rules.
metadata	The metadata keyword allows a rule writer to embed additional information about the rule, typically in a key-value format.

3.5 Payload Detection Rule Options

3.5.1 content

The content keyword is one of the more important features of Snort. It allows the user to set rules that search for specific content in the packet payload and trigger response based on that data. Whenever a content option pattern match is performed, the Boyer-Moore pattern match function is called and the (rather computationally expensive) test is performed against the packet contents. If data exactly matching the argument data string is contained anywhere within the packet's payload, the test is successful and the remainder of the rule option tests are performed. Be aware that this test is case sensitive.



The option data for the content keyword is somewhat complex; it can contain mixed text and binary data. The binary data is generally enclosed within the pipe (|) character and represented as bytecode. Bytecode represents binary data as hexadecimal numbers and is a good shorthand method for describing complex binary data. The example below shows use of mixed text and binary data in a Snort rule.

Note that multiple content rules can be specified in one rule. This allows rules to be tailored for less false positives.

If the rule is preceded by a !, the alert will be triggered on packets that do not contain this content. This is useful when writing rules that want to alert on packets that do not match a certain pattern



Also note that the following characters must be escaped inside a content rule:
; \ "

Format

```
content:[!]"<content string>";
```

Examples

```
alert tcp any any -> any 139 (content:"|5c 00|P|00|I|00|P|00|E|00 5c|";)

alert tcp any any -> any 80 (content:!"GET";)
```



A ! modifier negates the results of the entire content search, modifiers included. For example, if using `content:!"A"; within:50;` and there are only 5 bytes of payload and there is no "A" in those 5 bytes, the result will return a match. If there must be 50 bytes for a valid match, use `isdataat` as a pre-cursor to the content.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 146

Changing content behavior

The `content` keyword has a number of modifier keywords. The modifier keywords change how the previously specified content works. These modifier keywords are:

Table 3.5: Content Modifiers

Modifier	Section
nocase	3.5.2
rawbytes	3.5.3
depth	3.5.4
offset	3.5.5
distance	3.5.6
within	3.5.7
http_client_body	3.5.8
http_cookie	3.5.9
http_raw_cookie	3.5.10
http_header	3.5.11
http_raw_header	3.5.12
http_method	3.5.13
http_uri	3.5.14
http_raw_uri	3.5.15
http_stat_code	3.5.16
http_stat_msg	3.5.17
fast_pattern	3.5.19

16444709564601258531 9

3.5.2 nocase

The nocase keyword allows the rule writer to specify that the Snort should look for the specific pattern, ignoring case. nocase modifies the previous `content` keyword in the rule.

Format

```
nocase;
```

Example

```
alert tcp any any -> any 21 (msg:"FTP ROOT"; content:"USER root"; nocase;)
```

3.5.3 rawbytes

The rawbytes keyword allows rules to look at the raw packet data, ignoring any decoding that was done by preprocessors. This acts as a modifier to the previous content 3.5.1 option.

format

```
rawbytes;
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 147

Example

This example tells the content pattern matcher to look at the raw traffic, instead of the decoded traffic provided by the Telnet decoder.

```
alert tcp any any -> any 21 (msg:"Telnet NOP"; content:"|FF F1|"; rawbytes;)
```

3.5.4 depth

The depth keyword allows the rule writer to specify how far into a packet Snort should search for the specified pattern. depth modifies the previous 'content' keyword in the rule.

A depth of 5 would tell Snort to only look for the specified pattern within the first 5 bytes of the payload.

As the depth keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `depth` is specified.

This keyword allows values greater than or equal to the pattern length being searched. The minimum allowed value is 1. The maximum allowed value for this keyword is 65535.

The value can also be set to a string value referencing a variable extracted by the `byte_extract` keyword in the same rule.

The offset and depth keywords may be used together. You can not use depth with itself, distance, or within (to modify the same content).

1644470956460125853 19

Format

```
depth:[<number>|<var_name>];
```

3.5.5 offset

The offset keyword allows the rule writer to specify where to start searching for a pattern within a packet. offset modifies the previous 'content' keyword in the rule.

An offset of 5 would tell Snort to start looking for the specified pattern after the first 5 bytes of the payload.

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `offset` is specified.

This keyword allows values from -65535 to 65535.

The value can also be set to a string value referencing a variable extracted by the `byte_extract` keyword in the same rule.

The offset and depth keywords may be used together. You can not use offset with itself, distance, or within (to modify the same content).

Format

```
offset:[<number>|<var_name>];
```

Example

The following example shows use of a combined content, offset, and depth search rule.

```
alert tcp any any -> any 80 (content:"cgi-bin/phf"; offset:4; depth:20;)
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 148

03-02-2022-24064301-Hauptpost-0633

3.5.6 distance

The distance keyword allows the rule writer to specify how far into a packet Snort should ignore before starting to search for the specified pattern relative to the end of the previous pattern match.

This can be thought of as exactly the same thing as offset (See Section 3.5.5), except it is relative to the end of the last pattern match instead of the beginning of the packet.

This keyword allows values from -65535 to 65535.

The distance and within keywords may be used together. You can not use distance with itself, offset, or depth (to modify the same content).

The value can also be set to a string value referencing a variable extracted by the `byte_extract` keyword in the same rule.

Format

```
distance:[<byte_count>|<var_name>];
```

Example

The rule below maps to a regular expression of /ABC.{1}DEF/.

```
alert tcp any any -> any any (content:"ABC"; content:"DEF"; distance:1;)
```

16444709564601258531 9

3.5.7 within

The within keyword is a content modifier that makes sure that at most N bytes are between pattern matches using the content keyword (See Section 3.5.1). It's designed to be used in conjunction with the distance (Section 3.5.6) rule option.

This keyword allows values greater than or equal to pattern length being searched. The maximum allowed value for this keyword is 65535.

The distance and within keywords may be used together. You can not use within with itself, offset, or depth (to modify the same content).

The value can also be set to a string value referencing a variable extracted by the `byte_extract` keyword in the same rule.

Format

```
within:[<byte_count>|<var_name>];
```

Examples

This rule constrains the search of EFG to not go past 10 bytes past the ABC match.

```
alert tcp any any -> any any (content:"ABC"; content:"EFG"; within:10;)
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 149

3.5.8 http_client_body

The http_client_body keyword is a content modifier that restricts the search to the body of an HTTP client request.

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before 'http_client_body' is specified.

The amount of data that is inspected with this option depends on the `post_depth` config option of HttpInspect. Pattern matches with this keyword wont work when `post_depth` is set to -1.

Format

```
http_client_body;
```

Examples

This rule constrains the search for the pattern "EFG" to the raw body of an HTTP client request.

```
alert tcp any any -> any 80 (content:"ABC"; content:"EFG"; http_client_body;)
```



⚠NOTE

The `http_client_body` modifier is not allowed to be used with the `rawbytes` modifier for the same content.

3.5.9 http_cookie

The http_cookie keyword is a content modifier that restricts the search to the extracted Cookie Header field of a HTTP client request or a HTTP server response (per the configuration of HttpInspect 2.2.6). The cookie buffer also includes the header name (`Cookie` for HTTP requests or `Set-Cookie` for HTTP responses).

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `http_cookie` is specified. This keyword is dependent on the `enable_cookie` config option. The Cookie Header field will be extracted only when this option is configured. If `enable_cookie` is not specified, the cookie still ends up in HTTP header. When `enable_cookie` is not specified, using `http_cookie` is the same as using `http_header`.

The extracted Cookie Header field may be NORMALIZED, per the configuration of HttpInspect (see 2.2.6).

Format

```
http_cookie;
```

Examples

This rule constrains the search for the pattern "EFG" to the extracted Cookie Header field of a HTTP client request.

```
alert tcp any any -> any 80 (content:"ABC"; content:"EFG"; http_cookie;)
```

⚠NOTE

The `http_cookie` modifier is not allowed to be used with the `rawbytes` or `fast_pattern` modifiers for the same content.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 150

03-02-2022-24064301-Hauptpost-0634

3.5.10 http_raw_cookie

The http_raw_cookie keyword is a content modifier that restricts the search to the extracted UNNORMALIZED Cookie Header field of a HTTP client request or a HTTP server response (per the configuration of HttpInspect 2.2.6).

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `http_raw_cookie` is specified. This keyword is dependent on the `enable_cookie` config option. The Cookie Header field will be extracted only when this option is configured.

Format

```
http_raw_cookie;
```

Examples

This rule constrains the search for the pattern "EFG" to the extracted Unnormalized Cookie Header field of a HTTP client request.

```
alert tcp any any -> any 80 (content:"ABC"; content:"EFG"; http_raw_cookie;)
```



NOTE

The `http_raw_cookie` modifier is not allowed to be used with the `rawbytes`, `http_cookie` or `fast_pattern` modifiers for the same content.

3.5.11 http_header

The http_header keyword is a content modifier that restricts the search to the extracted Header fields of a HTTP client request or a HTTP server response (per the configuration of HttpInspect 2.2.6).

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `http_header` is specified.

The extracted Header fields may be NORMALIZED, per the configuration of HttpInspect (see 2.2.6).

Format

```
http_header;
```

Examples

This rule constrains the search for the pattern "EFG" to the extracted Header fields of a HTTP client request or a HTTP server response.

```
alert tcp any any -> any 80 (content:"ABC"; content:"EFG"; http_header;)
```

NOTE

The `http_header` modifier is not allowed to be used with the `rawbytes` modifier for the same content.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 151

3.5.12 http_raw_header

The http_raw_header keyword is a content modifier that restricts the search to the extracted UNNORMALIZED Header fields of a HTTP client request or a HTTP server response (per the configuration of HttpInspect 2.2.6).

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `http_raw_header` is specified.

Format

```
http_raw_header;
```

Examples

This rule constrains the search for the pattern "EFG" to the extracted Header fields of a HTTP client request or a HTTP server response.

```
alert tcp any any -> any 80 (content:"ABC"; content:"EFG"; http_raw_header;)
```



NOTE

The `http_raw_header` modifier is not allowed to be used with the `rawbytes`, `http_header` or `fast_pattern` modifiers for the same content.

3.5.13 http_method

The http_method keyword is a content modifier that restricts the search to the extracted Method from a HTTP client request.

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `http_method` is specified.

Format

```
http_method;
```

Examples

This rule constrains the search for the pattern "GET" to the extracted Method from a HTTP client request.

```
alert tcp any any -> any 80 (content:"ABC"; content:"GET"; http_method;)
```

NOTE

The `http_method` modifier is not allowed to be used with the `rawbytes` modifier for the same content.

3.5.14 http_uri

The http_uri keyword is a content modifier that restricts the search to the NORMALIZED request URI field . Using a content rule option followed by a http_uri modifier is the same as using a uricontent by itself (see: 3.5.20).

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `http_uri` is specified.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 152

03-02-2022-24064301-HauptPost-0635

Format

```
http_uri;
```

Examples

This rule constrains the search for the pattern "EFG" to the NORMALIZED URI.

```
alert tcp any any -> any 80 (content:"ABC"; content:"EFG"; http_uri;)
```

⚠NOTE

The `http_uri` modifier is not allowed to be used with the `rawbytes` modifier for the same content.

3.5.15 http_raw_uri

The http_raw_uri keyword is a content modifier that restricts the search to the UNNORMALIZED request URI field .

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `http_raw_uri` is specified.

Format

```
http_raw_uri;
```

Examples

This rule constrains the search for the pattern "EFG" to the UNNORMALIZED URI.

```
alert tcp any any -> any 80 (content:"ABC"; content:"EFG"; http_raw_uri;)
```

⚠NOTE

The `http_raw_uri` modifier is not allowed to be used with the `rawbytes`, `http_uri` or `fast_pattern` modifiers for the same content.

3.5.16 http_stat_code

The http_stat_code keyword is a content modifier that restricts the search to the extracted Status code field from a HTTP server response.

As this keyword is a modifier to the previous `content` keyword, there must be a content in the rule before `http_stat_code` is specified.

The Status Code field will be extracted only if the extended_reponse_inspection is configured for the HttpInspect (see 2.2.6).

Format

```
http_stat_code;
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 153

1644709564601258531 9

Examples

This rule constrains the search for the pattern "200" to the extracted Status Code field of a HTTP server response.

```
alert tcp any any -> any 80 (content:"ABC"; content:"200"; http_stat_code;)
```

NOTE

The http_stat_code modifier is not allowed to be used with the rawbytes or fast_pattern modifiers for the same content.

3.5.17 http_stat_msg

The http_stat_msg keyword is a content modifier that restricts the search to the extracted Status Message field from a HTTP server response.

As this keyword is a modifier to the previous content keyword, there must be a content in the rule before http_stat_msg is specified.

The Status Message field will be extracted only if the extended_reponse_inspection is configured for the HttpInspect (see 2.2.6).

'64447095646012585319

Format

```
http_stat_msg;
```

Examples

This rule constrains the search for the pattern "Not Found" to the extracted Status Message field of a HTTP server response.

```
alert tcp any any -> any 80 (content:"ABC"; content:"Not Found"; http_stat_msg;)
```

NOTE

The http_stat_msg modifier is not allowed to be used with the rawbytes or fast_pattern modifiers for the same content.

3.5.18 http_encode

The http_encode keyword will enable alerting based on encoding type present in a HTTP client request or a HTTP server response (per the configuration of HttpInspect 2.2.6).

There are eleven keywords associated with http_encode. The keywords 'uri', 'header' and 'cookie' determine the HTTP fields used to search for a particular encoding type. The keywords 'utf8', 'double_encode', 'non_ascii', 'base36', 'uencode', 'ascii', 'iis_encode' and 'bare_byte' determine the encoding type which would trigger the alert. These keywords can be combined using a OR operation. Negation is allowed on these keywords.

The config option 'normalize_headers' needs to be turned on for rules to work with the keyword 'header'. The keyword 'cookie' is dependent on config options 'enable_cookie' and 'normalize_cookies' (see 2.2.6). This rule option will not be able to detect encodings if the specified HTTP fields are not NORMALIZED.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 154

03-02-2022-24064301-HauptPost-0636

Option	Description
uri	Check for the specified encoding type in HTTP client request URI field.
header	Check for the specified encoding type in HTTP request or HTTP response header fields (depending on the packet flow)
cookie	Check for the specified encoding type in HTTP request or HTTP response cookie header fields (depending on the packet flow)
utf8	Check for utf8 encoding in the specified buffer
double_encode	Check for double encoding in the specified buffer
non_ascii	Check for non-ASCII encoding in the specified buffer
base36	Check for base36 encoding in the specified buffer
uencode	Check for u-encoding in the specified buffer
bare_byte	Check for bare byte encoding in the specified buffer
ascii	Check for ascii encoding in the specified buffer
iis_encode	Check for IIS Unicode encoding in the specified buffer

Format

```
http_encode:<http buffer type>, [!]<encoding type>
http_encode:[uri|header|cookie], [!][<utf8|double_encode|non_ascii|base36|uencode|bare_byte|ascii|iis_e
```

Examples

```
alert tcp any any -> any any (msg:"UTF8/UEncode Encoding present"; http_encode:uri,utf8|uencode;)
alert tcp any any -> any any (msg:"No UTF8"; http_encode:uri,!utf8;)
```



NOTE

Negation(!) and OR(|) operations cannot be used in conjunction with each other for the http_encode keyword. The OR and negation operations work only on the encoding type field and not on http buffer type field.

3.5.19 fast_pattern

The fast_pattern keyword is a content modifier that sets the content within a rule to be used with the fast pattern matcher. Since the default behavior of fast pattern determination is to use the longest content in the rule, it is useful if a shorter content is more "unique" than the longer content, meaning the shorter content is less likely to be found in a packet than the longer content.

The fast pattern matcher is used to select only those rules that have a chance of matching by using a content in the rule for selection and only evaluating that rule if the content is found in the payload. Though this may seem to be overhead, it can significantly reduce the number of rules that need to be evaluated and thus increases performance. The better the content used for the fast pattern matcher, the less likely the rule will needlessly be evaluated.

As this keyword is a modifier to the previous content keyword, there must be a content rule option in the rule before fast_pattern is specified. The fast_pattern option may be specified only once per rule.

NOTE

The fast_pattern modifier cannot be used with the following http content modifiers: http_cookie, http_raw_uri, http_raw_header, http_raw_cookie, http_stat_code, http_stat_msg. Note, however, that it is okay to use the fast_pattern modifier if another http content modifier not mentioned above is used in combination with one of the above to modify the same content.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 155



> **NOTE**
> The `fast_pattern` modifier can be used with negated contents only if those contents are not modified with `offset`, `depth`, `distance` or `within`.

Format

The `fast_pattern` option can be used alone or optionally take arguments. When used alone, the meaning is simply to use the specified content as the fast pattern content for the rule.

```
fast_pattern;
```

The optional argument `only` can be used to specify that the content should only be used for the fast pattern matcher and should not be evaluated as a rule option. This is useful, for example, if a known content must be located in the payload independent of location in the payload, as it saves the time necessary to evaluate the rule option. Note that (1) the modified content must be case insensitive since patterns are inserted into the pattern matcher in a case insensitive manner, (2) negated contents cannot be used and (3) contents cannot have any positional modifiers such as `offset`, `depth`, `distance` or `within`.

```
fast_pattern:only;
```

The optional argument `<offset>,<length>` can be used to specify that only a portion of the content should be used for the fast pattern matcher. This is useful if the pattern is very long and only a portion of the pattern is necessary to satisfy "uniqueness" thus reducing the memory required to store the entire pattern in the fast pattern matcher.

```
fast_pattern:<offset>,<length>;
```

> **NOTE**
> The optional arguments `only` and `<offset>,<length>` are mutually exclusive.

Examples

This rule causes the pattern "IJKLMNO" to be used with the fast pattern matcher, even though it is shorter than the earlier pattern "ABCDEFGH".

```
alert tcp any any -> any 80 (content:"ABCDEFGH"; content:"IJKLMNO"; fast_pattern;)
```

This rule says to use the content "IJKLMNO" for the fast pattern matcher and that the content should only be used for the fast pattern matcher and not evaluated as a `content` rule option.

```
alert tcp any any -> any 80 (content:"ABCDEFGH"; content:"IJKLMNO"; nocase; fast_pattern:only;)
```

This rule says to use "JKLMN" as the fast pattern content, but still evaluate the `content` rule option as "IJKLMNO".

```
alert tcp any any -> any 80 (content:"ABCDEFGH"; content:"IJKLMNO"; fast_pattern:1,5;)
```

3.5.20 uricontent

The `uricontent` keyword in the Snort rule language searches the NORMALIZED request URI field. This is equivalent to using the `http_uri` modifier to a `content` keyword. As such if you are writing rules that include things that are normalized, such as %2f or directory traversals, these rules will not alert. The reason is that the things you are looking for are normalized out of the URI buffer.

For example, the URI:

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 156

03-02-2022-24064301-Hauptpost-0637

```
/scripts/..%c0%af../winnt/system32/cmd.exe?/c+ver
```

will get normalized into:

```
/winnt/system32/cmd.exe?/c+ver
```

Another example, the URI:

```
/cgi-bin/aaaaaaaaaaaaaaaaaaaaaaaaa/..%252fp%68f?
```

will get normalized into:

```
/cgi-bin/phf?
```

When writing a `uricontent` rule, write the content that you want to find in the context that the URI will be normalized. For example, if Snort normalizes directory traversals, do not include directory traversals.

You can write rules that look for the non-normalized content by using the content option. (See Section 3.5.1)

`uricontent` can be used with several of the modifiers available to the `content` keyword. These include:

Table 3.6: Uricontent Modifiers

Modifier	Section
nocase	3.5.2
depth	3.5.4
offset	3.5.5
distance	3.5.6
within	3.5.7
fast_pattern	3.5.19

This option works in conjunction with the HTTP Inspect preprocessor specified in Section 2.2.6.

Format

```
uricontent:[!]"<content string>";
```

NOTE

`uricontent` cannot be modified by a `rawbytes` modifier or any of the other HTTP modifiers. If you wish to search the UNNORMALIZED request URI field, use the `http_raw_uri` modifier with a `content` option.

3.5.21 urilen

The `urilen` keyword in the Snort rule language specifies the exact length, the minimum length, the maximum length, or range of URI lengths to match.

Format

```
urilen:min<>max;
urilen:[<|>]<number>;
```

The following example will match URIs that are 5 bytes long:

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 157

16444709564601258531 9

```
urilen:5;
```

The following example will match URIs that are shorter than 5 bytes:

```
urilen:<5;
```

The following example will match URIs that are greater than 5 bytes and less than 10 bytes:

```
urilen:5<>10;
```

This option works in conjunction with the HTTP Inspect preprocessor specified in Section 2.2.6.

3.5.22 isdataat

Verify that the payload has data at a specified location, optionally looking for data relative to the end of the previous content match.



Format

```
isdataat:[!]<int>[, relative|rawbytes];
```

Example

```
alert tcp any any -> any 111 (content:"PASS"; isdataat:50,relative; \
    content:!"|0a|"; within:50;)
```

This rule looks for the string PASS exists in the packet, then verifies there is at least 50 bytes after the end of the string PASS, then verifies that there is not a newline character within 50 bytes of the end of the PASS string.

When the `rawbytes` modifier is specified with `isdataat`, it looks at the raw packet data, ignoring any decoding that was done by the preprocessors. This modifier will work with the `relative` modifier as long as the previous content match was in the raw packet data.

A ! modifier negates the results of the isdataat test. It will alert if a certain amount of data is not present within the payload. For example, the rule with modifiers `content:"foo"; isdataat:!10,relative;` would alert if there were not 10 bytes after "foo" before the payload ended.

3.5.23 pcre

The pcre keyword allows rules to be written using perl compatible regular expressions. For more detail on what can be done via a pcre regular expression, check out the PCRE web site `http://www.pcre.org`

Format

```
pcre:[!]"(/<regex>/|m<delim><regex><delim>)[ismxAEGRUBPHMCOIDKYS]";
```

The post-re modifiers set compile time flags for the regular expression. See tables 3.7, 3.8, and 3.9 for descriptions of each modifier.

⚠ NOTE

The modifiers R (relative) and B (rawbytes) are not allowed with any of the HTTP modifiers such as U, I, P, H, D, M, C, K, S and Y.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 158

03-02-2022-24064301-Hauptpost-0638

Table 3.7: Perl compatible modifiers for `pcre`

i	case insensitive
s	include newlines in the dot metacharacter
m	By default, the string is treated as one big line of characters. ^ and $ match at the beginning and ending of the string. When m is set, ^ and $ match immediately following or immediately before any newline in the buffer, as well as the very start and very end of the buffer.
x	whitespace data characters in the pattern are ignored except when escaped or inside a character class

Table 3.8: PCRE compatible modifiers for `pcre`

A	the pattern must match only at the start of the buffer (same as ^)
E	Set $ to match only at the end of the subject string. Without E, $ also matches immediately before the final character if it is a newline (but not before any other newlines).
G	Inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?".



Example

This example performs a case-insensitive search for the string BLAH in the payload.

```
alert ip any any -> any any (pcre:"/BLAH/i";)
```



Snort's handling of multiple URIs with PCRE does not work as expected. PCRE when used without a `uricontent` only evaluates the first URI. In order to use pcre to inspect all URIs, you must use either a content or a uricontent.

3.5.24 file_data

This option is used to place the cursor (used to walk the packet payload in rules processing) at the beginning of either the entity body of a HTTP response or the SMTP body data. For this option to work with HTTP response, certain HTTP Inspect options such as `extended_response_inspection` and `inspect_gzip` (for decompressed gzip data) needs to be turned on. See 2.2.6 for more details.

When used with argument `mime` it places the cursor at the beginning of the base64 decoded MIME attachment or base64 decoded MIME body. This is dependent on the SMTP config option `enable_mime_decoding`. See 2.2.7 for more details.

Format

```
file_data;
file_data:mime;
```

This option matches if there is HTTP response body or SMTP body or SMTP MIME base64 decoded data. This option will operate similarly to the `dce_stub_data` option added with DCE/RPC2, in that it simply sets a reference for other relative rule options (byte test, byte jump, pcre) to use. This `file_data` can point to either a file or a block of data.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 159

Table 3.9: Snort specific modifiers for pcre

R	Match relative to the end of the last pattern match. (Similar to distance:0;)
U	Match the decoded URI buffers (Similar to uricontent and http_uri). This modifier is not allowed with the unnormalized HTTP request uri buffer modifier(I) for the same content.
I	Match the unnormalized HTTP request uri buffer (Similar to http_raw_uri). This modifier is not allowed with the HTTP request uri buffer modifier(U) for the same content.
P	Match unnormalized HTTP request body (Similar to http_client_body)
H	Match normalized HTTP request or HTTP response header (Similar to http_header). This modifier is not allowed with the unnormalized HTTP request or HTTP response header modifier(D) for the same content.
D	Match unnormalized HTTP request or HTTP response header (Similar to http_raw_header). This modifier is not allowed with the normalized HTTP request or HTTP response header modifier(H) for the same content.
M	Match normalized HTTP request method (Similar to http_method)
C	Match normalized HTTP request or HTTP response cookie (Similar to http_cookie). This modifier is not allowed with the unnormalized HTTP request or HTTP response cookie modifier(K) for the same content.
K	Match unnormalized HTTP request or HTTP response cookie (Similar to http_raw_cookie). This modifier is not allowed with the normalized HTTP request or HTTP response cookie modifier(C) for the same content.
S	Match HTTP response status code (Similar to http_stat_code)
Y	Match HTTP response status message (Similar to http_stat_msg)
B	Do not use the decoded buffers (Similar to rawbytes)
O	Override the configured pcre match limit and pcre match limit recursion for this expression (See section 2.1.3). It completely ignores the limits while evaluating the pcre pattern specified.



NOTE

Multiple base64 encoded attachments in one packet are pipelined.

Example

```
alert tcp any 80 -> any any(msg:"foo at the start of http response body"; \
file_data; content:"foo"; nocase; within:3;)
alert tcp any any -> any any(msg:"MIME BASE64 Encoded Data";\
file_data:mime; content:"foo"; within:10;)
```

3.5.25 base64_decode

This option is used to decode the base64 encoded data. This option is particularly useful in case of HTTP headers such as HTTP authorization headers. This option unfolds the data before decoding it.

Format

```
base64_decode[:[bytes <bytes_to_decode>][, ][offset <offset>[, relative]]];
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 160

03-02-2022-24064301-Hauptpost-0639

Option	Description
bytes	Number of base64 encoded bytes to decode. This argument takes positive and non-zero values only. When this option is not specified we look for base64 encoded data till either the end of header line is reached or end of packet payload is reached.
offset	Determines the offset relative to the doe_ptr when the option relative is specified or relative to the start of the packet payload to begin inspection of base64 encoded data. This argument takes positive and non-zero values only.
relative	Specifies the inspection for base64 encoded data is relative to the doe_ptr.

The above arguments to base64_decode are optional.

⚠NOTE

This option can be extended to protocols with folding similar to HTTP. If folding is not present the search for base64 encoded data will end when we see a carriage return or line feed or both without a following space or tab.
This option needs to be used in conjunction with base64_data for any other relative rule options to work on base64 decoded buffer.



Examples

```
    alert tcp $EXTERNAL_NET any -> $HOME_NET any \
(msg:"Base64 Encoded Data"; base64_decode; base64_data; \
content:"foo bar"; within:20;)

    alert tcp $EXTERNAL_NET any -> $HOME_NET any \
(msg:"Authorization NTLM"; content:"Authorization: NTLM";
base64_decode:relative; base64_data; content:"NTLMSSP"; \
within:20;)

    alert tcp any any -> any any (msg:"Authorization NTLM"; \
content:"Authorization:"; http_header; \
base64_decode:bytes 12, offset 6, relative; base64_data; \
content:"NTLMSSP"; within:8;)
```

3.5.26 base64_data

This option is used to place the cursor (used to walk the packet payload in rules processing) at the beginning of the base64 decode buffer if present. This option does not take any arguments. The rule option base64_decode needs to be specified before the base64_data option.

Format

```
base64_data;
```

This option matches if there is base64 decoded buffer. This option will operate similarly to the file_data option, in that it simply sets a reference for other relative rule options (byte test, byte jump, pcre) to use.

⚠NOTE

Any non-relative rule options in the rule will reset the cursor(doe_ptr) from base64 decode buffer.
Fast pattern content matches are not allowed with this buffer.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 161

Example

```
alert tcp any any -> any any (msg:"Authorization NTLM"; \
    content:"Authorization:"; http_header; \
    base64_decode:bytes 12, offset 6, relative; base64_data; \
    content:"NTLMSSP"; within:8;)
```

3.5.27 byte_test

Test a byte field against a specific value (with operator). Capable of testing binary values or converting representative byte strings to their binary equivalent and testing them.

For a more detailed explanation, please read Section 3.9.5.

Format

```
byte_test:<bytes to convert>, [!]<operator>, <value>, <offset>  \
    [, relative][, <endian>][, string, <number type>][, dce];

bytes      = 1 - 10
operator   = '<' | '=' | '>' | '&' | '^'
value      = 0 - 4294967295
offset     = -65535 to 65535
```

Option	Description
bytes_to_convert	Number of bytes to pick up from the packet. The allowed values are 1 to 10 when used without dce. If used with dce allowed values are 1, 2 and 4.
operator	Operation to perform to test the value: • < - less than • > - greater than • = - equal • & - bitwise AND • ^ - bitwise OR
value	Value to test the converted value against
offset	Number of bytes into the payload to start processing
relative	Use an offset relative to last pattern match
endian	Endian type of the number being read: • big - Process data as big endian (default) • little - Process data as little endian
string	Data is stored in string format in packet
number type	Type of number being read: • hex - Converted string data is represented in hexadecimal • dec - Converted string data is represented in decimal • oct - Converted string data is represented in octal
dce	Let the DCE/RPC 2 preprocessor determine the byte order of the value to be converted. See section 2.2.13 for a description and examples (2.2.13 for quick reference).

Any of the operators can also include ! to check if the operator is not true. If ! is specified without an operator, then the operator is set to =.

NOTE

Snort uses the C operators for each of these operators. If the & operator is used, then it would be the same as using *if (data & value) { do_something(); }*

Examples

```
alert udp $EXTERNAL_NET any -> $HOME_NET any \
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 162

1644709564601258531 9

03-02-2022-24064301-Hauptpost-0640

```
    (msg:"AMD procedure 7 plog overflow"; \
    content:"|00 04 93 F3|"; \
    content:"|00 00 00 07|"; distance:4; within:4; \
    byte_test:4, >, 1000, 20, relative;)

alert tcp $EXTERNAL_NET any -> $HOME_NET any \
    (msg:"AMD procedure 7 plog overflow"; \
    content:"|00 04 93 F3|"; \
    content:"|00 00 00 07|"; distance:4; within:4; \
    byte_test:4, >, 1000, 20, relative;)

alert udp any any -> any 1234 \
    (byte_test:4, =, 1234, 0, string, dec; \
    msg:"got 1234!";)

alert udp any any -> any 1235 \
    (byte_test:3, =, 123, 0, string, dec; \
    msg:"got 123!";)

alert udp any any -> any 1236 \
    (byte_test:2, =, 12, 0, string, dec; \
    msg:"got 12!";)

alert udp any any -> any 1237 \
    (byte_test:10, =, 1234567890, 0, string, dec; \
    msg:"got 1234567890!";)

alert udp any any -> any 1238 \
    (byte_test:8, =, 0xdeadbeef, 0, string, hex; \
    msg:"got DEADBEEF!";)
```

1644709564601258531 9

3.5.28 byte_jump

The byte_jump keyword allows rules to be written for length encoded protocols trivially. By having an option that reads the length of a portion of data, then skips that far forward in the packet, rules can be written that skip over specific portions of length-encoded protocols and perform detection in very specific locations.

The byte_jump option does this by reading some number of bytes, convert them to their numeric representation, move that many bytes forward and set a pointer for later detection. This pointer is known as the detect offset end pointer, or doe_ptr.

For a more detailed explanation, please read Section 3.9.5.

Format

```
byte_jump:<bytes_to_convert>, <offset> \
    [, relative][, multiplier <mult_value>][, <endian>][, string, <number_type>]\
    [, align][, from_beginning][, post_offset <adjustment value>][, dce];

bytes       = 1 - 10
offset      = -65535 to 65535
mult_value  = 0 - 65535
post_offset = -65535 to 65535
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 163

Option	Description
bytes_to_convert	Number of bytes to pick up from the packet. The allowed values are 1 to 10 when used without dce. If used with dce allowed values are 1, 2 and 4.
offset	Number of bytes into the payload to start processing
relative	Use an offset relative to last pattern match
multiplier <value>	Multiply the number of calculated bytes by <value> and skip forward that number of bytes.
big	Process data as big endian (default)
little	Process data as little endian
string	Data is stored in string format in packet
hex	Converted string data is represented in hexadecimal
dec	Converted string data is represented in decimal
oct	Converted string data is represented in octal
align	Round the number of converted bytes up to the next 32-bit boundary
from_beginning	Skip forward from the beginning of the packet payload instead of from the current position in the packet.
post_offset <value>	Skip forward or backwards (positive of negative value) by <value> number of bytes after the other jump options have been applied.
dce	Let the DCE/RPC 2 preprocessor determine the byte order of the value to be converted. See section 2.2.13 for a description and examples (2.2.13 for quick reference).

Example

```
alert udp any any -> any 32770:34000 (content:"|00 01 86 B8|"; \
    content:"|00 00 00 01|"; distance:4; within:4; \
    byte_jump:4, 12, relative, align; \
    byte_test:4, >, 900, 20, relative; \
    msg:"statd format string buffer overflow";)
```

16444709564601258531 9

3.5.29 byte_extract

The byte_extract keyword is another useful option for writing rules against length-encoded protocols. It reads in some number of bytes from the packet payload and saves it to a variable. These variables can be referenced later in the rule, instead of using hard-coded values.



Only two byte_extract variables may be created per rule. They can be re-used in the same rule any number of times.

Format

```
byte_extract:<bytes_to_extract>, <offset>, <name> \
    [, relative][, multiplier <multiplier value>][, <endian>]\
    [, string, <number_type>][, align <align value>][, dce]
```

Option	Description
bytes_to_convert	Number of bytes to pick up from the packet
offset	Number of bytes into the payload to start processing
name	Name of the variable. This will be used to reference the variable in other rule options.
relative	Use an offset relative to last pattern match
multiplier <value>	Multiply the bytes read from the packet by <value> and save that number into the variable.
big	Process data as big endian (default)
little	Process data as little endian
dce	Use the DCE/RPC 2 preprocessor to determine the byte-ordering. The DCE/RPC 2 preprocessor must be enabled for this option to work.
string	Data is stored in string format in packet
hex	Converted string data is represented in hexadecimal
dec	Converted string data is represented in decimal
oct	Converted string data is represented in octal
align <value>	Round the number of converted bytes up to the next <value>-byte boundary. <value> may be 2 or 4.

Other options which use byte_extract variables

A byte_extract rule option detects nothing by itself. Its use is in extracting packet data for use in other rule options. Here is a list of places where byte_extract variables can be used:

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 164

03-02-2022-24064301-Hauptpost-0641

Rule Option	Arguments that Take Variables
content/uricontent	offset, depth, distance, within
byte_test	offset, value
byte_jump	offset
isdataat	offset

Examples

This example uses two variables to:

- Read the offset of a string from a byte at offset 0.
- Read the depth of a string from a byte at offset 1.
- Use these values to constrain a pattern match to a smaller area.

```
alert tcp any any -> any any (byte_extract:1, 0, str_offset; \
    byte_extract:1, 1, str_depth; \
    content:"bad stuff"; offset:str_offset; depth:str_depth; \
    msg:"Bad Stuff detected within field";)
```

3.5.30 ftpbounce

The ftpbounce keyword detects FTP bounce attacks.

Format

```
ftpbounce;
```

Example

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP PORT bounce attempt"; \
    flow:to_server,established; content:"PORT"; nocase; ftpbounce; pcre:"/^PORT/smi";\
    classtype:misc-attack; sid:3441; rev:1;)
```

164447095646012585319

3.5.31 asn1

The ASN.1 detection plugin decodes a packet or a portion of a packet, and looks for various malicious encodings.

Multiple options can be used in an 'asn1' option and the implied logic is boolean OR. So if any of the arguments evaluate as true, the whole option evaluates as true.

The ASN.1 options provide programmatic detection capabilities as well as some more dynamic type detection. If an option has an argument, the option and the argument are separated by a space or a comma. The preferred usage is to use a space between option and argument.

Format

```
asn1:[bitstring_overflow][, double_overflow][, oversize_length <value>][, absolute_offset <value>|relative_offset <value>];
```

Option	Description
bitstring_overflow	Detects invalid bitstring encodings that are known to be remotely exploitable.
double_overflow	Detects a double ASCII encoding that is larger than a standard buffer. This is known to be an exploitable function in Microsoft, but it is unknown at this time which services may be exploitable.
oversize_length <value>	Compares ASN.1 type lengths with the supplied argument. The syntax looks like, "oversize_length 500". This means that if an ASN.1 type is greater than 500, then this keyword is evaluated as true. This keyword must have one argument which specifies the length to compare against.
absolute_offset <value>	This is the absolute offset from the beginning of the packet. For example, if you wanted to decode snmp packets, you would say "absolute_offset 0". absolute_offset has one argument, the offset value. Offset may be positive or negative.
relative_offset <value>	This is the relative offset from the last content match or byte_test/jump. relative_offset has one argument, the offset number. So if you wanted to start decoding and ASN.1 sequence right after the content "foo", you would specify 'content:"foo"; asn1:bitstring_overflow, relative_offset 0'. Offset values may be positive or negative.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 165

Examples

```
alert udp any any -> any 161 (msg:"Oversize SNMP Length"; \
    asn1:oversize_length 10000, absolute_offset 0;)

alert tcp any any -> any 80 (msg:"ASN1 Relative Foo"; content:"foo"; \
    asn1:bitstring_overflow, relative_offset 0;)
```

3.5.32 cvs

The CVS detection plugin aids in the detection of: Bugtraq-10384, CVE-2004-0396: "Malformed Entry Modified and Unchanged flag insertion". Default CVS server ports are 2401 and 514 and are included in the default ports for stream reassembly.

NOTE

This plugin cannot do detection over encrypted sessions, e.g. SSH (usually port 22).

Format

```
cvs:<option>;
```

Option	Description
invalid-entry	Looks for an invalid Entry string, which is a way of causing a heap overflow (see CVE-2004-0396) and bad pointer dereference in versions of CVS 1.11.15 and before.

Examples

```
alert tcp any any -> any 2401 (msg:"CVS Invalid-entry"; \
    flow:to_server,established; cvs:invalid-entry;)
```

1644470956460125853 19

3.5.33 dce_iface

See the DCE/RPC 2 Preprocessor section 2.2.13 for a description and examples of using this rule option.

3.5.34 dce_opnum

See the DCE/RPC 2 Preprocessor section 2.2.13 for a description and examples of using this rule option.

3.5.35 dce_stub_data

See the DCE/RPC 2 Preprocessor section 2.2.13 for a description and examples of using this rule option.

3.5.36 ssl_version

See the SSL/TLS Preprocessor section 2.2.11 for a description and examples of using this rule option.

3.5.37 ssl_state

See the SSL/TLS Preprocessor section 2.2.11 for a description and examples of using this rule option.

3.5.38 Payload Detection Quick Reference

Table 3.10: Payload detection rule option keywords

Keyword	Description
content	The content keyword allows the user to set rules that search for specific content in the packet payload and trigger response based on that data.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 166

03-02-2022-24064301-Hauptpost-0642

rawbytes	The rawbytes keyword allows rules to look at the raw packet data, ignoring any decoding that was done by preprocessors.
depth	The depth keyword allows the rule writer to specify how far into a packet Snort should search for the specified pattern.
offset	The offset keyword allows the rule writer to specify where to start searching for a pattern within a packet.
distance	The distance keyword allows the rule writer to specify how far into a packet Snort should ignore before starting to search for the specified pattern relative to the end of the previous pattern match.
within	The within keyword is a content modifier that makes sure that at most N bytes are between pattern matches using the content keyword.
uricontent	The uricontent keyword in the Snort rule language searches the normalized request URI field.
isdataat	The isdataat keyword verifies that the payload has data at a specified location.
pcre	The pcre keyword allows rules to be written using perl compatible regular expressions.
byte_test	The byte_test keyword tests a byte field against a specific value (with operator).
byte_jump	The byte_jump keyword allows rules to read the length of a portion of data, then skip that far forward in the packet.
ftpbounce	The ftpbounce keyword detects FTP bounce attacks.
asn1	The asn1 detection plugin decodes a packet or a portion of a packet, and looks for various malicious encodings.
cvs	The cvs keyword detects invalid entry strings.
dce_iface	See the DCE/RPC 2 Preprocessor section 2.2.13.
dce_opnum	See the DCE/RPC 2 Preprocessor section 2.2.13.
dce_stub_data	See the DCE/RPC 2 Preprocessor section 2.2.13.



3.6 Non-Payload Detection Rule Options

3.6.1 fragoffset

The fragoffset keyword allows one to compare the IP fragment offset field against a decimal value. To catch all the first fragments of an IP session, you could use the fragbits keyword and look for the More fragments option in conjunction with a fragoffset of 0.

Format

```
fragoffset:[!|<|>]<number>;
```

Example

```
alert ip any any -> any any \
    (msg:"First Fragment"; fragbits:M; fragoffset:0;)
```

3.6.2 ttl

The ttl keyword is used to check the IP time-to-live value. This option keyword was intended for use in the detection of traceroute attempts. This keyword takes numbers from 0 to 255.

Format

```
ttl:[<, >, =, <=, >=]<number>;
ttl:[<number>]-[<number>];
```

Example

This example checks for a time-to-live value that is less than 3.

```
ttl:<3;
```

This example checks for a time-to-live value that between 3 and 5.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 167

```
ttl:3-5;
```

This example checks for a time-to-live value that between 0 and 5.

```
ttl:-5;
```

This example checks for a time-to-live value that between 5 and 255.

```
ttl:5-;
```

Few other examples are as follows:

```
ttl:<=5;
ttl:>=5;
ttl:=5;
```

The following examples are NOT allowed by ttl keyword:

```
ttl:=>5;
ttl:=<5;
ttl:5-3;
```



3.6.3 tos

The tos keyword is used to check the IP TOS field for a specific value.

Format

```
tos:[!]<number>;
```

Example

This example looks for a tos value that is not 4

```
tos:!4;
```

3.6.4 id

The id keyword is used to check the IP ID field for a specific value. Some tools (exploits, scanners and other odd programs) set this field specifically for various purposes, for example, the value 31337 is very popular with some hackers.

Format

```
id:<number>;
```

Example

This example looks for the IP ID of 31337.

```
id:31337;
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 168

03-02-2022-24064301-Hauptpost-0643

3.6.5 ipopts

The ipopts keyword is used to check if a specific IP option is present.

The following options may be checked:

rr - Record Route
eol - End of list
nop - No Op
ts - Time Stamp
sec - IP Security
esec - IP Extended Security
lsrr - Loose Source Routing
lsrre - Loose Source Routing (For MS99-038 and CVE-1999-0909)
ssrr - Strict Source Routing
satid - Stream identifier
any - any IP options are set

The most frequently watched for IP options are strict and loose source routing which aren't used in any widespread internet applications.

Format

```
ipopts:<rr|eol|nop|ts|sec|esec|lsrr|lsrre|ssrr|satid|any>;
```

Example

This example looks for the IP Option of Loose Source Routing.

```
ipopts:lsrr;
```

Warning

Only a single ipopts keyword may be specified per rule.

1644709564601258531 9

3.6.6 fragbits

The `fragbits` keyword is used to check if fragmentation and reserved bits are set in the IP header.

The following bits may be checked:

M - More Fragments
D - Don't Fragment
R - Reserved Bit

The following modifiers can be set to change the match criteria:

+ match on the specified bits, plus any others
***** match if any of the specified bits are set
! match if the specified bits are not set

Format

```
fragbits:[+*!]<[MDR]>;
```

Example

This example checks if the More Fragments bit and the Do not Fragment bit are set.

```
fragbits:MD+;
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 169

3.6.7 dsize

The dsize keyword is used to test the packet payload size. This may be used to check for abnormally sized packets. In many cases, it is useful for detecting buffer overflows.

Format

```
dsize:min<>max;
dsize:[<|>]<number>;
```

Example

This example looks for a dsize that is between 300 and 400 bytes.

```
dsize:300<>400;
```

Warning

dsize will fail on stream rebuilt packets, regardless of the size of the payload.



3.6.8 flags

The flags keyword is used to check if specific TCP flag bits are present.

The following bits may be checked:

F - FIN - Finish (LSB in TCP Flags byte)

S - SYN - Synchronize sequence numbers

R - RST - Reset

P - PSH - Push

A - ACK - Acknowledgment

U - URG - Urgent

1 - CWR - Congestion Window Reduced (MSB in TCP Flags byte)

2 - ECE - ECN-Echo (If SYN, then ECN capable. Else, CE flag in IP header is set)

0 - No TCP Flags Set

The following modifiers can be set to change the match criteria:

+ - match on the specified bits, plus any others

***** - match if any of the specified bits are set

! - match if the specified bits are not set

To handle writing rules for session initiation packets such as ECN where a SYN packet is sent with the previously reserved bits 1 and 2 set, an option mask may be specified. A rule could check for a flags value of S,12 if one wishes to find packets with just the syn bit, regardless of the values of the reserved bits.

Format

```
flags:[!|*|+]<FSRPAU120>[,<FSRPAU12>];
```

Example

This example checks if just the SYN and the FIN bits are set, ignoring reserved bit 1 and reserved bit 2.

```
alert tcp any any -> any any (flags:SF,12;)
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 170

03-02-2022-24064301-Hauptpost-0644

3.6.9 flow

The flow keyword is used in conjunction with TCP stream reassembly (see Section 2.2.2). It allows rules to only apply to certain directions of the traffic flow.

This allows rules to only apply to clients or servers. This allows packets related to $HOME_NET clients viewing web pages to be distinguished from servers running in the $HOME_NET.

The established keyword will replace the `flags:+A` used in many places to show established TCP connections.

Options

Option	Description
to_client	Trigger on server responses from A to B
to_server	Trigger on client requests from A to B
from_client	Trigger on client requests from A to B
from_server	Trigger on server responses from A to B
established	Trigger only on established TCP connections
not_established	Trigger only when no TCP connection is established
stateless	Trigger regardless of the state of the stream processor (useful for packets that are designed to cause machines to crash)
no_stream	Do not trigger on rebuilt stream packets (useful for dsize and stream5)
only_stream	Only trigger on rebuilt stream packets
no_frag	Do not trigger on rebuilt frag packets
only_frag	Only trigger on rebuilt frag packets

Format

```
flow:[(established|not_established|stateless)]
    [,(to_client|to_server|from_client|from_server)]
    [,(no_stream|only_stream)]
    [,(no_frag|only_frag)];
```

Examples

```
alert tcp !$HOME_NET any -> $HOME_NET 21 (msg:"cd incoming detected"; \
    flow:from_client; content:"CWD incoming"; nocase;)

alert tcp !$HOME_NET 0 -> $HOME_NET 0 (msg:"Port 0 TCP traffic"; \
    flow:stateless;)
```

3.6.10 flowbits

The `flowbits` keyword is used in conjunction with conversation tracking from the Stream preprocessor (see Section2.2.2). It allows rules to track states during a transport protocol session. The flowbits option is most useful for TCP sessions, as it allows rules to generically track the state of an application protocol.

There are eight keywords associated with flowbits. Most of the options need a user-defined name for the specific state that is being checked. This string should be limited to any alphanumeric string including periods, dashes, and underscores. The keywords set and toggle take an optional argument which specifies the group to which the keywords will belong. When no group name is specified the flowbits will belong to a default group. All the flowbits in a particular group (with an exception of default group) are mutually exclusive. A particular flow cannot belong to more than one group.

Option	Description
set	Sets the specified state for the current flow and unsets all the other flowbits in a group when a GROUP_NAME is specified.
unset	Unsets the specified state for the current flow.
toggle	Sets the specified state if the state is unset and unsets all the other flowbits in a group when a GROUP_NAME is specified, otherwise unsets the state if the state is set.
isset	Checks if the specified state is set.
isnotset	Checks if the specified state is not set.
noalert	Cause the rule to not generate an alert, regardless of the rest of the detection options.
reset	Reset all states on a given flow.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 171

16444709564601258531 9

Format

```
flowbits:[set|unset|toggle|isset|isnotset|noalert|reset][, <STATE_NAME>][, <GROUP_NAME>];
```

Examples

```
alert tcp any 143 -> any any (msg:"IMAP login";
    content:"OK LOGIN"; flowbits:set,logged_in;
    flowbits:noalert;)

alert tcp any any -> any 143 (msg:"IMAP LIST"; content:"LIST";
    flowbits:isset,logged_in;)
```

3.6.11 seq

The seq keyword is used to check for a specific TCP sequence number.

Format

```
seq:<number>;
```

Example

This example looks for a TCP sequence number of 0.

```
seq:0;
```

3.6.12 ack

The ack keyword is used to check for a specific TCP acknowledge number.

Format

```
ack:<number>;
```

Example

This example looks for a TCP acknowledge number of 0.

```
ack:0;
```

3.6.13 window

The window keyword is used to check for a specific TCP window size.

Format

```
window:[!]<number>;
```

Example

This example looks for a TCP window size of 55808.

```
window:55808;
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 172

'64447095646012585319

03-02-2022-24064301-Hauptpost-0645

3.6.14 itype

The itype keyword is used to check for a specific ICMP type value.

Format

```
itype:min<>max;
itype:[<|>]<number>;
```

Example

This example looks for an ICMP type greater than 30.

```
itype:>30;
```

3.6.15 icode

The icode keyword is used to check for a specific ICMP code value.



Format

```
icode:min<>max;
icode:[<|>]<number>;
```

Example

This example looks for an ICMP code greater than 30.

```
icode:>30;
```

3.6.16 icmp_id

The icmp_id keyword is used to check for a specific ICMP ID value.

This is useful because some covert channel programs use static ICMP fields when they communicate. This particular plugin was developed to detect the stacheldraht DDoS agent.

Format

```
icmp_id:<number>;
```

Example

This example looks for an ICMP ID of 0.

```
icmp_id:0;
```

3.6.17 icmp_seq

The icmp_seq keyword is used to check for a specific ICMP sequence value.

This is useful because some covert channel programs use static ICMP fields when they communicate. This particular plugin was developed to detect the stacheldraht DDoS agent.

Format

```
icmp_seq:<number>;
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 173

Example

This example looks for an ICMP Sequence of 0.

```
icmp_seq:0;
```

3.6.18 rpc

The rpc keyword is used to check for a RPC application, version, and procedure numbers in SUNRPC CALL requests.

Wildcards are valid for both version and procedure numbers by using '*';

Format

```
rpc:<application number>, [<version number>|*], [<procedure number>|*];
```

Example

The following example looks for an RPC portmap GETPORT request.

```
alert tcp any any -> any 111 (rpc:100000, *, 3;);
```



Warning

Because of the fast pattern matching engine, the RPC keyword is slower than looking for the RPC values by using normal content matching.

3.6.19 ip_proto

The ip_proto keyword allows checks against the IP protocol header. For a list of protocols that may be specified by name, see /etc/protocols.

Format

```
ip_proto:[!|>|<] <name or number>;
```

Example

This example looks for IGMP traffic.

```
alert ip any any -> any any (ip_proto:igmp;)
```

3.6.20 sameip

The sameip keyword allows rules to check if the source ip is the same as the destination IP.

Format

```
sameip;
```

Example

This example looks for any traffic where the Source IP and the Destination IP is the same.

```
alert ip any any -> any any (sameip;)
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 174

03-02-2022-24064301-Hauptpost-0646

3.6.21 stream_reassemble

The stream_reassemble keyword allows a rule to enable or disable TCP stream reassembly on matching traffic.

> **NOTE**
> The stream_reassemble option is only available when the Stream5 preprocessor is enabled.

Format

```
stream_reassemble:<enable|disable>, <server|client|both>[, noalert][, fastpath];
```

- The optional `noalert` parameter causes the rule to not generate an alert when it matches.
- The optional `fastpath` parameter causes Snort to ignore the rest of the connection.

Example

For example, to disable TCP reassembly for client traffic when we see a HTTP 200 Ok Response message, use:

```
alert tcp any 80 -> any any (flow:to_client, established; content:"200 OK";
   stream_reassemble:disable,client,noalert;)
```

1644470956460125853 19

3.6.22 stream_size

The stream_size keyword allows a rule to match traffic according to the number of bytes observed, as determined by the TCP sequence numbers.

> **NOTE**
> The stream_size option is only available when the Stream5 preprocessor is enabled.

Format

```
stream_size:<server|client|both|either>, <operator>, <number>;
```

Where the operator is one of the following:

- < - less than
- > - greater than
- = - equal
- != - not equal
- <= - less than or equal
- >= - greater than or equal

Example

For example, to look for a session that is less that 6 bytes from the client side, use:

```
alert tcp any any -> any any (stream_size:client,<,6;)
```

3.6.23 Non-Payload Detection Quick Reference

Table 3.11: Non-payload detection rule option keywords

Keyword	Description
`fragoffset`	The fragoffset keyword allows one to compare the IP fragment offset field against a decimal value.
`ttl`	The ttl keyword is used to check the IP time-to-live value.
`tos`	The tos keyword is used to check the IP TOS field for a specific value.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 175

id	The id keyword is used to check the IP ID field for a specific value.
ipopts	The ipopts keyword is used to check if a specific IP option is present.
fragbits	The fragbits keyword is used to check if fragmentation and reserved bits are set in the IP header.
dsize	The dsize keyword is used to test the packet payload size.
flags	The flags keyword is used to check if specific TCP flag bits are present.
flow	The flow keyword allows rules to only apply to certain directions of the traffic flow.
flowbits	The flowbits keyword allows rules to track states during a transport protocol session.
seq	The seq keyword is used to check for a specific TCP sequence number.
ack	The ack keyword is used to check for a specific TCP acknowledge number.
window	The window keyword is used to check for a specific TCP window size.
itype	The itype keyword is used to check for a specific ICMP type value.
icode	The icode keyword is used to check for a specific ICMP code value.
icmp_id	The icmp_id keyword is used to check for a specific ICMP ID value.
icmp_seq	The icmp_seq keyword is used to check for a specific ICMP sequence value.
rpc	The rpc keyword is used to check for a RPC application, version, and procedure numbers in SUNRPC CALL requests.
ip_proto	The ip_proto keyword allows checks against the IP protocol header.
sameip	The sameip keyword allows rules to check if the source ip is the same as the destination IP.



3.7 Post-Detection Rule Options

3.7.1 logto

The logto keyword tells Snort to log all packets that trigger this rule to a special output log file. This is especially handy for combining data from things like NMAP activity, HTTP CGI scans, etc. It should be noted that this option does not work when Snort is in binary logging mode.

Format

```
logto:"filename";
```

3.7.2 session

The session keyword is built to extract user data from TCP Sessions. There are many cases where seeing what users are typing in telnet, rlogin, ftp, or even web sessions is very useful.

There are three available argument keywords for the session rule option: `printable`, `binary`, or `all`.

The `printable` keyword only prints out data that the user would normally see or be able to type. The `binary` keyword prints out data in a binary format. The `all` keyword substitutes non-printable characters with their hexadecimal equivalents.

Format

```
session:[printable|binary|all];
```

Example

The following example logs all printable strings in a telnet packet.

```
log tcp any any <> any 23 (session:printable;)
```

Given an FTP data session on port 12345, this example logs the payload bytes in binary form.

```
log tcp any any <> any 12345 (metadata:service ftp-data; session:binary;)
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 176

Warnings

Using the session keyword can slow Snort down considerably, so it should not be used in heavy load situations. The session keyword is best suited for post-processing binary (pcap) log files.

The `binary` keyword does not log any protocol headers below the application layer, and Stream reassembly will cause duplicate data when the reassembled packets are logged.

3.7.3 resp

The resp keyword enables an active response that kills the offending session. Resp can be used in both passive or inline modes. See 2.11.3 for details.

3.7.4 react

The react keyword enables an active response that includes sending a web page or other content to the client and then closing the connection. React can be used in both passive and inline modes. See 2.11.4 for details.

3.7.5 tag

The tag keyword allow rules to log more than just the single packet that triggered the rule. Once a rule is triggered, additional traffic involving the source and/or destination host is *tagged*. Tagged traffic is logged to allow analysis of response codes and post-attack traffic. ***tagged*** alerts will be sent to the same output plugins as the original alert, but it is the responsibility of the output plugin to properly handle these special alerts. Currently, the database output plugin, described in Section 2.6.6, does not properly handle *tagged* alerts.

1644709564601258531 9

Format

```
tag:<type>, <count>, <metric>[, direction];
```

type

- `session` - Log packets in the session that set off the rule
- `host` - Log packets from the host that caused the tag to activate (uses [direction] modifier)

count

- `<integer>` - Count is specified as a number of units. Units are specified in the <metric> field.

metric

- `packets` - Tag the host/session for <count> packets
- `seconds` - Tag the host/session for <count> seconds
- `bytes` - Tag the host/session for <count> bytes

direction - only relevant if host type is used.

- `src` - Tag packets containing the source IP address of the packet that generated the initial event.
- `dst` - Tag packets containing the destination IP address of the packet that generated the initial event.

Note that neither subsequent alerts nor event filters will prevent a tagged packet from being logged. Subsequent tagged alerts will cause the limit to reset.

```
alert tcp any any <> 10.1.1.1 any (flowbits:isnotset,tagged;
    flowbits:set,tagged; tag:host,600,seconds,src;)
```

Also note that if you have a tag option in a rule that uses a metric other than `packets`, a `tagged_packet_limit` will be used to limit the number of tagged packets regardless of whether the `seconds` or `bytes` count has been reached. The default `tagged_packet_limit` value is 256 and can be modified by using a config option in your snort.conf file (see Section 2.1.3 on how to use the `tagged_packet_limit` config option). You can disable this packet limit for a particular rule by adding a `packets` metric to your tag option and setting its count to 0 (This can be done on a global scale by setting the `tagged_packet_limit` option in snort.conf to 0). Doing this will ensure that packets are tagged for the full amount of `seconds` or `bytes` and will not be cut off by the `tagged_packet_limit`. (Note that the `tagged_packet_limit` was introduced to avoid DoS situations on high bandwidth sensors for tag rules with a high `seconds` or `bytes` counts.)

```
alert tcp 10.1.1.4 any -> 10.1.1.1 any \
    (content:"TAGMYPACKETS"; tag:host,0,packets,600,seconds,src;)
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 177

Example

This example logs the first 10 seconds or the tagged_packet_limit (whichever comes first) of any telnet session.

```
alert tcp any any -> any 23 (flags:s,12; tag:session,10,seconds;)
```

3.7.6 activates

The activates keyword allows the rule writer to specify a rule to add when a specific network event occurs. See Section 3.2.6 for more information.

Format

```
activates:1;
```

3.7.7 activated_by

The activated_by keyword allows the rule writer to dynamically enable a rule when a specific activate rule is triggered. See Section 3.2.6 for more information.



Format

```
activated_by:1;
```

3.7.8 count

The count keyword must be used in combination with the activated_by keyword. It allows the rule writer to specify how many packets to leave the rule enabled for after it is activated. See Section 3.2.6 for more information.

Format

```
activated_by:1; count:50;
```

3.7.9 replace

The replace keyword is a feature available in inline mode which will cause Snort to replace the prior matching content with the given string. Both the new string and the content it is to replace must have the same length. You can have multiple replacements within a rule, one per content.

```
replace:"<string>";
```

3.7.10 detection_filter

detection_filter defines a rate which must be exceeded by a source or destination host before a rule can generate an event. detection_filter has the following format:

```
detection_filter: \
    track <by_src|by_dst>, \
    count <c>, seconds <s>;
```

Snort evaluates a detection_filter as the last step of the detection phase, after evaluating all other rule options (regardless of the position of the filter within the rule source). At most one detection_filter is permitted per rule.

Example - this rule will fire on every failed login attempt from 10.1.2.100 during one sampling period of 60 seconds, after the first 30 failed login attempts:

```
drop tcp 10.1.2.100 any > 10.1.1.100 22 ( \
    msg:"SSH Brute Force Attempt"; \
    flow:established,to_server; \
    content:"SSH"; nocase; offset:0; depth:4; \
    detection_filter:track by_src, count 30, seconds 60; \
    sid:1000001; rev:1;)
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 178

03-02-2022-24064301-Hauptpost-0648

Option	Description
`track by_src|by_dst`	Rate is tracked either by source IP address or destination IP address. This means count is maintained for each unique source IP address or each unique destination IP address.
`count c`	The maximum number of rule matches in s seconds allowed before the detection filter limit to be exceeded. C must be nonzero.
`seconds s`	Time period over which count is accrued. The value must be nonzero.

Since potentially many events will be generated, a `detection_filter` would normally be used in conjunction with an `event_filter` to reduce the number of logged events.

3.7.11 Post-Detection Quick Reference

Table 3.12: Post-detection rule option keywords

Keyword	Description
`logto`	The logto keyword tells Snort to log all packets that trigger this rule to a special output log file.
`session`	The session keyword is built to extract user data from TCP Sessions.
`resp`	The resp keyword is used attempt to close sessions when an alert is triggered.
`react`	This keyword implements an ability for users to react to traffic that matches a Snort rule by closing connection and sending a notice.
`tag`	The tag keyword allow rules to log more than just the single packet that triggered the rule.
`activates`	This keyword allows the rule writer to specify a rule to add when a specific network event occurs.
`activated_by`	This keyword allows the rule writer to dynamically enable a rule when a specific activate rule is triggered.
`count`	This keyword must be used in combination with the `activated_by` keyword. It allows the rule writer to specify how many packets to leave the rule enabled for after it is activated.
`replace`	Replace the prior matching content with the given string of the same length. Available in inline mode only.
`detection_filter`	Track by source or destination IP address and if the rule otherwise matches more than the configured rate it will fire.

16444709564601258531 9

3.8 Rule Thresholds

> **NOTE**
> Rule thresholds are deprecated and will not be supported in a future release. Use `detection_filters` (3.7.10) within rules, or `event_filters` (2.4.2) as standalone configurations instead.

`threshold` can be included as part of a rule, or you can use standalone thresholds that reference the generator and SID they are applied to. There is no functional difference between adding a threshold to a rule, or using a standalone threshold applied to the same rule. There is a logical difference. Some rules may only make sense with a threshold. These should incorporate the threshold into the rule. For instance, a rule for detecting a too many login password attempts may require more than 5 attempts. This can be done using the 'limit' type of threshold. It makes sense that the threshold feature is an integral part of this rule.

Format

```
threshold: \
    type <limit|threshold|both>, \
    track <by_src|by_dst>, \
    count <c>, seconds <s>;
```

Examples

This rule logs the first event of this SID every 60 seconds.

```
alert tcp $external_net any -> $http_servers $http_ports \
    (msg:"web-misc robots.txt access"; flow:to_server, established; \
    uricontent:"/robots.txt"; nocase; reference:nessus,10302; \
    classtype:web-application-activity; threshold:type limit, track \
    by_src, count 1, seconds 60; sid:1000852; rev:1;)
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 179

Option	Description
type limit|threshold|both	type limit alerts on the 1st m events during the time interval, then ignores events for the rest of the time interval. Type threshold alerts every m times we see this event during the time interval. Type both alerts once per time interval after seeing m occurrences of the event, then ignores any additional events during the time interval.
track by_src|by_dst	rate is tracked either by source IP address, or destination IP address. This means count is maintained for each unique source IP addresses, or for each unique destination IP addresses. Ports or anything else are not tracked.
count c	number of rule matching in s seconds that will cause event_filter limit to be exceeded. c must be nonzero value.
seconds s	time period over which count is accrued. s must be nonzero value.

```
alert tcp $external_net any -> $http_servers $http_ports \
    (msg:"web-misc robots.txt access"; flow:to_server, established; \
    uricontent:"/robots.txt"; nocase; reference:nessus,10302; \
    classtype:web-application-activity; threshold:type threshold, \
    track by_dst, count 10 , seconds 60 ; sid:1000852; rev:1;)
```

This rule logs at most one event every 60 seconds if at least 10 events on this SID are fired.

```
alert tcp $external_net any -> $http_servers $http_ports \
    (msg:"web-misc robots.txt access"; flow:to_server, established; \
    uricontent:"/robots.txt"; nocase; reference:nessus,10302; \
    classtype:web-application-activity; threshold:type both, track \
    by_dst, count 10, seconds 60; sid:1000852; rev:1;)
```

1644470956460125853 19

3.9 Writing Good Rules

There are some general concepts to keep in mind when developing Snort rules to maximize efficiency and speed.

3.9.1 Content Matching

Snort groups rules by protocol (ip, tcp, udp, icmp), then by ports (ip and icmp use slightly different logic), then by those with content and those without. For rules with content, a multi-pattern matcher is used to select rules that have a chance at matching based on a single content. Selecting rules for evaluation via this "fast" pattern matcher was found to increase performance, especially when applied to large rule groups like HTTP. The longer and more unique a content is, the less likely that rule and all of it's rule options will be evaluated unnecessarily - it's safe to say there is generally more "good" traffic than "bad". Rules without content are always evaluated (relative to the protocol and port group in which they reside), potentially putting a drag on performance. While some detection options, such as pcre and byte_test, perform detection in the payload section of the packet, they are not used by the fast pattern matching engine. If at all possible, try and have at least one content (or uricontent) rule option in your rule.

3.9.2 Catch the Vulnerability, Not the Exploit

Try to write rules that target the vulnerability, instead of a specific exploit.

For example, look for a the vulnerable command with an argument that is too large, instead of shellcode that binds a shell.

By writing rules for the vulnerability, the rule is less vulnerable to evasion when an attacker changes the exploit slightly.

3.9.3 Catch the Oddities of the Protocol in the Rule

Many services typically send the commands in upper case letters. FTP is a good example. In FTP, to send the username, the client sends:

```
user username_here
```

A simple rule to look for FTP root login attempts could be:

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 180

```
alert tcp any any -> any any 21 (content:"user root";)
```

While it may *seem* trivial to write a rule that looks for the username root, a good rule will handle all of the odd things that the protocol might handle when accepting the user command.

For example, each of the following are accepted by most FTP servers:

```
user root
user root
user root
user root
user<tab>root
```

To handle all of the cases that the FTP server might handle, the rule needs more smarts than a simple string match.

A good rule that looks for root login on ftp would be:

```
alert tcp any any -> any 21 (flow:to_server,established; \
    content:"root"; pcre:"/user\s+root/i";)
```

There are a few important things to note in this rule:

- The rule has a *flow* option, verifying this is traffic going to the server on an established session.
- The rule has a *content* option, looking for *root*, which is the longest, most unique string in the attack. This option is added to allow the fast pattern matcher to select this rule for evaluation only if the content *root* is found in the payload.
- The rule has a *pcre* option, looking for user, followed at least one space character (which includes tab), followed by root, ignoring case.

164447095646012585319

3.9.4 Optimizing Rules

The content matching portion of the detection engine has recursion to handle a few evasion cases. Rules that are not properly written can cause Snort to waste time duplicating checks.

The way the recursion works now is if a pattern matches, and if any of the detection options after that pattern fail, then look for the pattern again after where it was found the previous time. Repeat until the pattern is not found again or the opt functions all succeed.

On first read, that may not sound like a smart idea, but it is needed. For example, take the following rule:

```
alert ip any any -> any any (content:"a"; content:"b"; within:1;)
```

This rule would look for "a", immediately followed by "b". Without recursion, the payload "aab" would fail, even though it is obvious that the payload "aab" has "a" immediately followed by "b", because the first "a" is not immediately followed by "b".

While recursion is important for detection, the recursion implementation is not very smart.

For example, the following rule options are not optimized:

```
content:"|13|"; dsize:1;
```

By looking at this rule snippit, it is obvious the rule looks for a packet with a single byte of 0x13. However, because of recursion, a packet with 1024 bytes of 0x13 could cause 1023 too many pattern match attempts and 1023 too many dsize checks. Why? The content 0x13 would be found in the first byte, then the dsize option would fail, and because of recursion, the content 0x13 would be found again starting after where the previous 0x13 was found, once it is found, then check the dsize again, repeating until 0x13 is not found in the payload again.

Reordering the rule options so that discrete checks (such as dsize) are moved to the beginning of the rule speed up Snort.

The optimized rule snipping would be:

```
dsize:1; content:"|13|";
```

A packet of 1024 bytes of 0x13 would fail immediately, as the dsize check is the first option checked and dsize is a discrete check without recursion.

The following rule options are discrete and should generally be placed at the beginning of any rule:

- dsize
- flags
- flow

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 181

- fragbits
- icmp_id
- icmp_seq
- icode
- id
- ipopts
- ip_proto
- itype
- seq
- session
- tos
- ttl
- ack
- window
- resp
- sameip

16444709564601585319

3.9.5 Testing Numerical Values

The rule options *byte_test* and *byte_jump* were written to support writing rules for protocols that have length encoded data. RPC was the protocol that spawned the requirement for these two rule options, as RPC uses simple length based encoding for passing data.

In order to understand *why* byte_test and byte_jump are useful, let's go through an exploit attempt against the sadmind service.

This is the payload of the exploit:

```
89 09 9c e2 00 00 00 00 00 00 00 02 00 01 87 88  ................
00 00 00 0a 00 00 00 01 00 00 00 01 00 00 00 20  ................
40 28 3a 10 00 00 00 0a 4d 45 54 41 53 50 4c 4f  @(:.....metasplo
49 54 00 00 00 00 00 00 00 00 00 00 00 00 00 00  it..............
00 00 00 00 00 00 00 00 40 28 3a 14 00 07 45 df  ........@(:...e.
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 04  ................
7f 00 00 01 00 01 87 88 00 00 00 0a 00 00 00 04  ................
7f 00 00 01 00 01 87 88 00 00 00 0a 00 00 00 11  ................
00 00 00 1e 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 3b 4d 45 54 41 53 50 4c 4f  .......;metasplo
49 54 00 00 00 00 00 00 00 00 00 00 00 00 00 00  it..............
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 06 73 79 73 74 65 6d 00 00  ........system..
00 00 00 15 2e 2e 2f 2e 2e 2f 2e 2e 2f 2e 2e 2f  ....../../../../
2e 2e 2f 62 69 6e 2f 73 68 00 00 00 00 00 04 1e  ../bin/sh.......
<snip>
```

Let's break this up, describe each of the fields, and figure out how to write a rule to catch this exploit.

There are a few things to note with RPC:

- Numbers are written as uint32s, taking four bytes. The number 26 would show up as 0x0000001a.
- Strings are written as a uint32 specifying the length of the string, the string, and then null bytes to pad the length of the string to end on a 4 byte boundary. The string "bob" would show up as 0x00000003626f6200.

```
89 09 9c e2     - the request id, a random uint32, unique to each request
00 00 00 00     - rpc type (call = 0, response = 1)
00 00 00 02     - rpc version (2)
00 01 87 88     - rpc program (0x00018788 - 100232 - sadmind)
00 00 00 0a     - rpc program version (0x0000000a = 10)
00 00 00 01     - rpc procedure (0x00000001 = 1)
00 00 00 01     - credential flavor (1 = auth\_unix)
00 00 00 20     - length of auth\_unix data (0x20 = 32

## the next 32 bytes are the auth\_unix data
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 182

03-02-2022-24064301-HauptPost-0650

```
40 28 3a 10 - unix timestamp (0x40283a10 = 1076378128 = feb 10 01:55:28 2004 gmt)
00 00 00 0a - length of the client machine name (0x0a = 10)
4d 45 54 41 53 50 4c 4f 49 54 00 00  - metasploit

00 00 00 00 - uid of requesting user (0)
00 00 00 00 - gid of requesting user (0)
00 00 00 00 - extra group ids (0)

00 00 00 00     - verifier flavor (0 = auth\_null, aka none)
00 00 00 00     - length of verifier (0, aka none)
```

The rest of the packet is the request that gets passed to procedure 1 of sadmind.

However, we know the vulnerability is that sadmind trusts the uid coming from the client. sadmind runs any request where the client's uid is 0 as root. As such, we have decoded enough of the request to write our rule.

First, we need to make sure that our packet is an RPC call.

```
content:"|00 00 00 00|"; offset:4; depth:4;
```

Then, we need to make sure that our packet is a call to sadmind.

```
content:"|00 01 87 88|"; offset:12; depth:4;
```

Then, we need to make sure that our packet is a call to the procedure 1, the vulnerable procedure.

```
content:"|00 00 00 01|"; offset:16; depth:4;
```

Then, we need to make sure that our packet has auth_unix credentials.

```
content:"|00 00 00 01|"; offset:20; depth:4;
```

We don't care about the hostname, but we want to skip over it and check a number value after the hostname. This is where byte_test is useful. Starting at the length of the hostname, the data we have is:

```
00 00 00 0a 4d 45 54 41 53 50 4c 4f 49 54 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00
```

We want to read 4 bytes, turn it into a number, and jump that many bytes forward, making sure to account for the padding that RPC requires on strings. If we do that, we are now at:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00
```

which happens to be the exact location of the uid, the value we want to check.

In english, we want to read 4 bytes, 36 bytes from the beginning of the packet, and turn those 4 bytes into an integer and jump that many bytes forward, aligning on the 4 byte boundary. To do that in a Snort rule, we use:

```
byte_jump:4,36,align;
```

then we want to look for the uid of 0.

```
content:"|00 00 00 00|"; within:4;
```

Now that we have all the detection capabilities for our rule, let's put them all together.

```
content:"|00 00 00 00|"; offset:4; depth:4;
content:"|00 01 87 88|"; offset:12; depth:4;
content:"|00 00 00 01|"; offset:16; depth:4;
content:"|00 00 00 01|"; offset:20; depth:4;
byte_jump:4,36,align;
content:"|00 00 00 00|"; within:4;
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 183

16444709564601258531 9

The 3rd and fourth string match are right next to each other, so we should combine those patterns. We end up with:

```
content:"|00 00 00 00|"; offset:4; depth:4;
content:"|00 01 87 88|"; offset:12; depth:4;
content:"|00 00 00 01 00 00 00 01|"; offset:16; depth:8;
byte_jump:4,36,align;
content:"|00 00 00 00|"; within:4;
```

If the sadmind service was vulnerable to a buffer overflow when reading the client's hostname, instead of reading the length of the hostname and jumping that many bytes forward, we would check the length of the hostname to make sure it is not too large.

To do that, we would read 4 bytes, starting 36 bytes into the packet, turn it into a number, and then make sure it is not too large (let's say bigger than 200 bytes). In Snort, we do:

```
byte_test:4,>,200,36;
```

Our full rule would be:

```
content:"|00 00 00 00 "; offset:4; depth:4;
content:"|00 01 87 88|"; offset:12; depth:4;
content:"|00 00 00 01 00 00 00 01|"; offset:16; depth:8;
byte_test:4,>,200,36;
```



16444709564601258531 9

03-02-2022-24064301-Hauptpost-0651

Chapter 4

Dynamic Modules

Preprocessors, detection capabilities, and rules can now be developed as dynamically loadable module to snort. When enabled via the *--enable-dynamicplugin* configure option, the dynamic API presents a means for loading dynamic libraries and allowing the module to utilize certain functions within the main snort code.

The remainder of this chapter will highlight the data structures and API functions used in developing preprocessors, detection engines, and rules as a dynamic plugin to snort.

Beware: the definitions herein may be out of date; check the appropriate header files for the current definitions.

16444709564601258531 9

4.1 Data Structures

A number of data structures are central to the API. The definition of each is defined in the following sections.

4.1.1 DynamicPluginMeta

The *DynamicPluginMeta* structure defines the type of dynamic module (preprocessor, rules, or detection engine), the version information, and path to the shared library. A shared library can implement all three types, but typically is limited to a single functionality such as a preprocessor. It is defined in `sf_dynamic_meta.h` as:

```
#define MAX_NAME_LEN 1024

#define TYPE_ENGINE 0x01
#define TYPE_DETECTION 0x02
#define TYPE_PREPROCESSOR 0x04

typedef struct _DynamicPluginMeta
{
    int type;
    int major;
    int minor;
    int build;
    char uniqueName[MAX_NAME_LEN];
    char *libraryPath;
} DynamicPluginMeta;
```

4.1.2 DynamicPreprocessorData

The *DynamicPreprocessorData* structure defines the interface the preprocessor uses to interact with snort itself. This includes functions to register the preprocessor's configuration parsing, restart, exit, and processing functions. It includes function to log messages, errors, fatal errors, and debugging info. It also includes information for setting alerts, handling Inline drops, access to the StreamAPI, and it provides access to the normalized http and alternate data buffers. This data structure should be initialized when the preprocessor shared library is loaded. It is defined in `sf_dynamic_preprocessor.h`. Check the header file for the current definition.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 185

4.1.3 DynamicEngineData

The *DynamicEngineData* structure defines the interface a detection engine uses to interact with snort itself. This includes functions for logging messages, errors, fatal errors, and debugging info as well as a means to register and check flowbits. It also includes a location to store rule-stubs for dynamic rules that are loaded, and it provides access to the normalized http and alternate data buffers. It is defined in `sf_dynamic_engine.h` as:

```
typedef struct _DynamicEngineData
{
    int version;
    u_int8_t *altBuffer;
    UriInfo *uriBuffers[MAX_URIINFOS];
    RegisterRule ruleRegister;
    RegisterBit flowbitRegister;
    CheckFlowbit flowbitCheck;
    DetectAsn1 asn1Detect;
    LogMsgFunc logMsg;
    LogMsgFunc errMsg;
    LogMsgFunc fatalMsg;
    char *dataDumpDirectory;

    GetPreprocRuleOptFuncs getPreprocOptFuncs;

    SetRuleData setRuleData;
    GetRuleData getRuleData;

    DebugMsgFunc debugMsg;
#ifdef HAVE_WCHAR_H
    DebugWideMsgFunc debugWideMsg;
#endif

    char **debugMsgFile;
    int *debugMsgLine;

    PCRECompileFunc pcreCompile;
    PCREStudyFunc pcreStudy;
    PCREExecFunc pcreExec;

} DynamicEngineData;
```

16444709564601258531 9

4.1.4 SFSnortPacket

The *SFSnortPacket* structure mirrors the snort Packet structure and provides access to all of the data contained in a given packet.

It and the data structures it incorporates are defined in `sf_snort_packet.h`. Additional data structures may be defined to reference other protocol fields. Check the header file for the current definitions.

4.1.5 Dynamic Rules

A dynamic rule should use any of the following data structures. The following structures are defined in `sf_snort_plugin_api.h`.

Rule

The *Rule* structure defines the basic outline of a rule and contains the same set of information that is seen in a text rule. That includes protocol, address and port information and rule information (classification, generator and signature IDs, revision, priority, classification, and a list of references). It also includes a list of rule options and an optional evaluation function.

```
#define RULE_MATCH 1
#define RULE_NOMATCH 0

typedef struct _Rule
{
    IPInfo ip;
    RuleInformation info;

    RuleOption **options; /* NULL terminated array of RuleOption union */

    ruleEvalFunc evalFunc;
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 186

03-02-2022-24064301-Hauptpost-0652

```
    char initialized;     /* Rule Initialized, used internally */
    u_int32_t numOptions; /* Rule option count, used internally */
    char noAlert;         /* Flag with no alert, used internally */
    void *ruleData;    /* Hash table for dynamic data pointers */
} Rule;
```

The rule evaluation function is defined as

```
typedef int (*ruleEvalFunc)(void *);
```

where the parameter is a pointer to the SFSnortPacket structure.

RuleInformation

The *RuleInformation* structure defines the meta data for a rule and includes generator ID, signature ID, revision, classification, priority, message text, and a list of references.

```
typedef struct _RuleInformation
{
    u_int32_t genID;
    u_int32_t sigID;
    u_int32_t revision;
    char     *classification; /* String format of classification name */
    u_int32_t priority;
    char     *message;
    RuleReference **references; /* NULL terminated array of references */
    RuleMetaData **meta; /* NULL terminated array of references */
} RuleInformation;
```

16444709564601258531 9

RuleReference

The *RuleReference* structure defines a single rule reference, including the system name and rereference identifier.

```
typedef struct _RuleReference
{
    char *systemName;
    char *refIdentifier;
} RuleReference;
```

IPInfo

The *IPInfo* structure defines the initial matching criteria for a rule and includes the protocol, src address and port, destination address and port, and direction. Some of the standard strings and variables are predefined - any, HOME_NET, HTTP_SERVERS, HTTP_PORTS, etc.

```
typedef struct _IPInfo
{
    u_int8_t protocol;
    char *   src_addr;
    char *   src_port; /* 0 for non TCP/UDP */
    char     direction;     /* non-zero is bi-directional */
    char *   dst_addr;
    char *   dst_port; /* 0 for non TCP/UDP */
} IPInfo;

#define ANY_NET         "any"
#define HOME_NET        "$HOME_NET"
#define EXTERNAL_NET    "$EXTERNAL_NET"
#define ANY_PORT        "any"
#define HTTP_SERVERS    "$HTTP_SERVERS"
#define HTTP_PORTS      "$HTTP_PORTS"
#define SMTP_SERVERS    "$SMTP_SERVERS"
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 187

RuleOption

The *RuleOption* structure defines a single rule option as an option type and a reference to the data specific to that option. Each option has a flags field that contains specific flags for that option as well as a "Not" flag. The "Not" flag is used to negate the results of evaluating that option.

```
typedef enum DynamicOptionType {
     OPTION_TYPE_PREPROCESSOR,
     OPTION_TYPE_CONTENT,
     OPTION_TYPE_PCRE,
     OPTION_TYPE_FLOWBIT,
     OPTION_TYPE_FLOWFLAGS,
     OPTION_TYPE_ASN1,
     OPTION_TYPE_CURSOR,
     OPTION_TYPE_HDR_CHECK,
     OPTION_TYPE_BYTE_TEST,
     OPTION_TYPE_BYTE_JUMP,
     OPTION_TYPE_BYTE_EXTRACT,
     OPTION_TYPE_SET_CURSOR,
     OPTION_TYPE_LOOP,
     OPTION_TYPE_MAX
};

typedef struct _RuleOption
{
    int optionType;
    union
    {
        void *ptr;
        ContentInfo *content;
        CursorInfo *cursor;
        PCREInfo *pcre;
        FlowBitsInfo *flowBit;
        ByteData *byte;
        ByteExtract *byteExtract;
        FlowFlags *flowFlags;
        Asn1Context *asn1;
        HdrOptCheck *hdrData;
        LoopInfo    *loop;
        PreprocessorOption *preprocOpt;
    } option_u;
} RuleOption;

#define NOT_FLAG                0x10000000
```

Some options also contain information that is initialized at run time, such as the compiled PCRE information, Boyer-Moore content information, the integer ID for a flowbit, etc.

The option types and related structures are listed below.

- OptionType: Content & Structure: *ContentInfo*

  The *ContentInfo* structure defines an option for a content search. It includes the pattern, depth and offset, and flags (one of which must specify the buffer – raw, URI or normalized – to search). Additional flags include nocase, relative, unicode, and a designation that this content is to be used for snorts fast pattern evaluation. The most unique content, that which distinguishes this rule as a possible match to a packet, should be marked for fast pattern evaluation. In the dynamic detection engine provided with Snort, if no *ContentInfo* structure in a given rules uses that flag, the one with the longest content length will be used.

  ```
  typedef struct _ContentInfo
  {
      u_int8_t *pattern;
      u_int32_t depth;
      int32_t  offset;
      u_int32_t flags;        /* must include a CONTENT_BUF_X */
      void     *boyer_ptr;
      u_int8_t *patternByteForm;
      u_int32_t patternByteFormLength;
      u_int32_t incrementLength;
  } ContentInfo;

  #define CONTENT_NOCASE          0x01
  #define CONTENT_RELATIVE        0x02
  #define CONTENT_UNICODE2BYTE    0x04
  ```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 188

16444709564601258531 9

03-02-2022-24064301-Hauptpost-0653

```
#define CONTENT_UNICODE4BYTE    0x08
#define CONTENT_FAST_PATTERN    0x10
#define CONTENT_END_BUFFER      0x20

#define CONTENT_BUF_NORMALIZED  0x100
#define CONTENT_BUF_RAW         0x200
#define CONTENT_BUF_URI         0x400
```

- OptionType: PCRE & Structure: *PCREInfo*

  The *PCREInfo* structure defines an option for a PCRE search. It includes the PCRE expression, pcre_flags such as caseless, as defined in PCRE.h, and flags to specify the buffer.

```
/*
pcre.h provides flags:

PCRE_CASELESS
PCRE_MULTILINE
PCRE_DOTALL
PCRE_EXTENDED
PCRE_ANCHORED
PCRE_DOLLAR_ENDONLY
PCRE_UNGREEDY
*/

typedef struct _PCREInfo
{
    char     *expr;
    void     *compiled_expr;
    void     *compiled_extra;
    u_int32_t compile_flags;
    u_int32_t flags; /* must include a CONTENT_BUF_X */
} PCREInfo;
```

- OptionType: Flowbit & Structure: *FlowBitsInfo*

  The *FlowBitsInfo* structure defines a flowbits option. It includes the name of the flowbit and the operation (set, unset, toggle, isset, isnotset).

```
#define FLOWBIT_SET       0x01
#define FLOWBIT_UNSET     0x02
#define FLOWBIT_TOGGLE    0x04
#define FLOWBIT_ISSET     0x08
#define FLOWBIT_ISNOTSET  0x10
#define FLOWBIT_RESET     0x20
#define FLOWBIT_NOALERT   0x40

typedef struct _FlowBitsInfo
{
    char      *flowBitsName;
    u_int8_t   operation;
    u_int32_t  id;
    u_int32_t  flags;
} FlowBitsInfo;
```

- OptionType: Flow Flags & Structure: *FlowFlags*

  The *FlowFlags* structure defines a flow option. It includes the flags, which specify the direction (from_server, to_server), established session, etc.

```
#define FLOW_ESTABLISHED 0x10
#define FLOW_IGNORE_REASSEMBLED 0x1000
#define FLOW_ONLY_REASSMBLED    0x2000
#define FLOW_FR_SERVER   0x40
#define FLOW_TO_CLIENT   0x40 /* Just for redundancy */
#define FLOW_TO_SERVER   0x80
#define FLOW_FR_CLIENT   0x80 /* Just for redundancy */

typedef struct _FlowFlags
{
    u_int32_t   flags;
} FlowFlags;
```

- OptionType: ASN.1 & Structure: *Asn1Context*

  The *Asn1Context* structure defines the information for an ASN1 option. It mirrors the ASN1 rule option and also includes a flags field.

```
#define ASN1_ABS_OFFSET 1
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 189

16444709564601258531 9

```
#define ASN1_REL_OFFSET 2

typedef struct _Asn1Context
{
    int bs_overflow;
    int double_overflow;
    int print;
    int length;
    unsigned int max_length;
    int offset;
    int offset_type;
    u_int32_t  flags;
} Asn1Context;
```

- OptionType: Cursor Check & Structure: *CursorInfo*

  The *CursorInfo* structure defines an option for a cursor evaluation. The cursor is the current position within the evaluation buffer, as related to content and PCRE searches, as well as byte tests and byte jumps. It includes an offset and flags that specify the buffer. This can be used to verify there is sufficient data to continue evaluation, similar to the isdataat rule option.

```
typedef struct _CursorInfo
{
    int32_t   offset;
    u_int32_t flags;        /* specify one of CONTENT_BUF_X */
} CursorInfo;
```

- OptionType: Protocol Header & Structure: *HdrOptCheck*

  The *HdrOptCheck* structure defines an option to check a protocol header for a specific value. It includes the header field, the operation (<,>,=,etc), a value, a mask to ignore that part of the header field, and flags.

```
#define IP_HDR_ID           0x0001  /* IP Header ID */
#define IP_HDR_PROTO        0x0002  /* IP Protocol */
#define IP_HDR_FRAGBITS     0x0003  /* Frag Flags set in IP Header */
#define IP_HDR_FRAGOFFSET   0x0004  /* Frag Offset set in IP Header */
#define IP_HDR_OPTIONS      0x0005  /* IP Options -- is option xx included */
#define IP_HDR_TTL          0x0006  /* IP Time to live */
#define IP_HDR_TOS          0x0007  /* IP Type of Service */
#define IP_HDR_OPTCHECK_MASK 0x000f

#define TCP_HDR_ACK         0x0010  /* TCP Ack Value */
#define TCP_HDR_SEQ         0x0020  /* TCP Seq Value */
#define TCP_HDR_FLAGS       0x0030  /* Flags set in TCP Header */
#define TCP_HDR_OPTIONS     0x0040  /* TCP Options -- is option xx included */
#define TCP_HDR_WIN         0x0050  /* TCP Window */
#define TCP_HDR_OPTCHECK_MASK 0x00f0

#define ICMP_HDR_CODE       0x1000  /* ICMP Header Code */
#define ICMP_HDR_TYPE       0x2000  /* ICMP Header Type */
#define ICMP_HDR_ID         0x3000  /* ICMP ID for ICMP_ECHO/ICMP_ECHO_REPLY */
#define ICMP_HDR_SEQ        0x4000  /* ICMP ID for ICMP_ECHO/ICMP_ECHO_REPLY */
#define ICMP_HDR_OPTCHECK_MASK 0xf000

typedef struct _HdrOptCheck
{
    u_int16_t hdrField;   /* Field to check */
    u_int32_t op;         /* Type of comparison */
    u_int32_t value;      /* Value to compare value against */
    u_int32_t mask_value; /* bits of value to ignore */
    u_int32_t flags;
} HdrOptCheck;
```

- OptionType: Byte Test & Structure: *ByteData*

  The *ByteData* structure defines the information for both ByteTest and ByteJump operations. It includes the number of bytes, an operation (for ByteTest, <,>,=,etc), a value, an offset, multiplier, and flags. The flags must specify the buffer.

```
#define CHECK_EQ            0
#define CHECK_NEQ           1
#define CHECK_LT            2
#define CHECK_GT            3
#define CHECK_LTE           4
#define CHECK_GTE           5
#define CHECK_AND           6
#define CHECK_XOR           7
#define CHECK_ALL           8
```


Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 190

16444709564601258531 9

```
#define CHECK_ATLEASTONE    9
#define CHECK_NONE          10

typedef struct _ByteData
{
    u_int32_t bytes;      /* Number of bytes to extract */
    u_int32_t op;         /* Type of byte comparison, for checkValue */
    u_int32_t value;      /* Value to compare value against, for checkValue, or extracted value */
    int32_t   offset;     /* Offset from cursor */
    u_int32_t multiplier; /* Used for byte jump -- 32bits is MORE than enough */
    u_int32_t flags;      /* must include a CONTENT_BUF_X */
} ByteData;
```

- OptionType: Byte Jump & Structure: *ByteData*

  See *Byte Test* above.

- OptionType: Set Cursor & Structure: *CursorInfo*

  See *Cursor Check* above.

- OptionType: Loop & Structures: *LoopInfo,ByteExtract,DynamicElement*

  The *LoopInfo* structure defines the information for a set of options that are to be evaluated repeatedly. The loop option acts like a FOR loop and includes start, end, and increment values as well as the comparison operation for termination. It includes a cursor adjust that happens through each iteration of the loop, a reference to a RuleInfo structure that defines the RuleOptions are to be evaluated through each iteration. One of those options may be a ByteExtract.

```
typedef struct _LoopInfo
{
    DynamicElement *start;      /* Starting value of FOR loop (i=start) */
    DynamicElement *end;        /* Ending value of FOR loop (i OP end) */
    DynamicElement *increment;  /* Increment value of FOR loop (i+= increment) */
    u_int32_t op;               /* Type of comparison for loop termination */
    CursorInfo *cursorAdjust;   /* How to move cursor each iteration of loop */
    struct _Rule *subRule;      /* Pointer to SubRule & options to evaluate within
                                 * the loop */
    u_int8_t initialized;       /* Loop initialized properly (safeguard) */
    u_int32_t flags;            /* can be used to negate loop results, specifies                    * relative. */
} LoopInfo;
```

The *ByteExtract* structure defines the information to use when extracting bytes for a DynamicElement used a in Loop evaltion. It includes the number of bytes, an offset, multiplier, flags specifying the buffer, and a reference to the DynamicElement.

```
typedef struct _ByteExtract
{
    u_int32_t bytes;      /* Number of bytes to extract */
    int32_t   offset;     /* Offset from cursor */
    u_int32_t multiplier; /* Multiply value by this (similar to byte jump) */
    u_int32_t flags;      /* must include a CONTENT_BUF_X */
    char *refId;          /* To match up with a DynamicElement refId */
    void *memoryLocation; /* Location to store the data extracted */
} ByteExtract;
```

The *DynamicElement* structure is used to define the values for a looping evaluation. It includes whether the element is static (an integer) or dynamic (extracted from a buffer in the packet) and the value. For a dynamic element, the value is filled by a related ByteExtract option that is part of the loop.

```
#define DYNAMIC_TYPE_INT_STATIC 1
#define DYNAMIC_TYPE_INT_REF    2

typedef struct _DynamicElement
{
    char dynamicType;           /* type of this field - static or reference */
    char *refId;                /* reference ID (NULL if static) */
    union
    {
        void *voidPtr;          /* Holder */
        int32_t staticInt;        /* Value of static */
        int32_t *dynamicInt;  /* Pointer to value of dynamic */
    } data;
} DynamicElement;
```

4.2 Required Functions

Each dynamic module must define a set of functions and data objects to work within this framework.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 191

16444709564601258531 9

4.2.1 Preprocessors

Each dynamic preprocessor library must define the following functions. These are defined in the file sf_dynamic_preproc_lib.c. The metadata and setup function for the preprocessor should be defined sf_preproc_info.h.

- *int LibVersion(DynamicPluginMeta *)*
  This function returns the metadata for the shared library.
- *int InitializePreprocessor(DynamicPreprocessorData *)*
  This function initializes the data structure for use by the preprocessor into a library global variable, _dpd and invokes the setup function.

4.2.2 Detection Engine

Each dynamic detection engine library must define the following functions.

- *int LibVersion(DynamicPluginMeta *)*
  This function returns the metadata for the shared library.
- *int InitializeEngineLib(DynamicEngineData *)*
  This function initializes the data structure for use by the engine.

The sample code provided with Snort predefines those functions and defines the following APIs to be used by a dynamic rules library.



- *int RegisterRules(Rule **)*
  This is the function to iterate through each rule in the list, initialize it to setup content searches, PCRE evalution data, and register flowbits.
- *int DumpRules(char *,Rule **)*
  This is the function to iterate through each rule in the list and write a rule-stop to be used by snort to control the action of the rule (alert, log, drop, etc).
- *int ruleMatch(void *p, Rule *rule)*
  This is the function to evaluate a rule if the rule does not have its own Rule Evaluation Function. This uses the individual functions outlined below for each of the rule options and handles repetitive content issues.
  Each of the functions below returns RULE_MATCH if the option matches based on the current criteria (cursor position, etc).
  - *int contentMatch(void *p, ContentInfo* content, u_int8_t **cursor)*
    This function evaluates a single content for a given packet, checking for the existence of that content as delimited by ContentInfo and cursor. Cursor position is updated and returned in *cursor.
    With a text rule, the with option corresponds to depth, and the distance option corresponds to offset.
  - *int checkFlow(void *p, FlowFlags *flowflags)*
    This function evaluates the flow for a given packet.
  - *int extractValue(void *p, ByteExtract *byteExtract, u_int8_t *cursor)*
    This function extracts the bytes from a given packet, as specified by ByteExtract and delimited by cursor. Value extracted is stored in ByteExtract memoryLocation parameter.
  - *int processFlowbits(void *p, FlowBitsInfo *flowbits)*
    This function evaluates the flowbits for a given packet, as specified by FlowBitsInfo. It will interact with flowbits used by text-based rules.
  - *int setCursor(void *p, CursorInfo *cursorInfo, u_int8_t **cursor)*
    This function adjusts the cursor as delimited by CursorInfo. New cursor position is returned in *cursor. It handles bounds checking for the specified buffer and returns RULE_NOMATCH if the cursor is moved out of bounds.
    It is also used by contentMatch, byteJump, and pcreMatch to adjust the cursor position after a successful match.
  - *int checkCursor(void *p, CursorInfo *cursorInfo, u_int8_t *cursor)*
    This function validates that the cursor is within bounds of the specified buffer.
  - *int checkValue(void *p, ByteData *byteData, u_int32_t value, u_int8_t *cursor)*
    This function compares the *value* to the value stored in ByteData.
  - *int byteTest(void *p, ByteData *byteData, u_int8_t *cursor)*
    This is a wrapper for extractValue() followed by checkValue().
  - *int byteJump(void *p, ByteData *byteData, u_int8_t **cursor)*
    This is a wrapper for extractValue() followed by setCursor().
  - *int pcreMatch(void *p, PCREInfo *pcre, u_int8_t **cursor)*
    This function evaluates a single pcre for a given packet, checking for the existence of the expression as delimited by PCREInfo and cursor. Cursor position is updated and returned in *cursor.
  - *int detectAsn1(void *p, Asn1Context *asn1, u_int8_t *cursor)*
    This function evaluates an ASN.1 check for a given packet, as delimited by Asn1Context and cursor.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 192

03-02-2022-24064301-Hauptpost-0655

- *int checkHdrOpt(void *p, HdrOptCheck *optData)*
  This function evaluates the given packet's protocol headers, as specified by HdrOptCheck.
- *int loopEval(void *p, LoopInfo *loop, u_int8_t **cursor)*
  This function iterates through the SubRule of LoopInfo, as delimited by LoopInfo and cursor. Cursor position is updated and returned in *cursor.
- *int preprocOptionEval(void *p, PreprocessorOption *preprocOpt, u_int8_t **cursor)*
  This function evaluates the preprocessor defined option, as spepcifed by PreprocessorOption. Cursor position is updated and returned in *cursor.
- *void setTempCursor(u_int8_t **temp_cursor, u_int8_t **cursor)*
  This function is used to handled repetitive contents to save off a cursor position temporarily to be reset at later point.
- *void revertTempCursor(u_int8_t **temp_cursor, u_int8_t **cursor)*
  This function is used to revert to a previously saved temporary cursor position.

> **NOTE**
> If you decide to write you own rule evaluation function, patterns that occur more than once may result in false negatives. Take extra care to handle this situation and search for the matched pattern again if subsequent rule options fail to match. This should be done for both content and PCRE options.

4.2.3 Rules

Each dynamic rules library must define the following functions. Examples are defined in the file `sfnort_dynamic_detection_lib.c`. The metadata and setup function for the preprocessor should be defined in `sfsnort_dynamic_detection_lib.h`.

- *int LibVersion(DynamicPluginMeta *)*
  This function returns the metadata for the shared library.
- *int EngineVersion(DynamicPluginMeta *)*
  This function defines the version requirements for the corresponding detection engine library.
- *int DumpSkeletonRules()*
  This functions writes out the rule-stubs for rules that are loaded.
- *int InitializeDetection()*
  This function registers each rule in the rules library. It should set up fast pattern-matcher content, register flowbits, etc.

The sample code provided with Snort predefines those functions and uses the following data within the dynamic rules library.

- *Rule *rules[]*
  A NULL terminated list of Rule structures that this library defines.

4.3 Examples

This section provides a simple example of a dynamic preprocessor and a dynamic rule.

4.3.1 Preprocessor Example

The following is an example of a simple preprocessor. This preprocessor always alerts on a Packet if the TCP port matches the one configured.

This assumes the the files *sf_dynamic_preproc_lib.c* and *sf_dynamic_preproc_lib.h* are used.

This is the metadata for this preprocessor, defined in *sf_preproc_info.h*.

```
#define MAJOR_VERSION 1
#define MINOR_VERSION 0
#define BUILD_VERSION 0
#define PREPROC_NAME    "SF_Dynamic_Example_Preprocessor"

#define DYNAMIC_PREPROC_SETUP   ExampleSetup
extern void ExampleSetup();
```

The remainder of the code is defined in *spp_example.c* and is compiled together with *sf_dynamic_preproc_lib.c* into lib_sfdynamic_preprocessor_example.so.

Define the Setup function to register the initialization function.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 193

16444709564601258531 9

```
#define GENERATOR_EXAMPLE 256
extern DynamicPreprocessorData _dpd;

void ExampleInit(unsigned char *);
void ExampleProcess(void *, void *);

void ExampleSetup()
{
    _dpd.registerPreproc("dynamic_example", ExampleInit);

    DEBUG_WRAP(_dpd.debugMsg(DEBUG_PLUGIN, "Preprocessor: Example is setup\n");
}
```

The initialization function to parse the keywords from snort.conf.

```
u_int16_t portToCheck;

void ExampleInit(unsigned char *args)
{
    char *arg;
    char *argEnd;
    unsigned long port;

    _dpd.logMsg("Example dynamic preprocessor configuration\n");

    arg = strtok(args, " \t\n\r");

    if(!strcasecmp("port", arg))
    {
        arg = strtok(NULL, "\t\n\r");
        if (!arg)
        {
            _dpd.fatalMsg("ExamplePreproc: Missing port\n");
        }

        port = strtoul(arg, &argEnd, 10);
        if (port < 0 || port > 65535)
        {
            _dpd.fatalMsg("ExamplePreproc: Invalid port %d\n", port);
        }
        portToCheck = port;

        _dpd.logMsg("    Port: %d\n", portToCheck);
    }
    else
    {
        _dpd.fatalMsg("ExamplePreproc: Invalid option %s\n", arg);
    }

    /* Register the preprocessor function, Transport layer, ID 10000 */
    _dpd.addPreproc(ExampleProcess, PRIORITY_TRANSPORT, 10000);

    DEBUG_WRAP(_dpd.debugMsg(DEBUG_PLUGIN, "Preprocessor: Example is initialized\n");
}
```

The function to process the packet and log an alert if the either port matches.

```
#define SRC_PORT_MATCH  1
#define SRC_PORT_MATCH_STR "example_preprocessor: src port match"
#define DST_PORT_MATCH  2
#define DST_PORT_MATCH_STR "example_preprocessor: dest port match"
void ExampleProcess(void *pkt, void *context)
{
    SFSnortPacket *p = (SFSnortPacket *)pkt;
    if (!p->ip4_header || p->ip4_header->proto != IPPROTO_TCP || !p->tcp_header)
    {
        /* Not for me, return */
        return;
    }
```


1644709564601258531 9

03-02-2022-24064301-HauptPost-0656

```
    if (p->src_port == portToCheck)
    {
        /* Source port matched, log alert */
        _dpd.alertAdd(GENERATOR_EXAMPLE, SRC_PORT_MATCH,
                      1, 0, 3, SRC_PORT_MATCH_STR, 0);
        return;
    }

    if (p->dst_port == portToCheck)
    {
        /* Destination port matched, log alert */
        _dpd.alertAdd(GENERATOR_EXAMPLE, DST_PORT_MATCH,
                      1, 0, 3, DST_PORT_MATCH_STR, 0);
        return;
    }
}
```

4.3.2 Rules

The following is an example of a simple rule, take from the current rule set, SID 109. It is implemented to work with the detection engine provided with snort.

The snort rule in normal format:

```
alert tcp $HOME_NET 12345:12346 -> $EXTERNAL_NET any \
(msg:"BACKDOOR netbus active"; flow:from_server,established; \
content:"NetBus"; reference:arachnids,401; classtype:misc-activity; \
sid:109; rev:5;)
```

This is the metadata for this rule library, defined in *detection_lib_meta.h*.

```
/* Version for this rule library */
#define DETECTION_LIB_MAJOR_VERSION 1
#define DETECTION_LIB_MINOR_VERSION 0
#define DETECTION_LIB_BUILD_VERSION 1
#define DETECTION_LIB_NAME "Snort_Dynamic_Rule_Example"

/* Required version and name of the engine */
#define REQ_ENGINE_LIB_MAJOR_VERSION 1
#define REQ_ENGINE_LIB_MINOR_VERSION 0
#define REQ_ENGINE_LIB_NAME "SF_SNORT_DETECTION_ENGINE"
```

The definition of each data structure for this rule is in *sid109.c*.

Declaration of the data structures.

- Flow option

  Define the *FlowFlags* structure and its corresponding *RuleOption*. Per the text version, flow is from_server,established.

  ```
  static FlowFlags sid109flow =
  {
      FLOW_ESTABLISHED|FLOW_TO_CLIENT
  };

  static RuleOption sid109option1 =
  {
      OPTION_TYPE_FLOWFLAGS,
      {
          &sid109flow
      }
  };
  ```

- Content Option

  Define the *ContentInfo* structure and its corresponding *RuleOption*. Per the text version, content is "NetBus", no depth or offset, case sensitive, and non-relative. Search on the normalized buffer by default. **NOTE**: This content will be used for the fast pattern matcher since it is the longest content option for this rule and no contents have a flag of *CONTENT_FAST_PATTERN*.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 195

1344470956460125853 19

```
static ContentInfo sid109content =
{
    "NetBus",               /* pattern to search for */
    0,                      /* depth */
    0,                      /* offset */
    CONTENT_BUF_NORMALIZED, /* flags */
    NULL,                   /* holder for boyer/moore info */
    NULL,                   /* holder for byte representation of "NetBus" */
    0,                      /* holder for length of byte representation */
    0                       /* holder for increment length */
};

static RuleOption sid109option2 =
{
    OPTION_TYPE_CONTENT,
    {
        &sid109content
    }
};
```

- Rule and Meta Data

Define the references.

```
static RuleReference sid109ref_arachnids =
{
    "arachnids",    /* Type */
    "401"           /* value */
};

static RuleReference *sid109refs[] =
{
    &sid109ref_arachnids,
    NULL
};
```

The list of rule options. Rule options are evaluated in the order specified.

```
RuleOption *sid109options[] =
{
    &sid109option1,
    &sid109option2,
    NULL
};
```

The rule itself, with the protocol header, meta data (sid, classification, message, etc).

```
Rule sid109 =
{
    /* protocol header, akin to => tcp any any -> any any               */
    {
        IPPROTO_TCP,        /* proto */
        HOME_NET,           /* source IP     */
        "12345:12346",      /* source port(s) */
        0,                  /* Direction */
        EXTERNAL_NET,       /* destination IP     */
        ANY_PORT,           /* destination port   */
    },
    /* metadata */
    {
        3,                  /* genid -- use 3 to distinguish a C rule */
        109,                /* sigid */
        5,                  /* revision */
        "misc-activity",    /* classification */
        0,                  /* priority */
        "BACKDOOR netbus active",     /* message */
        sid109refs          /* ptr to references */
    },
    sid109options,          /* ptr to rule options */
    NULL,                   /* Use internal eval func */
    0,                      /* Holder, not yet initialized, used internally */
    0,                      /* Holder, option count, used internally */
    0,                      /* Holder, no alert, used internally for flowbits */
    NULL                    /* Holder, rule data, used internally */
```


164447095646012585319

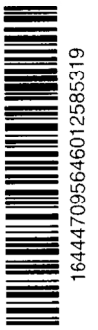
Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 196

03-02-2022-24064301-Hauptpost-0657

- The List of rules defined by this rules library
  The NULL terminated list of rules. The InitializeDetection iterates through each Rule in the list and initializes the content, flowbits, pcre, etc.

```
extern Rule sid109;
extern Rule sid637;

Rule *rules[] =
{
    &sid109,
    &sid637,
    NULL
};
```



Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 197

03-02-2022-24064301-Hauptpost-0657-R

Chapter 5

Snort Development

Currently, this chapter is here as a place holder. It will someday contain references on how to create new detection plugins and preprocessors. End users don't really need to be reading this section. This is intended to help developers get a basic understanding of whats going on quickly.

If you are going to be helping out with Snort development, please use the HEAD branch of cvs. We've had problems in the past of people submitting patches only to the stable branch (since they are likely writing this stuff for their own IDS purposes). Bug fixes are what goes into STABLE. Features go into HEAD.

1644709564012585319

5.1 Submitting Patches

Patches to Snort should be sent to the `snort-devel@lists.sourceforge.net` mailing list. Patches should done with the command `diff -nu snort-orig snort-new`

5.2 Snort Data Flow

First, traffic is acquired from the network link via libpcap. Packets are passed through a series of decoder routines that first fill out the packet structure for link level protocols then are further decoded for things like TCP and UDP ports.

Packets are then sent through the registered set of preprocessors. Each preprocessor checks to see if this packet is something it should look at.

Packets are then sent through the detection engine. The detection engine checks each packet against the various options listed in the Snort config files. Each of the keyword options is a plugin. This allows this to be easily extensible.

5.2.1 Preprocessors

For example, a TCP analysis preprocessor could simply return if the packet does not have a TCP header. It can do this by checking:

```
if (p->tcph==null)
   return;
```

Similarly, there are a lot of packet_flags available that can be used to mark a packet as "reassembled" or logged. Check out src/decode.h for the list of pkt_* constants.

5.2.2 Detection Plugins

Basically, look at an existing output plugin and copy it to a new item and change a few things. Later, we'll document what these few things are.

5.2.3 Output Plugins

Generally, new output plugins should go into the barnyard project rather than the Snort project. We are currently cleaning house on the available output options.

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 198

03-02-2022-24064301-Hauptpost-0658
03-02-2022-24064301-HauptPost-0658

5.3 The Snort Team

Creator and Lead Architect	Marty Roesch
Lead Snort Developers	Steve Sturges Bhagyashree Bantwal Hui Cao Russ Combs Ryan Jordan
Snort Rules Team	Matt Watchinski Sojeong Hong Nigel Houghton Richard Johnson Alex Kambis Alex Kirk Chris Marshall Kevin Miklavcic Patrick Mullen Matt Olney Ryan Pentney Alain Zidoemba
Win32 Maintainer	Snort Team
Major Contributors	Erek Adams Andrew Baker Scott Campbell Brian Caswell Dilbagh Chahal Roman D. Michael Davis Chris Green Lurene Grenier Jed Haile Jeremy Hewlett Victor Julien Glenn Mansfield Keeni Adam Keeton Chad Kreimendahl Kevin Liu Rob McMillen William Metcalf Andrew Mullican Jeff Nathan Marc Norton Judy Novak Andreas Ostling Chris Reid Daniel Roelker Dragos Ruiu JP Vossen Todd Wease Daniel Wittenberg Phil Wood Fyodor Yarochkin



Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 199

03-02-2022-24064301-Hauptpost-0658-R

Bibliography

[1] http://packetstorm.securify.com/mag/phrack/phrack49/p49-06

[2] http://www.nmap.org

[3] http://public.pacbell.net/dedicated/cidr.html

[4] http://www.whitehats.com

[5] http://www.incident.org/snortdb

[6] http://www.pcre.org



1644470956460125853 19

Patent Owner, Centripetal Networks, LLC - Ex. 2014, p. 200