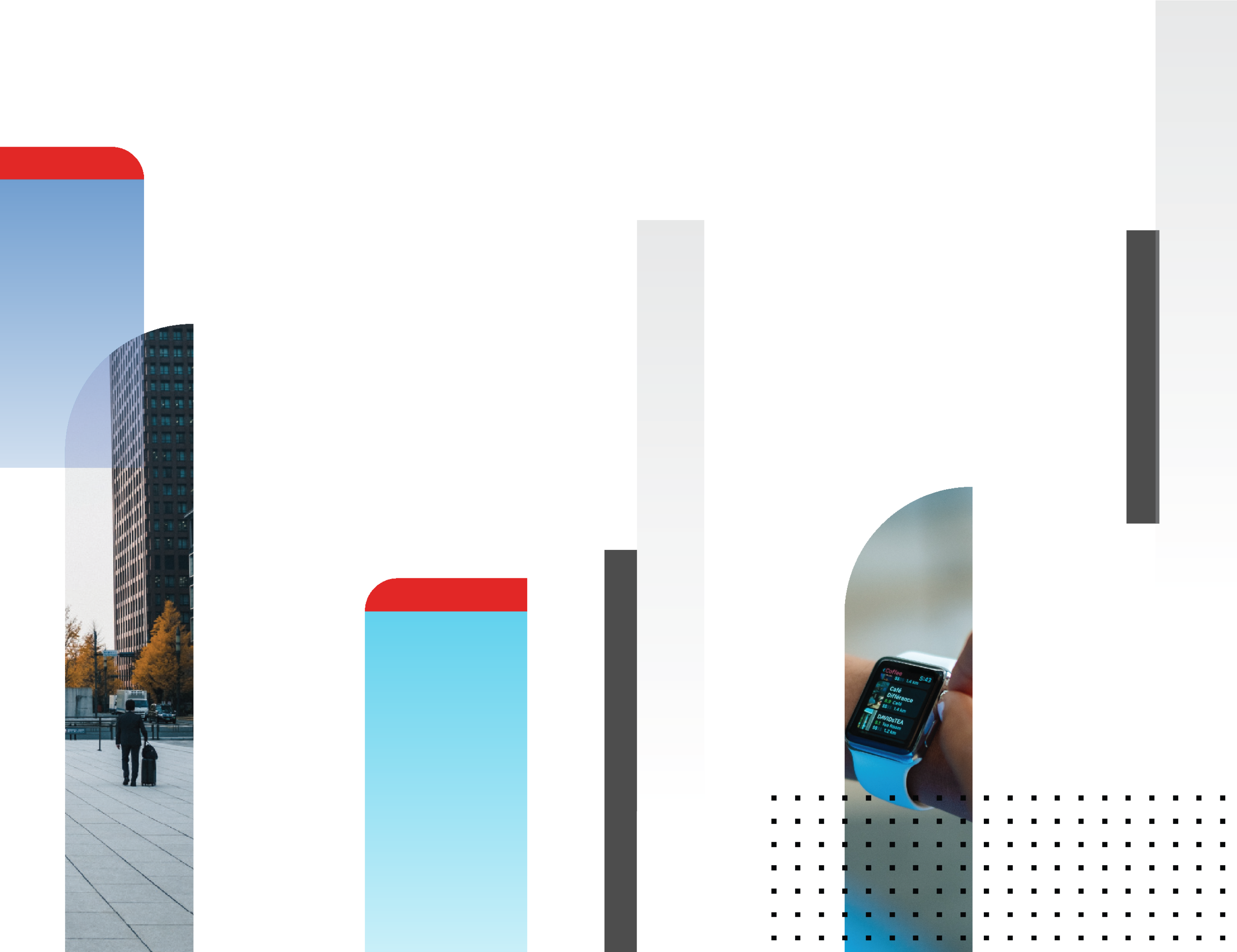
Syntax Guide

Custom IPS and Application Control Signature 7.0



FORTINET DOCUMENT LIBRARY
https://docs.fortinet.com

FORTINET VIDEO GUIDE
https://video.fortinet.com

FORTINET BLOG
https://blog.fortinet.com

CUSTOMER SERVICE & SUPPORT
https://support.fortinet.com

FORTINET TRAINING & CERTIFICATION PROGRAM
https://www.fortinet.com/training-certification

NSE INSTITUTE
https://training.fortinet.com

FORTIGUARD CENTER
https://www.fortiguard.com

END USER LICENSE AGREEMENT
https://www.fortinet.com/doc/legal/EULA.pdf

FEEDBACK
Email: techdoc@fortinet.com

July 07, 2022
Custom IPS and Application Control Signature 7.0 Syntax Guide
43-70-824408-20220707

TABLE OF CONTENTS

Change Log

Date	Change Description
2022-07-07	Initial release.

Creating IPS and application control signatures

IPS and application control signatures allow you to identify packet types as they pass through your FortiGate. After you create a signature that identifies a certain packet type, you add the signature to an IPS or application control sensor. Within the sensor you specify the action to apply to packets that match the signature: block, monitor, allow, or quarantine. You then add the sensor to a firewall policy. When the firewall policy accepts a packet that matches your custom signature, the FortiGate takes the specified action with the packet.

IPS signatures employ a lightweight signature definition language to identify packets. All signatures include a type header (F-SBID) and a series of option/value pairs. You use the option/value pairs to uniquely identify a packet. Each option starts with -- followed by the option name, a space, and usually an option value. Option names are case-insensitive and some options do not need a value. Custom signatures can be up to 4095 characters long.

Custom signature syntax:

```
F-SBID( --<option1> [<value1>]; --<option2> [<value2>];...)
```

IPS signatures include the following option types:

- Protocol: options to inspect IP/ICMP/UDP/TCP protocol headers for the value paired with the option.
- Payload: options to inspect the packet payload for the value paired with the option.
- Special: options to inspect other aspects (such as application control) of the packet for the value paired with the option.
- Application options on page 48: options to inspect other aspects unique to application control for the value paired with the option.

Signature definition notes

- We recommend you use lower case, although keywords in a signature are not case-sensitive.
- To match patterns using `--pattern`, you must enclose the pattern in double quotation marks (") and follow it with a semicolon. The special characters (" ; \ | :) must be written as (`|22|, |3B|` or `|3b|, |5C|` or `|5c|, |7C|` or `|7c|, |3A|` or `|3a|)`. Although you can use backslash (\) to escape any character except a semicolon (;), we do not recommend this.
- To match patterns using `--pcre`, you must enclose the pattern in double quotation marks (") and follow it with a semicolon (;). The special characters (" ; /) must be written as (`\x22, \x3B` or `\x3b, \x2F` or `\x2f`). Regular expressions should conform to the Perl Compatible Regular Expression (PCRE) standard. See pcre on page 31 for syntax details.
- If some encoded content is always the same, you can make a signature to match the encoded form. This allows for detection of the encoded content, even though the engine does not support decoding.
- Do not use the `no_case` option on a non-alphabetic pattern.
- Do not use the `no_case` option on case-sensitive patterns.

Range modifier notes

- The Snort/PCRE `R` option is no longer part of our PCRE. Use `--distance 0;` instead.
- If you do not use a `range` modifier with `pattern` or `pcre`, matching is done from the beginning to the end of the buffer.
- If you only use `distance` or `distance_abs` with `pattern` or `pcre`, matching is done from the location that is relative to the reference specified by `<refer>` to the end of the buffer.

Basic options

Attack IDs may be a required option if you are using FortiManager. If you are configuring a customer signature directly on the FortiGate, FortiOS automatically generates attack IDs if you do not provide them. For the following FortiManager versions, FortiManager does not automatically generate attack IDs, so you must define attack IDs:

- 6.0.0 to 6.0.6
- 6.2.0 to 6.2.2

name

Syntax:

```
--name <"string">;
```

The `name` keyword provides a signature name that is displayed in the GUI and the CLI. This is an optional field. The name should only contain printable characters.

- The string should be enclosed by double quotation marks.
- The maximum length of a signature name is 64 characters.
- The period replaces the use of a space.
- The signature name must be unique for each custom signature.

Example:

```
--name "IBM.Domino.iNotes.Foldername.Buffer.Overflow";
```

service

Use the `service` keyword to specify the session type associated with a packet. In order for this keyword to work, the session that is being identified should be supported by a suitable dissector. To see a list of services currently supported by the IPS engine dissectors, refer to the table, Supported service types. You can use the `service` keyword once in a signature.

Syntax:

```
--service <service_name>;
```

Examples:

```
--service HTTP;
```

```
--service DNS;
```

Supported service types

Session Type	Criterion	Service Option
Back_office (bo, bo2k)	TCP/UDP, any port	service BO
COTP	TCP, 102	service COTP
DCE RPC	TCP/UDP, any port	service DCERPC
DHCP	UDP, any port	service DHCP
DNP3	TCP, any port	service DNP3
DNS	TCP/UDP, 53	service DNS
FTP	TCP, any port	service FTP
H323	TCP, 1720	service H323
HTTP	TCP, any port	service HTTP
IEC104	TCP, 2024	service IEC104
IM (yahoo, msn, aim, qq)	TCP/UDP, any port	service IM
IMAP	TCP, any port	service IMAP
LDAP	TCP, 389	service LDAP
MODBUS	TCP, 502	service MODBUS
MSSQL	TCP, 1433	service MSSQL
NBSS	TCP, 139, 445	service NBSS
NNTP	TCP, any port	service NNTP
P2P (skype, BT, eDonkey, kazaz, gnutella, dc++)	TCP/UDP, any port	service P2P
POP3	TCP, any port	service POP3
RADIUS	UDP, 1812, 1813	service RADIUS
RDT	TCP, any port, by RTSP	service RDT
RTCP	TCP, any port, by RTSP	service RTCP
RTP	TCP, any port, by RTSP	service RTP
RTSP	TCP, any port	service RTSP
SCCP (skinny)	TCP, 2000	service SCCP
SIP	TCP/UDP any port	service SIP
SMTP	TCP, any port	service SMTP
SNMP	UDP, 161, 162	service SNMP
SSH	TCP, any port	service SSH

Session Type	Criterion	Service Option
SSL	TCP, any port	service SSL
SUN RPC	TCP/UDP, 111, 32771	service RPC
TELNET	TCP, 23	service TELNET
TFN	ICMP, any port	service TFN
TFTP	UDP, any port	service TFTP
WebSocket	TCP, any port	service websocket

protocol

The protocol keyword specifies the type of protocol that is associated with the signature.

In IPv4 [RFC791] there is a field called “Protocol” to identify the next level protocol. This is an 8-bit field. In IPv6 [RFC2460], this field is called the “Next Header” field. This is an optional field.

Besides ICMP, TCP, and UDP, you can also specify protocols by their protocol numbers.

Syntax:

```
    --protocol [icmp | tcp | udp | <number>];
```

Tests are available to check the properties of the header:

- IP header options on page 17
- TCP header options on page 21
- UDP header options on page 25
- ICMP header options on page 27

Protocol numbers

#	Protocol	Protocol full name
0	HOPOPT	IPv6 Hop-by-Hop Option
1	ICMP	Internet Control Message Protocol
2	IGMP	Internet Group Management
3	GGP	Gateway-to-Gateway
4	IPv4	IPv4 encapsulation Protocol
5	ST	Stream
6	TCP	Transmission Control Protocol
7	CBT	CBT

#	Protocol	Protocol full name
8	EGP	Exterior Gateway Protocol
9	IGP	Any private interior gateway (used by Cisco for their IGRP)
10	BBN-RCC-MON	BBN RCC Monitoring
11	NVP-II	Network Voice Protocol
12	PUP	PUP
13	ARGUS	ARGUS
14	EMCON	EMCON
15	XNET	Cross Net Debugger
16	CHAOS	Chaos
17	UDP	User Datagram Protocol
18	MUX	Multiplexing
19	DCN-MEAS	DCN Measurement Subsystems
20	HMP	Host Monitoring
21	PRM	Packet Radio Measurement
22	XNS-IDP	XEROX NS IDP
23	TRUNK-1	Trunk-1
24	TRUNK-2	Trunk-2
25	LEAF-1	Leaf-1
26	LEAF-2	Leaf-2
27	RDP	Reliable Data Protocol
28	IRTP	Internet Reliable Transaction
29	ISO-TP4	ISO Transport Protocol Class 4
30	NETBLT	Bulk Data Transfer Protocol
31	MFE-NSP	MFE Network Services Protocol
32	MERIT-INP	MERIT Internodal Protocol
33	DCCP	Datagram Congestion Control Protocol
34	3PC	Third Party Connect Protocol
35	IDPR	Inter-Domain Policy Routing Protocol
36	XTP	XTP
37	DDP	Datagram Delivery Protocol

#	Protocol	Protocol full name
38	IDPR-CMTP	IDPR Control Message Transport Proto
39	TP++	TP++ Transport Protocol
40	IL	IL Transport Protocol
41	IPv6	IPv6 encapsulation
42	IPv6	SDRPSource Demand Routing Protocol
43	IPv6-Route	Routing Header for IPv6
44	IPv6-Frag	Fragment Header for IPv6
45	IDRP	Inter-Domain Routing Protocol
46	RSVP	Reservation Protocol
47	GRE	General Routing Encapsulation
48	DSR	Dynamic Source Routing Protocol
49	BNA	BNA
50	ESP	Encap Security Payload
51	AH	Authentication Header
52	I-NLSP	Integrated Net Layer Security TUBA
53	SWIPE	IP with Encryption
54	NARP	NBMA Address Resolution Protocol
55	MOBILE	IP Mobility
56	TLSP	Transport Layer Security Protocol using Kryptonet key management
57	SKIP	SKIP
58	IPv6-ICMP	ICMP for IPv6
59	IPv6-NoNxt	No Next Header for IPv6
60	IPv6-Opts	Destination Options for IPv6
61		any host internal protocol
62	CFTP	CFTP
63		any local network
64	SAT-EXPAK	SATNET and Backroom EXPAK
65	KRYPTOLAN	Kryptolan
66	RVD	MIT Remote Virtual Disk Protocol
67	IPPC	Internet Pluribus Packet Core

#	Protocol	Protocol full name
68		any distributed file system
69	SAT-MON	SATNET Monitoring
70	VISA	VISA Protocol
71	IPCV	Internet Packet Core Utility
72	CPNX	Computer Protocol Network Executive
73	CPHB	Computer Protocol Heart Beat
74	WSN	Wang Span Network
75	PVP	Packet Video Protocol
76	BR-SAT-MON	Backroom SATNET Monitoring
77	SUN-ND	SUN ND PROTOCOL-Temporary
78	WB-MON	WIDEBAND Monitoring
79	WB-EXPAK	WIDEBAND EXPAK
80	ISO-IP	ISO Internet Protocol
81	VMTP	VMTP
82	SECURE-VMTP	SECURE-VMTP
83	VINES	VINES
84	TTP	TTP
84	IPTM	Protocol Internet Protocol Traffic
85	NSFNET-IGP	NSFNET-IGP
86	DGP	Dissimilar Gateway Protocol
87	TCF	TCF
88	EIGRP	EIGRP
89	OSPFIGP	OSPFIGP
90	Sprite-RPC	Sprite RPC Protocol
91	LARP	Locus Address Resolution Protocol
92	MTP	Multicast Transport Protocol
93	AX.25	AX.25 Frames
94	IPIP	IP-within-IP Encapsulation Protocol
95	MICP	Mobile Internetworking Control Pro.
96	SCC-SP	Semaphore Communications Sec. Pro.

#	Protocol	Protocol full name
97	ETHERIP	Ethernet-within-IP Encapsulation
98	ENCAP	Encapsulation Header
99		any private encryption scheme
100	GMTP	GMTP
101	IFMP	Ipsilon Flow Management Protocol
102	PNNI	PNNI over IP
103	PIM	Protocol Independent Multicast
104	ARIS	ARIS
105	SCPS	SCPS
106	QNX	QNX
107	A/N	Active Networks
108	IPComp	IP Payload Compression Protocol
109	SNP	Sitara Networks Protocol
110	Compaq-Peer	Compaq Peer Protocol
111	IPX-in-IP	IPX in IP
112	VRRP	Virtual Router Redundancy Protocol
113	PGM	PGM Reliable Transport Protocol
114		any 0-hop protocol
115	L2TP	Layer Two Tunneling Protocol
116	DDX	D-II Data Exchange (DDX)
117	IATP	Interactive Agent Transfer Protocol
118	STP	Schedule Transfer Protocol
119	SRP	SpectraLink Radio Protocol
120	UTI	UTI
121	SMP	Simple Message Protocol
122	SM	SM
123	PTP	Performance Transparency Protocol
124	ISIS over IPv4	
125	FIRE	
126	CRTP	Combat Radio Transport Protocol

#	Protocol	Protocol full name
127	CRUDP	Combat Radio User Datagram
128	SSCOPMCE	
129	IPLT	
119	SRP	SpectraLink Radio Protocol
120	UTI	UTI
121	SMP	Simple Message Protocol
122	SM	SM
130	SPS	Secure Packet Shield
131	PIPE	Private IP Encapsulation within IP
132	SCTP	Stream Control Transmission Protocol
133	FC	Fibre Channel
134	RSVP-E2E-IGNORE	
135	Mobility Header	
136	UDPLite	
137	MPLS-in-IP	
138	manet	
139	HIP	
140	Shim6	
141	WESP	
142	ROHC	
143 to 252	Unassigned	Unassigned
253		Use for experimentation and testing
254		Use for experimentation and testing
255	Reserved	

severity

The `severity` keyword is used to specify the severity of the vulnerability that a signature covers. It is optional. The severity keyword allows the IPS ngine to inspect different sets of attacks in different firewall profiles. You can enable a subset of severity levels in a firewall profile so that a packet associated with the profile is only checked by signatures of the selected severity levels. This ability provides the user with another dimension of control over IPS Engine performance and signature false positive rates. The severity keyword accepts one of the following values: critical, high, medium, low, and info. The default severity is “critical”.

Syntax:

```
--severity <severity level>;
```

Examples:

```
--severity medium;
```

description

The `description` keyword is used to give a text description of a signature. It is optional. The description should be enclosed in double quotation marks.

Syntax:

```
--description “text string”;
```

Examples:

```
--description “Overlong Chunk Size”;
```

Protocol options

IP header options

Use IP header options to check the properties of the IP header.

ip_id

Check the IP ID field for a specific value.

Syntax:

```
--ip_id <number>;
```

Example:

```
--ip_id 32212;
```

ip_tos

Check the IP TOS field for a specific value.

Syntax:

```
--ip_tos <number>;
```

Example:

```
--ip_tos 4;
```

ip_ttl

Check the IP time-to-live field value.

Syntax:

```
--ip_ttl <number>;
--ip_ttl ><number>;
--ip_ttl <<number>;
```

Example:

```
--ip_ttl <4;
```

ip_option

Check the IP options.

Syntax:

```
--ip_option <option>;
```

The following values can be tested:

<option>	Description
rr	Record route
eol	End of list
nop	No operation
ts	Internet timestamp
sec	Security
lsrr	Loose source routing
lsrre	Loose source routing for MS99-038 and CVE 199-0909
ssrr	Strict source routing
satid	Stream ID

Example:

```
--ip_option ts;
```

same_ip

Check whether `src_addr` is the same as `dst_addr`. No value required for this option.

Example:

```
--same_ip;
```

src_addr

Check the source IP address.

Syntax:

```
--src_addr <IP address>;
```

The IP address can be in the following formats:

- `x.y.z.u`
- `x.y.z.u/n`

- `x.y.z.u:n`
- `ab:cd:ef:gh:ij:kl:mn:op`
- `ab:cd:ef::mn:op`

The prefix `!` means exclude the addresses. Multiple addresses should be between square brackets `[ ]`, separated by commas.

Examples:

```
--src_addr !10.10.10.1;
```

```
--src_addr 10.10.10.0:24;
```

```
--src_addr fde0:6477:1e3f::1:b9;
```

dst_addr

Check the destination IP address.

Syntax:

```
--dst_addr <IP address>;
```

Refer to `src_addr` for the IP address format.

Examples:

```
--dst_addr 10.10.10.0/24;
```

```
--dst_addr ![10.10.0/24, 10.10.20.0:24]:
```

```
--dst_addr fde0:6477:1e3f::2:ba;
```

ip_ver

Checks the IP version number.

Example:

Detect IP version 6 packets

```
--ipver 6
```

ipv6h

Detect next header value in IPv6 header. The value must be a decimal number. `ipv6h` can only be used when `ipver 6` is present.

Examples:

Detect IPV6 packets for which the next header is a hop-by-hop option:

```
--ipver6; --ipv6h 0;
```

Detect ICMPv6 packets for which the type value is 135 and the code value is 0:

```
--ipver6; --ipv6h 58; --protocol icmp; --icmp_type 135; --icmp code 0;
```

ip.total_length, ip.id, ip.ttl, ip.checksum

Check fields `total_length`, `id`, `ttl`, and `checksum` in the IPv4 header.

Syntax:

```
--ip.[decorations] <operator> <value>;
```

Valid operators: `=`, `!=`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Examples:

```
--ip.total_length >= 402;
```

```
--ip.id & 0xff = 0x37;
```

```
--ip.ttl in [64,65];
```

```
--ip.checksum != 0xff;
```

ip6.payload_length, ip6.next_header, ip6.hop_limit

Check fields `payload_length`, `next_header`, and `hop_limit` in IPv6 header.

Syntax:

```
--ip6.[decorations] <operator> <value>;
```

Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Examples:

```
--ip6.payload_length > 40;
```

```
--ip6.hop_limit < 0x4f;
```

```
--ip6.next_header in [1, 2];
```

ip [offset]

Access any fields in IPv4 header in a freelance mode.

Syntax:

```
--ip[offset] <operator> <value> [, word size] [, endianness];
```

Both `word size` and `endianness` are optional. By default, the engine uses `big endian` and `BYTE`, respectively. Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Examples:

```
--ip[2] >= 402,word;
--ip[4] & 0xff = 0x37,word;
```

ip6 [offset]

Access any fields in IPv6 header in a freelance mode.

Syntax:

```
--ip6[offset] <operator> <value> [, word size] [, endianness];
```

Both `word size` and `endianness` are optional. By default, the engine uses `big endian` and `BYTE`, respectively. Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Example:

```
--ip6[4] > 40,word;
```

TCP header options

Use TCP header options to check the properties of the TCP header.

src_port

Check the source port number or range.

Syntax:

```
--src_port [!]<number>;
```

The placement of `:` indicates less than or equal to:

```
--src_port [!]:<number>;
```

The placement of `:` indicates greater than or equal to:

```
--src_port [!]<number>:;
```

The placement of `:` indicates a range, exclusive of endpoints:

```
--src_port [!]<number>:<number>;
```

The optional prefix `!` means exclude.

Example:

Greater than or equal to 1000

```
--src_port 1000:;
```

dst_port

Check the destination port number or range.

Syntax:

```
--dst_port [!]<number>;
```

Equal to:

```
--dst_port [!]:<number>;
```

Greater than or equal to:

```
--dst_port [!]<number>:;
```

Range, exclusive of endpoints:

```
--dst_port [!]<number>:<number>; placement of : indicates  a range, exclusive of
endpoints
```

The optional prefix `!` means exclude.

Example:

Greater than or equal to 100 and less than or equal to 200:

```
--dst_port 100:200;
```

seq

Check the TCP sequence number value or range.

Syntax:

```
--seq <number>[,relative];
```

Equal to:

```
--seq =,<number>[,relative];
```

Greater than:

```
--seq >,<number>[,relative]:;
```

Less than:

```
--seq <,<number>[,relative];
```

Not equal to :

```
--seq !,<number>[,relative];
```

The optional field `relative` indicates the value is relative to the initial sequence number of the TCP session. No prefix defaults to "equal to."

Examples:

```
--seq <,12345;
```

```
--seq !,12345;
```

ack

Check the TCP acknowledge number for a specific value.

Syntax:

```
--ack <number>;
```

Equal to:

```
--ack =,<number>[,relative];
```

Greater than:

```
--ack >,<number>[,relative]:;
```

Less than:

```
--ack <,<number>[,relative];
```

Not equal to :

```
--ack !,<number>[,relative];
```

Examples:

```
--ack <,12345;
```

```
--ack !,12345;
```

tcp_flags

Specify the TCP flags to match in a TCP packet.

Syntax:

```
--tcp_flags <!*+FSRPAU120>[,<FSRPAU120>];
```

Flag	Description	Note
S	SYN	upper case required
A	ACK	upper case required
F	FIN	upper case required

Flag	Description	Note
R	RST	upper case required
U	URG	upper case required
P	PSH	upper case required
1	reserved bit 1	
2	reserved bit 2	
0	No TCP flags set	No TCP flags set

The first part defines the bits to match:

- The flags S, A, F, R, U, and P must be in upper case.
- If the first digit is 0, it will stop and ignore all of the following flags.
- * matches any one of the specified bits.
- + matches all of the specified bits, plus any others.
- ! matches if none of the specified bits is set.
- Default matches the specified bits exactly.

The second part is optional. It identifies the bits that should be masked off before matching.

Examples:

```
--tcp_flags 0,12;
--tcp_flags !SAFRUP,12;
 --tcp_flags S,12;
--tcp_flags S+;
--tcp_flags *SAFRUP12;
```

window_size

Check for the specified TCP window size.

Syntax:

```
--window_size [!]<number>;
--window_size [!] 0x<number>;
--window_size [>]<number>;
--window_size [<]<number>;
```

Examples:

```
--window_size 1000;
--window_size !0x1000;
```

tcp.src_port, tcp.dst_port, tcp.seq, tcp.ack, tcp.flags, tcp.window_size, tcp.checksum, tcp.urgent, tcp.any_option, tcp.payload_length

Check for these fields in the TCP header.

Syntax:

```
--tcp.[decorations] <operator><value>;
```

Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Examples:

```
--tcp.src_port in [1111,2222];
```

```
--tcp.flags & 0x0f = 0x6;
```

Iterate over all options:

```
--tcp.any_option = 0x6052, dword;
```

tcp [offset]

Access any fields in TCP header in freelance mode.

Syntax:

```
--tcp[offset] <operator><value> [, word size] [, endianness];
```

Both `word size` and `endianness` are optional. By default, the engine uses `BYTE` and `big endian`.

Valid operators: `=`, `!=`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Example:

```
--tcp[20] &0xF0 = 0x30;
```

UDP header options

Use these options to check the UDP header:

src_port

Check the source port number or range.

Syntax:

```
--src_port [!]<number>;
```

The placement of `:` indicates less than or equal to:

```
    --src_port [!]:<number>;
```

The placement of **:** indicates greater than or equal to:

```
    --src_port [!]<number>:;
```

The placement of **:** indicates a range, exclusive of endpoints:

```
    --src_port [!]<number>:<number>;
```

The optional prefix `!` means exclude.

Example:

```
    --src_port 1000:;
```

dst_port

Check the destination port number or range.

Syntax:

```
    --dst_port [!]<number>;
```

Equal to:

```
    --dst_port [!]:<number>;
```

Greater than or equal to:

```
    --dst_port [!]<number>:;
```

Range, exclusive of endpoints:

```
    --dst_port [!]<number>:<number>; placement of : indicates  a range, exclusive of
    endpoints
```

The optional prefix `!` means exclude.

Example:

```
    --dst_port 200:300;
```

udp.src_port, udp.dst_port, udp.length, udp.checksum

Check these fields in the UDP header.

Syntax:

```
     --udp.[decorations] <operator> <value>;
```

Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Example:

```
     --udp.scr_port in [1111,2222];
```

udp[offset]

Access any fields in UDP header in freelance mode.

Syntax:

```
--udp[offset] <operator> <value> [, word size] [, endianness];
```

Both `word size` and `endianness` are optional. By default, the engine uses `BYTE` and `big endian`.

Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Example:

```
--udp[20] &0xF0 = 0x30;
```

ICMP header options

Use these options to check the ICMP header:

icmp_type

Specify the ICMP type to match. Covers both ICMPv4 and ICMPv6.

Syntax:

```
--icmp_type <number>;
```

icmp_code

Specify the ICMP code to match. Covers both ICMPv4 and ICMPv6.

Syntax:

```
--icmp_code <number>;
```

icmp_id

Check for the specified ICMP ID value. This keyword is only used for packets with ICMP type `ECHO_REQUEST` or `ECHO_REPLY`.

Syntax:

```
--icmp_id <number>;
```

icmp_seq

Check for the specified ICMP sequence value. This keyword is only used for packets with ICMP type `ECHO_REQUEST` or `ECHO_REPLY`.

Syntax:

```
--icmp_seq <number>;
```

icmp.code, icmp.type, icmp.checksum

Check these fields in ICMPv4 header.

Syntax:

```
--icmp.[decorations] <operator> <value>;
```

Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Example:

```
--icmp.code in [1,2];
```

icmp [offset]

Access any fields in ICMPv4 header in a freelance mode.

Syntax:

```
--icmp[offset] <operator> <value> [, word size] [, endianness];
```

Both `word size` and `endianness` are optional. By default, the engine uses `BYTE` and `big endian`.

Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Example:

```
--icmp[1] in [1,2];
```

icmp6.code, icmp6.type, icmp6.checksum

Check these fields in ICMPv6 header.

Syntax:

```
--icmp6.[decorations] <operator> <value>;
```

Valid operators: `=`, `!=`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

icmp6 [offset]

Access any fields in ICMPv6 header in a freelance mode.

Syntax:

```
--icmp6[offset] <operator> <value> [, word size] [, endianness];
```

Both `word size` and `endianness` are optional. By default, the engine uses `BYTE` and `big endian`.

Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Example:

```
--icmp6[0] = 135;
```

Application level protocol options

dnp3.function_code, dnp3.group, dnp3.variation

Checks specific fields in the DNP3 protocol.

Syntax:

```
--dnp3.[Decorations] <operator> <value>
```

The following fields are current supported: `function_code`, `group`, and `variation`.

Valid operators: `=`, `!`, `>=`, `<=`, `&`, `|`, `^`, and `in`.

Example:

```
--dnp3.function_code in {0x81,0x82};

--dnp3.group = 0x33;

--dnp3.variation = 1;
```

ssl.fingerprint

Checks the fingerprint of SSL clients.

Syntax:

```
--ssl.fingerprint <operator> <value>
```

Valid operators: `=`, and `in`.

Example:

```
--ssl.fingerprint 0x1581DE884A87803B;
```

```
--ssl.fingerprint in {0x188A9C4DE686DD8,0x3B3C90A2C4571BA4};
```

dns.query_type

Checks the DNS query type.

Syntax:

```
--dns.query_type <operator> <value>
```

Valid optional operator: `=`.

Example:

```
--dns.query_type 16;
```

Payload options

You can use these options to detect contents in the payload of a packet or stream. IPS signatures use pattern matching for inspecting a packet payload. A pattern definition starts with a `--pattern` or a `--pcre` option name, and is followed by a series of modifiers.

The general format of a pattern definition is:

```
--pattern <string>; [--context c;] [--no_case;] [--distance n[,<refer>]]; [--within n
[,<refer>]];
```

Or, for PCRE patterns:

```
--pcre <string>; [--context c;] [--distance n[,<refer>]]; [--within n [,<refer>]];
```

pattern

Use the `pattern` keyword is specify which content to match. The pattern can contain mixed text and binary data. The binary data is generally enclosed with the pipe "`|`" characters, and is represented as hexadecimal numbers. It can match content in all packets for all protocols.

You must enclose the pattern to be matched in double quotation marks and follow it with a semicolon. The special characters (" ; \ | :) must be written as (`|22|`, `|3B|` or `|3b|`, `|5C|` or `|5c|`, `|7C|` or `|7c|`, `|3A|`| or `|3a|`). You can use backslash (\) to escape any character except a semicolon (;). However, using hexadecimal representation for `|5C|` for backslash is recommended as a good practice..

Syntax:

```
--pattern [!]"<text>";
```

`[!]` indicates the content is matched if it does not appear in the packet.

Examples:

```
--pattern "/level";

--pattern"|E8 D9FF FFFF|/bin/sh";

--pattern !"|20|RTSP/";
```

pcre

Use the `pcre` keyword to specify the content to match using Perl Compatible Regular Expression (PCRE). For the PCRE syntax, please refer to http://perldoc.perl.org/perlre.html.

The pattern to be matched must be enclosed in double quotation marks and followed by a semicolon. Certain special characters must be written as noted in the table below.

Special character	Expression
"	\x22
;	\x3B or \x3b
/	\x2F or \x2f

The IPS Engine handles PCRE a lot slower compared to normal pattern matching. PCRE should be used very carefully, especially for signatures that detect traffic from HTTP servers or traffic that does not specify a port.

Syntax:

```
    --pcre [!]"/<regular expression>/[<op>]";
```

The optional use of `[!]` indicates the content is matched if it does not appear.

<op>	Description
i	Case insensitive
s	Include new lines in the dot (.) meta character
m	By default, the string is treated as one big line of characters. ^ and $ match at the beginning and ending of the string. When you set m, ^ and $ match immediately following or immediately before any new line in the buffer, as well as the very start and very end of the buffer.
x	White space data characters in the pattern are ignored except when escaped or inside a character class.
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:

```
    --pcre "/\sLIST\s[^\n]*?\s\{/smi";
```

context

Use the `context` keyword to specify which protocol field the engine should search for a pattern in. If it is not present, the IPS engine searches for the pattern in the whole packet.

Syntax:

```
    --<context <field>;
```

<field>	Description
`PACKET`	Searches for the pattern in the whole packet This is the default setting.
`PACKET_ORIGIN`	Searches the original packet without protocol decoding

`URI`	This is only used to match content in the URI field of an HTTP request.
	Since there are various encoding standards that can be used in a URI, a character can be expressed in several ways. For example, %2f, %u002f, and %c0%af all represent "/". In order to cope with evasion attempts based on this, the content to be searched for in a URI must be decoded.
	The HTTP dissector decodes and normalizes the original URI field, placing the results in three buffers. The following three URI buffers search for the specified pattern.
	Original URI:
	`/scripts/. .%c0%af../winnt/system32/cmd.exe?/c+ver`
	Decoded URI:
	`/scripts/. ./../winnt/system32/cmd.exe?/c+ver`
	("\" is also converted to "/" in this phase.)
	rmdir URI:
	`winnt/system32/cmd.exe?/c+ver`

`HEADER`	The search range is the entire header of scanned HTTP, IMAP, SMTP, POP3 or SSH traffic.
`BODY`	The search range is the entire body of scanned HTTP, IMAP, SMTP, or POP3 traffic. The decoder has no separate buffer for the body section of above-mentioned traffic. Because of this, body data in different packets is not reassembled. The decoder just locates the beginning and end of the body in a packet payload and tries to match inside of it. If a signature has two patterns in a body section that are to be matched, but the patterns span across two separate packets, the second pattern will not be matched.
`BANNER`	The search range is the entire banner of scanned HTTP, IMAP, SMTP, POP3 or SSH traffic.
`HOST`	For an HTTP session, the search range is the "`Host:`" field of an HTTP header. For an HTTPS session, the search ranges is the server name field of Server Name Indication (SNI) in the client Hello packet and the Common Name (CN) field in the server certificate packet. For a DNS session, the search range is the query name field in a DNS request or response packet.
`FILE`	The search range for the file context can be one of: • decoded attachments for email protocols. • data sessions for FTP. • the body for HTTP. • Data sessions for TFTP (introduced in 7.0.18)

Examples:

```
    --context URI;
```

```
--context PACKET_ORIGIN;
```

Notes

- The IPS engine supports "packet-based" inspection, which means it inspects packets even if there are no sessions associated with them Many keywords, for example those for matching TCP/IP header fields, are enabled in packet-based inspection. If a pattern has the context value `PACKET`, `PACKET_ORIGIN`, or no context, it will be inspected using packet-based inspection.
- The `BANNER` and `BODY` are in the packet buffer.
- There is no body context in FTP, so file context should be used instead.
- For HTTP, the body context and the file context are the same. You can use either `--context file` or `--context body` to indicate where to match the pattern.
- If the file itself is zipped or archived, the engine currently does NOT decompress it.
- MIME parsing is supported for the email protocols SMTP, IMAP, POP3 and NNTP. Currently, all attachments fall under `--context file`. Most of the encoding methods are decoded, including *base64*, *uuencode*, *7/8bit*, *quota*, *binary*, and *quoted-printable*.
- For email protocols, use `--context body` to inspect content located in the body and is not an attachment.

no_case

Use the `no_case` keyword to indicate that the pattern should be matched in a case insensitive manner.

Syntax:

```
--no_case;
```

Examples:

```
--no_case;
```

distance, distance_abs, within, within_abs

Use these four keywords to specify the range (in bytes) of where the engine will search for a pattern.

- `distance` indicates the offset from the last reference point to start searching for a pattern
- `within` indicates the range of bytes from the last reference point which the engine should search for a pattern.

Syntax:

```
--distance <range> [,<refer>];

--distance_abs <range>[,<refer>];

--within <range>[, <refer>];

--within_abs <range>[, <refer>];
```

The `<refer>` field is the reference point for the `<range>`. If it is not included, the default is `MATCH`.

<refer>	Description
MATCH	The reference is the last matched pattern. This is the default setting.
PACKET	The reference is the beginning of the packet.
CONTEXT	The reference is the beginning of the pattern context.
REVERSE	Search for the pattern relative to the end of the packet or context. This is only accepted with the `--distance` option, and the reference must be `PACKET` or `CONTEXT`.
LASTTAG	The reference is the one set by last PSET.

Examples:

Search for the pattern within 50 bytes of the last matched pattern:

```
--pattern "/disp_album.php?"; --context uri; --no_case; --within 50,context;
```

```
--pattern "|05 00|"; --distance 0; --pattern "|6e 00|"; --distance 5; --within
2;
```

Count 10 bytes back from the end of the packet, then search for the pattern within 5 bytes:

```
--pattern "Host: "; --context header; --pattern !"|0a|"; --context header; --
within_abs 80; --distance 10,packet,reverse; --within 5,packet;
```

Notes

- If you use the keywords `distance` and `within` with the first pattern of a signature, set the `<refer>` field to `context`, as there are no previous matched patterns.
- The keywords `distance` and `distance_abs` indicate the minimum distance from the end of the last reference point to the beginning of the current pattern. The distance is counted from the next character after the last reference point. Both these keywords support negative range value. In this case, `distance` does not require the designated amount of data before the reference point while `distance_abs` does. For example, the following signature makes sure no `?` character is before the `/BBBB` pattern in the URI:

  ```
  --pattern "/BBBB"; --context uri; --within 200,context; --pattern!"?"; --context
  uri; --distance 200; --within 200;
  ```

  This signature works even if the `/BBBB` pattern in the URI is not preceded by 200 bytes of data.
- The keywords `within` and `within_abs` require that the whole pattern appear within the given range following the last reference point. If the `distance` or `distance_abs` keywords are also present, with the same reference point, the pattern will be matched from the specified distance to the range of bytes specified by the `within` or `within_abs` keywords.
- Use the keywords `distance_abs` and `within_abs` only for negative matches (patterns with the ! modifier). They indicate that the buffer following the reference point must be longer than or equal to the value specified by `<range>`. Compare the following two cases:

  ```
  --pattern !"|0a|"; --within 100,match;
  ```

  ```
  --pattern !"|0a|"; --within_abs 100,match;
  ```

- If the buffer after the previous match is shorter than 100, the first signature is matched. It is not recommended to use `distance_abs` and `within_abs` for a positive match because the behavior of these keywords is unreliable. It is better to use the keyword `data_at` instead.

For example:

```
--pattern "BBBBBB"; --pattern "DDDDDD"; --within_abs 200;

--pattern "BBBBBB"; --data_at 200,relative; --pattern "DDDDDD";
```

These two signatures are equivalent but the second one is recommended for a reliable match. A negative `<range>` value can be used to specify the range before the reference. Different types and references can be combined as range modifiers.

byte_jump, byte_test

Use the `byte_jump` keyword to move the reference point. The distance to be skipped is calculated from the value of bytes at a specified offset.

Use the `byte_test` keyword to compare the value of bytes at the specified offset with a given value. The keyword does not move the reference point.

If the data to be processed or skipped is beyond the end of the packet, the option is considered unmatched.

Syntax:

```
--byte_jump <”|bytes>,<offset|variable>[,<multiplier>[,modifiers]];

--byte_test <”|bytes>,<op>,<value>,<offset>[,<multiplier>[,modifiers]];
```

<field>	Description
`*|bytes`	Specifies the number of bytes from the payload to be converted. The value to be converted can be an ASCII string or binary. If the value is in binary, select between 1,2, or 4 bytes to be converted. If the value is an ASCII string, use the `string` modifier. For a fixed length ASCII field, specify the field's length. If it is a variable length ASCII field, use *, which will convert all bytes from the offset until the first nondigit character in the chosen base has been detected.
`op`	Defines the operator used to compare the value converted from the packet with the value specified. The following operators are accepted: > The value converted must be greater than the value specified. < The value converted must be less than the value specified. = The value converted must be equal to the value specified. ! The value converted must be not equal to the value specified. & The value converted AND the value specified must be not equal to zero. ~ The value converted AND the value specified must be equal to zero. ^ The value converted XOR the value specified must be not equal to zero.

<table>
<tr><th>&lt;field&gt;</th><th>Description</th></tr>
<tr><td><code>value</code></td><td>This is only used to match content in the URI field of an HTTP request.
Specifies the value to be compared. A hexadecimal number can be specified with the prefix 0x.
This also accepts variables and arithmetic operations (+ * /).
The following predefined variable is accepted:
<code>$PKT_SIZE</code>: the data will be compared with the packet size</td></tr>
<tr><td><code>offset</code></td><td>Specifies the starting point where the content should be converted in the payload. Negative offsets are accepted. See the <code>relative</code> modifier for more details.</td></tr>
<tr><td><code>multiplier</code></td><td>Optional. It must be a numerical value when present. The converted value multiplied by this number is the result to be compared or skipped.</td></tr>
<tr><td><code>modifiers</code></td><td>Accepts a combination (separated by commas) of the following values:
<table>
<tr><td>relative</td><td>Indicates that the offset should start from the last match point. Without it, the offset starts from the beginning of the packet.</td></tr>
<tr><td>big</td><td>Indicates that the data to be converted is in big endian (default).</td></tr>
<tr><td>little</td><td>Indicates that the data to be converted is in little endian.</td></tr>
<tr><td>string</td><td>Indicates that the data to be converted is a string.</td></tr>
<tr><td>hex</td><td>Indicates that the data to be converted is in hexadecimal.</td></tr>
<tr><td>dec</td><td>Indicates that the data to be converted is in decimal.</td></tr>
<tr><td>oct</td><td>Indicates that the data to be converted is in octal.</td></tr>
<tr><td>align</td><td>Rounds the number of converted bytes up to the next 32bit boundary, only used with <code>byte_jump</code>.</td></tr>
</table>
</td></tr>
</table>

Examples:

```
--byte_jump 4,0,relative;

--byte_test 4,>,3536,0,relative;

--byte_jump 4,20,relative,align;

--byte_jump 4,0,4,relative,little;

--byte_test 4,>,0x7FFF,4,relative;

--byte_ttest 4,>,$PKT_SIZE,4,relative;

--byte_test 4,>,$PKT_SIZE,4,2,relative;
```

Special options

This section addresses options that do not fall into the other categories covered in this guide.

crc32

Use the crc32 keyword to introduce to help in detection of file-based vulnerability.

Syntax:

```
--crc32 <checksum>,<file_length>;
```

`<checksum>` is a hexadecimal number representing the crc32 checksum of the file

`<file_length>` is a decimal number representing the file length.

Example:

```
--crc32 3174B5C8,20480;
```

data_at

Use the data_at keyword to verify the presence of data at the specified location in the payload.

Syntax:

```
--data_at <number>[,relative];
```

`<number>` is the payload offset to be checked for data.

`[relative]` indicates that the offset is relative to the end of the previous content match.

Example:

```
--data_at 100,relative
```

data_size

The data_size keyword was originally used to test the TCP/UDP/ICMP payload size of the packet being inspected. It has since been extended to support other size related fields in application protocols.

Because TCP is stream-based, not packet-based, the sender can intentionally fragment the original packets before they are transmitted to evade detection. For this reason using data_size on TCP packets may not always be reliable.

Syntax:

```
--data_size [op]<value[,field];
```

`[op]` is not required. The following operators are accepted:

<op>	Description
>	The data size must be greater than the value specified.
<	The data size must be less than the value specified.
=	The data size must be equal to the value specified. When `[op]` is not present, this is the default operator.

`<value>` is required. It is a decimal number that specifies the data size.

`[field]` is optional. One of the following keywords can be used:

[field]	Description
`payload`	The TCP/UDP/ICMP payload size is checked. This is the default setting.
`uri`	The URI length is checked.
`header`	The length of the header is checked.
`body`	The length of the body is checked.
`http_content`	The value of "Content-Length:" in an HTTP header is checked.
`http_chunk`	The chunk length value in the chunk header is checked.
`http_host`	The length of the `"HOST:"` line in an HTTP header is checked. The length count includes CRLF characters, the field name `"HOST:"`, all white spaces between the field name to the field value, and the field value. For example, `"HOST: www.example.com\r\n"` has a `data_size` of 25.
`smtp_bdat`	The SMTP data length in a BDAT command is checked.
`smtp_xexch50`	The SMTP data length in an XEXCH50 command is checked.

Examples:

```
--data_size <128;

--pattern "/admin_/help/"; --context uri; --no_case; --data_size >1024,uri;

--parsed_type HTTP_POST: --pattern "nsiislog.dll"; -context uri; --no_case: --data_size
>1000,http_content;
```

dhcp_type

The dhcp_type keyword is used to match DHCP request/response types. Any numeric value is allowed. The following table shows the types defined in RFC.

Syntax:

```
--dhcp_type <value>;
```

Type	<value>
DISCOVER	1
OFFER	2
REQUEST	3
DECLINE 4	4
ACK 5	5
NAK 6	6
RELEASE	7
INFORM	8
RELAY_CLIENTREQUEST	9
RELAY_SERVERREPLY	10

Example:

```
--dhcp_type 1;
```

file_type

Use the `file_type` keyword to match a class of file types, where each class contains several related subtypes. The IPS engine file type matching uses "file magic" to decide what type of file the content is, working in a manner similar to the Linux `file` command.

Currently, for the HTTP protocol, the first 13 or more bytes of body content will be categorized into a file type. If the result is a subtype of the class specified by a `--file_type <class>` option in a signature, it is a match.

In most cases, the identification of file type is handled by the file type function. However, when you are unsure about the file type, you can rely on the protocol fields if they contain some fields such as `content-type`. So, file type may not be limited to the subtypes listed below. For example, a tiff file will be marked as file type IMAGE by the IPS engine, even though it is not included in our own file type function.

The feature works in this manner:

1. The traffic is parsed by the protocol decoder.
2. A check is done to determine the presence of a file for HTTP, MIME, and FTP.

3. If the decoder finds that there is a file in the traffic, it will call the file type function to identify what type of file it is.
4. To narrow down the file type results, a class is selected based on the file type.
5. The result is saved with the protocol, for signature use. If a signature includes this keyword, it will check whether the given type has been matched.

Syntax:

```
--file_type <class>;
```

The file type classes are listed in the following table with their associated subtypes:

<class>	subtypes
COMPRESS	arj, bzip, bzip2, cab, gzip,lzh,lzw,rar, rpm, tar, upx, zip
IMAGE	gif, gif87a, gif89a, jpeg, png
SCRIPT	.bat, .css, .hta, .vba, .vbs, genscript, javascript, perlscript, shellscript, wordbasic
VIDEO	.avi, MPEG
AUDIO	.mp3
STREAM	stream
MSOFFICE	MSOFFICE, PPT
PDF	.pdf
FLASH	FLASH
EXE	.com, .dll, .exe
HTML	HTML
XML	XML, WORDML
UNKNOWN	unknown, ActiveMIME, AIM, FORM, HLP, MIME, .txt

Examples:

```
--file_type PDF;
--file_type EXE;
```

flow

The `flow` keyword is used to specify the direction of the detection packet. It can only appear once in a signature and is used in pattern and dissector signatures. It can be applied to TCP and UDP sessions. It accepts one of the following direction values:

<direction>	Description
from_client	Matches packets sent from the client to the server.

`<direction>`	Description
`from_server`	Matches packets sent from the server to the client.
`bi_direction`	Matches packets sent from the client to the server and from the server to the client.
`reversed`	Specifies that the attack is in the opposite direction from the detected packet. A typical case is when a brute force login is detected by matching a server packet indicating that a login has failed. This keyword will not affect detection. Its purpose is to tell the GUI to display the correct location for the vulnerability (client or server).

Syntax:

```
--flow <direction>;
```

Examples:

```
--flow from_client;
--flow bi_direction; dst_port 123;     //match if source or destination port is 123
```

log

Use the log option to specify additional types of information that can be included in the log messages generated by a signature.

Syntax:

```
    --log <keyword>;
```

<keyword>	Description
`dhcp_client`	DHCP client MAC addresses will be added to the log message in the format `dhcp_client=xx:xx:xx:xx:xx:xx;`
`dhcp_cc_id`	Circuit ID in DHCP relay messages will be added to the log in the format `dhcp_cc_id=4F4C2D30303143;`
`dns_query`	DNS query strings will be added to the log in the format `dns_query=www.yahoo.com;`

Example:

```
    --log DHCP_CLIENT;
```

parsed_type

Use the `parsed_type` keyword to match a packet or session attribute that can be identified by the dissectors. A signature can have more than one `--parsed_type` keyword.

Syntax:

```
--parsed_type <type>;
```

<type>	Description

`SSL_PCT`
`SSL_V2`
`SSL_V3`
`TLS_V1`
`TLS_V2`
	These types are used to identify the SSL and TLS versions.

`SOCK4` `SOCK5`	These match sessions using the SOCKS 4 or SOCKS 5 proxy protocols.

`HTTP_GET`	The HTTP request method to be matched is GET. This is valid for the lifetime of the request.
	In most cases, a signature using `--parsed_type`, similar to the one below:
	`--service HTTP; --parsed_type HTTP_GET;`
	can replace a pattern-based signature like this:
	`--service HTTP; --pattern "GET|20|" context uri; --within 4,context;`

`HTTP_POST`	The HTTP request method to be matched is POST. This if valid for the lifetime of the request.

`HTTP_CHUNKED`	The Transfer-Encoding type of the HTTP request to be matched is chunked. This is valid for the lifetime of the request.
	In most cases, a signature using the `parsed_type` keyword, similar to the one below:
	`--parsed_type HTTP_CHUNKED;`
	can replace one that looks for strings, like this:
	`--service HTTP; --pattern "TransferEncoding"; --context header; --no_case; --pattern "chunked"; --context header; --no_case; --distance 1;`

Examples:

```
--parsed_type HTTP_POST;

--parsed_type HTTP_GET;
```

rate, track

These two keywords make it possible to tell the IPS engine that instead of triggering a signature every time it is matched, it should only trigger if the signature is matched a given number of times within a specified time period. This feature can be used in reporting slow port scans, brute-force login attempts, and similar behavior.

For a regular signature, the IPS engine first compares all of the keyword options other than rate and track. If all the options are matched, IPS checks whether rate is specified for the signature. If it is not, IPS triggers the signature. If it is, IPS increases the counter and updates the timestamp, and checks whether the trigger rate has been reached.

Syntax:

```
--rate <count>,<duration>[,<limit>];
```

field	Description
`<count>`	The number of matches that must be seen before a log entry is generated.
`<duration>`	The time period over which matches are counted, in seconds.
`[limit]`	This improves the accuracy of the matched packet count by counting in strict time rather than averaging over a period of time. For example, `--rate 400,1,limit;`

```
--track <keyword>;
```

`<keyword>` specifies the packet property to track. The following case insensitive keywords are accepted:

<keyword>	Description
`src_ip`	Track the packet's source IP address.
`dst_ip`	Track the packet's destination IP address.
`dhcp_client`	Track the DHCP client's MAC address.
`dns_domain`	Track the domain name in the DNS query record.
`dns_domain_and_ip`	Track the DNS response with same domain name and IP address.

Notes

- If `--track` is specified, only matched packets which have the same specified keyword tracked are added to the counter.
- If `--rate` is used without `--track`, all matched packets are added to the counter and the signature is reported once the threshold is reached.
- IPS counts the average number of packets over a period of time. This might allow some extra packets to go through. Therefore, to ensure accuracy, the `limit` keyword can be used to allow counting to be done in a strict time. When `limit` is enabled the packet count is more accurate.

Example:

```
F-SBID( --name DHCP.FLOOD; --protocol UDP; --service DHCP; --dhcp_type 1; --rate 100,10;
--track DHCP_CLIENT; )
```

This signature generates an alert if IPS sees DHCP discover requests (`--dhcp_type 1;`) more than 100 times within 10 seconds (`--rate 100,10;`) from the same DHCP client (`--track dhcp_client;`).

rpc_num

The `rpc_num` keyword is used to check the RPC application, version and procedure numbers in a SUNRPC CALL request.

Syntax:

```
--rpc_num <application_number>[,<version_number>[,<procedure_number>]];
```

Examples:

```
--rpc_num 100221,*,*;

--rpc_num 100005,2,*;

--rpc_num 100000,*,4;
```

The `*` indicates that the number can have any value.

tag

Use this keyword in a signature to mark a session with a named tag, or to check whether a tag has been set for a session.

Pattern matching with IPS signatures is essentially packet-based. The tag keyword is mainly used when attack patterns appear in more than one packet or in different directions. A signature that matches an earlier packet in an attack can mark the session with a named tag, and the existence of the tag can be tested when ensuing packets in the same session are scanned.

The matching algorithm guarantees the order in which signatures are scanned. The signatures are sorted based on their tag dependencies. During packet inspection, the signatures are matched in this order, so that signatures that depend on other signatures are always scanned later in the process.

Syntax:

```
--tag <op>, [!}<name>[,timer,tuple[,all_sessions]];
```

`<name>` indicates the name of a tag.

`[!]` is only allowed in `test` operations. It returns true if the tag does not exist.

The `<op>` value determines which operation is performed.

`<op>`	Description
`set`	Mark the session with a named tag.

`<op>`	Description
`pset`	Mark the session with a named tag and remember the last reference point. This reference point can be referred by using `lasttag` for keywords `distance`, `within`, `distance_abs`, and `within_abs`.
`clear`	Remove the specified tag from the session.
`toggle`	Toggle the specified tag (`set <=> clear`) in the session.
`test`	Test the existence of the specified tag. Add `!` if the signature is to test the nonexistence of the specified tag.
`reset`	Clear all tags from the session.
`quiet`	Suppress logging when the signature is matched and ignores the signature's action. `QUIET` is normally included in the signature that `SET` the tag. Signatures with `--tag set;` should also have `--tag quiet` and `--status hidden;`.
`cset`	Set a cross session tag. Modifier `timer`, `tuple` and `all_sessions` are valid only when the `op` is `cset`. These modifiers altogether define which session should be marked with the named `tag`.
`timer`	The tag will be automatically removed after given seconds. If it equals 0 the specified tag will be removed immediately.

`tuple`	Accepts a combination (separated by ",") of `src_ip`, `dst_ip`, `src_port`, `dst_port`, and `protocol`. To reduce the performance impact, it only accepts following combinations:
	`src_ip`
	`dst_ip`
	`src_ip,dst_ip`
	`src_ip,dst_ip,dst_port`
	`src_ip,dst_ip,dst_port,protocol`
	`src_ip,dst_ip,protocol`
	`src_ip,protocol`
	`src_ip,src_port`
	`src_ip,src_port,protocol`
	`dst_ip,dst_port`
	`dst_ip,protocol`
	`Dst_ip,dst_port,protocol`

`all_sessions`	Copy the tag into both existing and new sessions. Without this, engine only copies the tag to the new sessions to reduce the performance impact.

The name of a tag should only contain printable characters. It should not contain spaces, commas, exclamation marks, or semicolons.

By default, a newly-created tag is in the un-set state.

Patterns in `tag set` and `tag test` signatures can appear in the same packet together.

Examples:

```
--tag set,Tag.Rsync.Argument;

--tag clear,tag.login;
```

```
--tag test,Tag.Rsync.Argument;
--tag test, !DHTML.EDIT.CONTROL.CLSID;
```

Application options

The keywords in the list that follows mainly deal with application signatures.

app_cat

The app_cat keyword specifies the category of the application signature. This is a required keyword for application control signatures. These signatures will appear under Application Control instead of IPS configuration.

ID	Category
2	P2P
3	VoIP
5	Video/Audio
6	Proxy
7	Remote.Access
8	Game
12	General.Interest
15	Network.Service
17	Update
21	Email
22	Storage.Backup
23	Social.Media
25	Web.Client
26	Industrial
28	Collaboration
29	Business
30	Cloud.IT
31	Mobile

To display a complete and current list of application signature categories and their corresponding ID numbers, enter the following CLI commands:

```
config application list
   edit default
      config entries
         edit 1
```

```
            set category ?
         next
      end
   next
end
```

Syntax:

```
--app_cat <category_id>;
```

weight

Use this keyword to specify the weight to be assigned to the signature. While optional, this keyword is useful because it allows a signature with the higher weight to have priority over a signature with a lower weight.

The weight must be between 0 and 255. Most of the signatures in the Application Control signature database have weights of 10; botnet signatures are set to 250. A range of 20 to 50 is recommended for custom signatures.

Syntax:

```
--weight <weight_int>;
```

Example:

```
F-SBID(--attack_id 8151; --vuln_id 8151; --name "Windows.NT.5.Web.Surfing"; --default_
action drop_session; --service HTTP; --protocol tcp; --app_cat 25; --flow from_client; -
- pattern !"FCT"; --pattern "Windows NT 5.1"; --no_case; --context header; --weight 40;
)
```

FORTINET

Syntax Guide

Custom IPS and Application Control Signature 7.0