PSP Architecture Specification

April 27, 2022

Document Purpose

The PSP Security Protocol (PSP) is a security protocol created by Google for encryption in transit. PSP uses several of the concepts from IPsec ESP to provide an encryption encapsulation layer on-top of IP that is streamlined and custom-built to address the requirements of large-scale data centers. The purpose of this document is to publicly disclose (i.e., open source) the PSP architecture specification to the technical community in the hope that it will be beneficial to others. More specifically, that it will lead to additional implementations and broader adoption.

Introduction

PSP uses the concept of a "Security Association" (SA) to represent the set of traffic to be handled with a particular set of crypto state (keys, IVs, sequence numbers, etc.). An SA is always unidirectional (simplex). Bidirectional traffic, even in the same TCP connection, requires at least one SA in each direction.

PSP is designed to enable scaling to large numbers of SAs without requiring per-SA hardware state in the NIC. On transmit, the host networking stack can provide the SA details with each transmit descriptor. On receive, the NIC hardware can derive the SA data (including the decryption key) using only the data in the received packet, plus a secret kept internally to the NIC (described below).

Alternatively, implementations may choose to maintain per-SA hardware state. For example, an implementation may choose to carry an SA index, instead of the entire SA in the transmit descriptor. This choice might be made as a tradeoff that favors higher packet per second rates (due to the smaller descriptor size) over capacity and connection setup rate scaling. Additional details about transmit key management are provided in Appendix A. Similarly, an implementation may choose to lookup the decryption key using only the data in received packets, instead of deriving the SA data. However, such an implementation choice would negate one of the key advantages of PSP.

PSP supports multiple modes of operation, distinguished by the Version field in the PSP header:

- Encryption and authentication for packet traffic, using AES-GCM
- Authentication (but not encryption) for packet traffic, using AES-GMAC

PSP only authenticates the portion of the packet after and including its own header. In particular, it does not authenticate the outer IP or UDP headers.

When operating in AES-GCM mode, PSP will also encrypt the packet data starting from a “crypt offset”. The crypt offset can be specified by the transmit descriptor or by configuration. The crypt offset is carried in the PSP header and is used by the receiver to decrypt the packet data. The crypt offset field is irrelevant in AES-GMAC authentication-only mode.

Why PSP

Google designed PSP specifically to address the line-speed encryption requirements of large-scale data centers. Google has deployed PSP, is committed to continued use of PSP, and encourages vendor implementations. While PSP leverages many concepts from IPsec, it provides several advantages. At a high-level, those advantages are:

1. Simplicity,
2. Improved Functionality, and
3. Increased Scalability.

Simplicity is achieved in two primary ways:

1. Streamlining the feature set to only those needed in large-scale data centers, and
2. Altering the encapsulation format in a few strategic ways.

The feature streamlining includes:

- Simplifying replay protection by increasing reliance on other network layers so less state is needed at the IP layer,
- Eliminating the use of authentication headers at the IP level (i.e., IPsec AH protocol), and
- Use of a small number of strong crypto algorithms, AES-128-GCM and AES-256-GCM (IPsec mandates support for multiple other crypto standards).

Simplifiying encapsulation format alterations include:

- Moving the “Next Header” field so that it appears at the beginning, rather than near the end (which makes hardware header generation/parsing easier),
- Adding a “Header Extension Length” field that contains the total length of the PSP header (which also makes hardware parsing easier),
- Making the IV field required rather than optional, and
- Removing the sequence number field (instead relying on the IV to be monotonically increasing).

Improved functionality is achieved by:

- Use of a UDP encapsulation
    - This allows information from the inner headers to be exposed in the source port field of the UDP header. Storing a flow hash of inner header fields in the UDP source port field allows switches to use entropy from the inner headers when

performing load balance operations to select a next hop, which is important for maximizing utilization of available hardware resources.

    - The base IPsec ESP RFC, 4403, does not specify a UDP encapsulation; however, a UDP encapsulation is specified in UDP Encapsulation of IPsec ESP Packets, RFC 3948.

- Capability for starting the data encryption at a flexible offset
    - This allows exposing things like the source and destination port numbers in cleartext, which enables firewalling/routing without decryption by intermediate nodes in the network and richer network monitoring based on packet sampling.
- Use of a timestamp-based IV system, which can be useful for doing transit-time measurements

Increased scalability is associated with not requiring per-SA hardware state. The increased scalability has multiple dimensions:

1. The SA capacity is not constrained by NIC hardware resources
2. The connection setup rate is not constrained by the rate at which SAs can be inserted/deleted by the NIC hardware
3. NIC hardware can be simpler and more cost-effective

The number of secure connections established by a single server in a large-scale data center can currently reach 8 digits (i.e., 10M+). The amount of memory required to maintain two SA data structures for each of those connections is significant. Assuming the size of each SA data structure is 256B, the total DRAM requirement for the SAs of 10M connections is 256B * 2* 10M = 5GB. The PSP receive key derivation property eliminates the requirement for half of that memory; this savings of 2.5GB can represent a significant percentage of the NIC's DRAM capacity. Additionally, no NIC-resident memory is required for SAs when PSP is implemented by a NIC that supports specification of the SA info in the TX descriptor (or specification of an address where the SA resides in host memory).

Similarly, the rate at which secure connections are established by a single server in a large-scale data center can currently reach 6 digits (i.e., 100K+ cps). Supporting insertion/deletion into a NIC-resident SA database at these rates can be challenging and may require dedicated hardware resources. Since PSP can eliminate the need for a hardware SA database, the NIC hardware can be simpler, and the associated NIC operations are no longer a potential bottleneck on the maximum connection setup rate.

The scaling advantages associated with PSP are especially important because of projections for more and faster connections moving forward in time, since each new generation of NIC hardware will no longer have to try and keep pace with rapidly increasing SA capacity and connection rate requirements.

The scaling advantages can also enable additional applications, such as using PSP as a TLS alternative, given that PSP is much easier to offload to hardware than TLS.

PSP Architecture

Security Association (SA)

An SA defines a grouping of packets that are treated identically from an encryption and an authentication standpoint. An SA can be any granularity ranging from individual TCP connections to VM-to-VM tunnels aggregating all traffic between a pair of VMs. Configuration of an SA includes:

- SPI (Security Parameters Index)
- Encryption key (stored on the sending side; derived on the receiver side)
- Crypt Offset specifying the amount of packet after the IV allowed to be cleartext (but still authenticated) when operating in AES-GCM mode
- Traffic flow confidentiality configuration (TFC using the techniques described for IPSec)
- Lifetime of the SA (requirements for when it must be destroyed)

Key Derivation

To avoid storing per-SA decryption keys on the receiver side, the decryption keys are derived from the SPI (provided in the packet) plus a secret known only to the receiver side of the NIC. During initial handshaking, the sender asks the receiver to create an SA encryption key, which the receiver then communicates securely to the sender. The protocol used for this key exchange is outside the scope of this document. The sender must remember the encryption key for the SA, but the receiver can re-derive it for each packet. The same arrangement is repeated in reverse, to provide encryption for response packets flowing from the receiver to the sender.

Specifically, each NIC has two 256-bit AES keys, called *master keys*, not shared with any hosts including its own, or with any other NICs. The master keys are "critical security parameters", which are kept ephemerally in on-NIC RAM, and must not be stored on any persistent medium. To promote master key protection, NIC RAM must not be accessible via the PCIe bus, and efforts should be taken to prevent key extraction via trivial physical access (e.g., JTAG debugging ports disabled).

One master key is “active.” All new requests for encryption keys derive from the active key. When all the possible SPIs for that key have been used, or after a prescribed "rotation period" (nominally 24 hours), the other master key will become active. The old key is kept in-place, to allow decryption of existing connections. Once all existing connections are done, or their SA lifetime’s have expired (which forces transfer to a new SA), the old key is replaced, and is now a candidate to take over as active once the currently active key runs out of SPIs.

The receiver must know which master key to use for derivation using only the data provided in the packet. Therefore, the high-order bit of the SPI is reserved to indicate this: 0 implies master_key0, 1 implies master_key1. As only the receiver knows which master key is currently active, this means the receiver must select the SPI. Thus the initial handshake is not just requesting the derived key, but also the matching SPI.

Since the key for an SA is derived entirely from its SPI and master key, and keys must not be reused between SAs. SPIs must also not be reused until the master key assigned to that SPI is rotated. The NIC firmware responsible for generating SPIs must guarantee this. There will be no API that allows user-space to provide a SPI; the SPI is always generated by NIC.

PSP supports the use of either AES-GCM-128 (PSP version 0) or AES-GCM-256 (PSP version 1) as the encryption algorithm used for packet payloads.

Algorithm requirements

The key derivation function for PSP is intended to fulfill the following requirements:

- It must be capable of generating security-association keys for both PSP version 0 (which uses AES-GCM-128 encryption) and PSP version 1 (which uses AES-GCM-256 encryption)
- It must derive security-association keys from a pair of 256-bit master keys (which are presumed to be generated using strong, approved cryptographic techniques)
- It must be fully compliant with all requirements in FIPS Publication 140-2 and similar best practices

Algorithm elements and implementation

The basic elements are as follows:

1. Key derivation is performed via an implementation of NIST SP 800-108 *Recommendation for Key Derivation Using Pseudorandom Functions* Section 5.1 *KDF in Counter Mode*
2. The associated pseudorandom function (PRF) is based on NIST SP 800-38B *Recommendation for Block Cipher Modes of Operation – The CMAC Mode for Authentication* which is accepted for this purpose by NIST SP 800-38B, Section 4
3. The block cipher which underlies the CMAC PRF is AES-256. This block cipher has an input and output block size of 128 bits.

The *KDF in Counter Mode* specification provides a framework for embedding information in the blocks of data fed to the PRF. Each input block consists of four parts:

1. A counter, which is set to 1 for the first or only block of keying material generated (in our case, the first 128 bits of the security association key) and is incremented for each additional block of keying material generated (e.g. the second 128 bits of a 256-bit security association key)
2. A label, which identifies the usage of the keying material (in our case, it identifies the version of the PSP protocol for which the key is being generated)
3. A context, which identifies the specific use of the key (in our case, the Security Parameters Index which identifies the security association using the key, and also identifies the master key with which the SPI is associated)
4. A length field, which identifies the amount of keying material being generated (in our case, either 128 or 256 bits)

The *KDF in Counter Mode* specification does not define the encodings of these fields. For the purpose of PSP, they are defined as four 32-bit fields, in the order given above. When representing integers (counter, context, and length) the integers are encoded in network byte order ("big-endian"). Specifically:

1. The "counter" field is defined as the hexadecimal byte values 00 00 00 01 when generating a single 128-bit block of keying material for a PSP version 0 key, or when generating the first 128 bits of a 256-bit PSP Version 1 key. It is defined as the hexadecimal byte values 00 00 00 02 when generating the second half of a 256-bit PSP Version 1 key.
2. The "label" field is defined as the hexadecimal bytes 50 76 30 00 ("Pv0" in C string form) when generating a key for PSP Version 0, and 50 76 31 00 ("Pv1" in C string form) when generating a key for PSP Version 1. An implementation of the KDF may legitimately construct the label field by OR'ing the PSP version number from a PSP packet header into the third byte of the 50 76 30 00 byte string, if that's convenient.
3. The "context" field is the Security Parameters Index. As the SPI is present in network byte order in the PSP packet header, an implementation of the KDF may simply copy the four SPI bytes from the header without rearrangement. The most significant bit of the SPI (and thus the most significant bit of the first byte) acts as a selector, controlling whether the KDF uses master key 0 (MSB is clear) or master key 1 (MSB is set) when executing the PRF.
4. The "length" field is a network byte order representation of the total length of the key being generated: either 00 00 00 80 for a 128-bit key for PSP Version 0, or 00 00 01 00 for a 256-bit key for PSP Version 1.

Examples of key derivation

Assume the existence of two AES-256 master keys K_0 and K_1.

K_0 = 34 44 8a 06 42 92 60 1b 11 a0 97 8f 56 a2 d3 4c f3 fc 35 ed e1 a6 bc 04 f8 db 3e 52 43 a2 b0 ca
K_1 = 56 39 52 56 5d 3a 78 ae 77 3e c1 b7 79 f2 f2 d9 9f 4a 7f 53 a6 fb b9 b0 7d 5b 71 f3 93 64 d7 39

When generating a PSP version 0 key for SPI 0x12345678, the KDF will perform one call to the CMAC function:

key := CMAC(K_0, 00 00 00 01 50 76 30 00 12 34 56 78 00 00 00 80)

key = 96 c2 2d c7 99 19 80 90 b7 4b 70 ae 46 8e 4e 30

When generating a PSP version 0 key for SPI 0x9A345678, the KDF will perform one call to the CMAC function:

key := CMAC(K_1, 00 00 00 01 50 76 30 00 9A 34 56 78 00 00 00 80)

key = 39 46 da 25 54 ea e4 6a d1 ef 77 a6 43 72 ed c4

When generating a PSP version 1 key for SPI 0x12345678, the KDF will perform two calls to the CMAC function, and then catenate the results to construct the key:

key_1 := CMAC(K_0, 00 00 00 01 50 76 31 00 12 34 56 78 00 00 01 00)
key_2 := CMAC(K_0, 00 00 00 02 50 76 31 00 12 34 56 78 00 00 01 00)

key := key_1 || key_2

key = 2b 7d 72 07 4e 42 ca 33 44 87 f2 99 0e 3f 8c 40 37 e4 36 f3 82 83 44 9b 76 46 3e 9b 7f b2 e3 de

(All CMAC calculations were performed using http://artjomb.github.io/cryptojs-extension/ and were confirmed using a test program based on the OpenSSL crypto libraries.)

IV Generation

The AES GCM encryption used by PSP demands that per-packet initialization vector (IV) never be repeated for the same encryption key. A single duplicate IV can undermine the encryption of the entire stream. This has multiple implications:

1. The lifetime of an SA must be limited, as eventually all possible IV's will be exhausted. See next section for more details.
2. The NIC must provide a mechanism to guarantee unique IV's

NIC implementations must guarantee uniqueness of the IV by implementing a timestamp counter and generating IVs using the process described below:

1. The NIC must implement a 64-bit timestamp counter that increments in 1-picosecond units. This 64-bit picosecond counter wraps around every 2^64/10^12/3600/24 = 213 days. If the core clock of the NIC is 800MHz, the timestamp counter increments by 1250 every clock cycle.
2. On transmit, the NIC must use the 64-bit timestamp counter along with the 32-bit SPI to generate a 96-bit IV required by AES-GCM. The NIC must guarantee that each packet will have a strictly increasing IV (modulo 64-bit wraparound) and that consecutive packets can never be sent with the same IV value.
3. On receive, the NIC must also provide a mechanism to pass through the IV from each decrypted and validated (and possibly marked) packet to the final consumer of the packet.

Master Key Lifetime

As noted above, an IV must never be repeated within the lifetime of an SA. As SA traffic keys are generated from the SPI and the master key, and SPI's may have arbitrarily long lifetimes, the master keys must have limited lifetime to ensure limited SA lifetime.

Limited SA lifetimes are achieved by using master key rotation which refers to swapping of which of the two master keys is active. This means new SAs will no longer be generated with the now-rotated key, but old SAs will continue to use the old key. A "double rotation" is needed to actually evict a master key and guarantee that no further data with its SPIs is usable.

Another consideration that factors into the computation of when master keys must be rotated is the wrap-around of the SPI. The SPI field is 31 bits long (since 1 bit is taken as the master key indicator), so the master keys must be rotated before the key's SPI space is exhausted. Assuming the master key is rotated daily and the SPI space is 2^31, the maximum average SPI rate consumption rate is 2^31 / (3600*24) = 24855 SPI/second.

Entropy for Master Key Generation

Management firmware generates master keys on the NIC. A good source of entropy is required to generate unpredictable keys. The NIC must provide a true random number generator (TRNG) that can be used as the source of entropy.

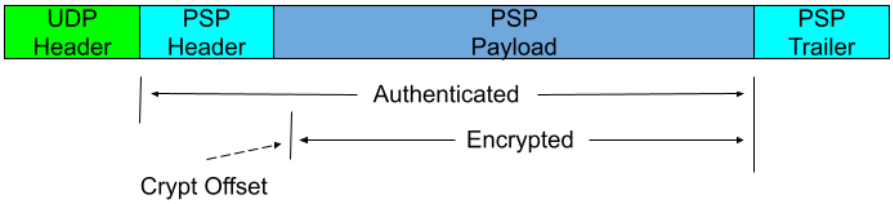
Replay Protection

PSP does not provide replay protection. It is assumed that replay protection will be provided by the layer 4 protocol. For example, TCP PAWS in conjunction with ISN (initial sequence number) randomization is expected to provide good replay protection. Other transports using PSP must provide similar replay protection.

Wire Formats

This section documents the PSP packet formats. The PSP header format complies with A Uniform Format for IPv6 Extension Headers.

PSP Packet Format



The format of the PSP packet is shown in the figure above. Currently, PSP packets are always encapsulated in a UDP packet, but alternative encapsulations may be used in the future.

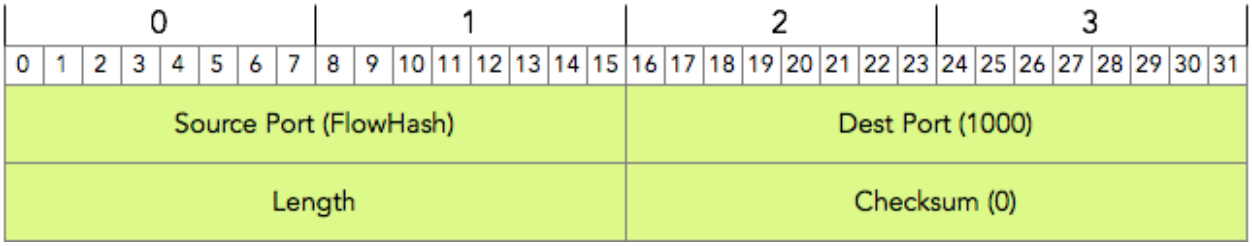
More specifically, a native IPv6 encapsulation, where the PSP Header is an IPv6 Extension Header, may be used in the future. The UDP encapsulation was originally selected for IPv6 due to a lack of switch fabric support for next-hop load balancing based on the Flow Label field of the IPv6 header. When switches support load balancing based on the IPv6 Flow Label, it may be desirable to support an alternative IPv6 encapsulation that eliminates the UDP header in order to reduce the bandwidth overhead of the PSP encapsulation. In that case:

- The next-hop load balancing entropy that is currently carried in the UDP source port would be carried in the IPv6 Flow Label, and
- support for the UDP IPv6 encapsulation will still be required for backwards compatibility.

No alternative IPv4 encapsulations are currently envisioned.

The PSP header follows the UDP header and is followed by the PSP payload data and the PSP trailer which contains the Integrity Checksum Value (ICV).

UDP Header



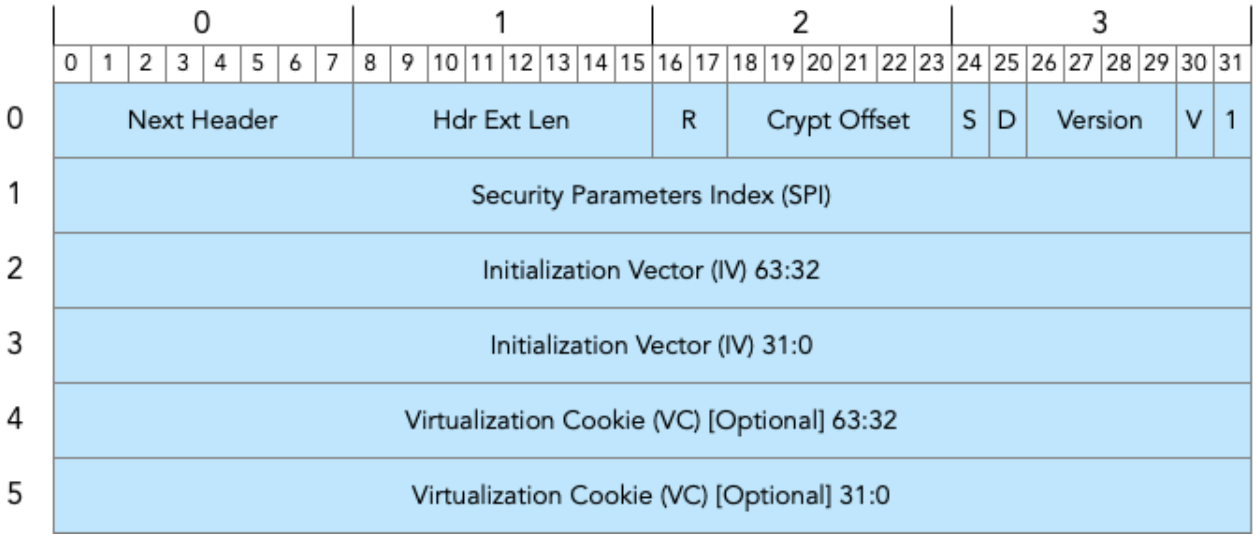
The format of the UDP header is shown in the figure above. PSP uses UDP destination port 1000 with a value based on a flow hash of inner header fields stored in the UDP source port. Including a value based on the flow hash in the UDP source port allows switches to use entropy from the inner headers in selecting a next hop based on ECMP / WCMP routing.

An option should be provided where the value stored in the UDP source port is obtained by a table lookup. With this option, the value computed by the flow hash would be used as an index into a lookup table. The lookup table should support at least 16 entries. The value stored in the UDP source port should be provided as an input to the PSP hardware interface.

An option should be provided where the UDP source port value is provided by software in a manner that is transparent to the NIC hardware. An option where the UDP source port value is determined by NIC-resident packet forwarding logic that is performed before the PSP hardware interface is desirable.

A UDP checksum value of 0 must be supported. A configuration option to populate the UDP checksum field with a valid checksum may be provided. The receiver must accept packets with a UDP checksum of 0. These UDP checksum requirements apply to both IPv4 and IPv6, and are in compliance with RFC 6935 and RFC 6936.

PSP Header Version 0



The format of the PSP Header Version 0 is shown in the figure above. The various fields are described in the table below. Additional versions of the PSP Header may be defined in the future.

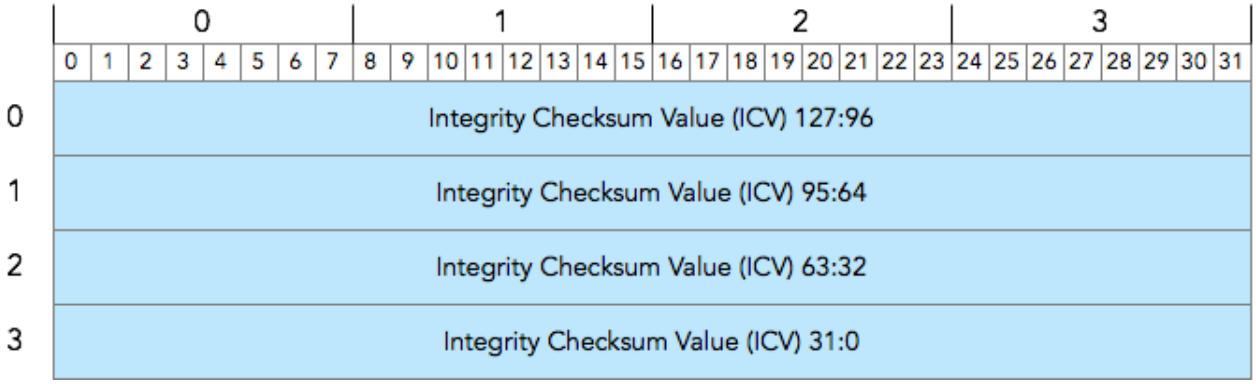
Field	Description
Next Header (8 bits)	An IP protocol number, identifying the type of the next header. For example: ● 6 for transport mode when next header is TCP ● 17 for transport mode when next header is UDP ● 4 for tunnel mode when next header is IPv4 ● 41 for tunnel mode when next header is IPv6
Hdr Ext Len (8 bits)	Length of this header in 8-octet units, not including the first 8 octets. When the Hdr Ext Len is non-zero, a Virtualization Cookie and/or other header extension fields may be present. The presence of a Virtualization Cookie is determined by the state of the V bit. The format of other header extension fields is determined by the applications and is opaque to the PSP hardware.
R (2 bits)	Reserved bits. Opaque to crypto engines. Set to zero on transmit, ignored on receive.
Crypt Offset (6 bits)	The offset from the end of the Initialization Vector to the start of the encrypted portion of the payload, measured in 4-octet units. In AES-GCM mode, it is permissible for the encrypted portion of the payload to be empty *if and only* if Crypt Offset "points" immediately after the last octet of the payload. This requires that the payload length mod 4 be equal to 0, and that the length of the payload (starting immediately after the Initialization Vector) be 252 octets or less. PSP packets are not allowed to have a "negative length" encrypted portion - the Crypt Offset must not be set to a value higher than "immediately past the last octet of the payload". The behavior of PSP when encountering such packets is implementation-dependent. PSP implementations may either process such packets as valid (tagging upon transmit and validating upon reception), or discard them as malformed. Users of PSP should not generate such packets nor depend on being able to receive them. In AES-GMAC mode, the Crypt Offset is reserved, and should be set to zero.
S (1 bit)	Sample at Receiver. This bit is opaque to crypto engines, but will be used to trigger packet sampling at the receiver.
D (1 bit)	Drop after Sampling. This bit is opaque to crypto engines, but will be used by received packet processing to drop the packet after sampling.

<table>
<tr><td>Version (4 bits)</td><td>Codepoint identifying the PSP Header Version, the operational mode, (AES-GCM with encryption and authentication, or AES-GMAC with authentication only) and the specific encryption/authentication algorithm to be used. Values currently defined:

<table><tr><td>0</td><td>PSP Header Version 0, AES-GCM-128</td></tr><tr><td>1</td><td>PSP Header Version 0, AES-GCM-256</td></tr><tr><td>2</td><td>PSP Header Version 0, AES-GMAC-128</td></tr><tr><td>3</td><td>PSP Header Version 0, AES-GMAC-256</td></tr></table>
All other values are reserved, and should not be transmitted. Packets received with a reserved (or unsupported) Version Codepoint should be discarded.

Support for Version Codepoint 0 is required for all PSP implementations. Support for Version Codepoint 1 is strongly encouraged. Support for Version Codepoints 2 and 3 is optional.</td></tr>
<tr><td>V (1 bit)</td><td>The Virtualization-Cookie-Present bit. The PSP Security payload includes the Virtualization Cookie field if and only if V is set.</td></tr>
<tr><td>Security Parameters Index (32 bits)</td><td>An arbitrary 32-bit value that is used by a receiver to identify the Security Association (SA) to which an incoming packet is bound.</td></tr>
<tr><td>Initialization Vector (64 bits)</td><td>A unique value for each packet sent over a Security Association.</td></tr>
<tr><td>Virtualization Cookie (64 bits)</td><td>An optional field, present if and only if V is set. It may contain a Virtual Network Identifier (VNI) or other data, as defined by the implementation.</td></tr>
</table>

PSP Trailer

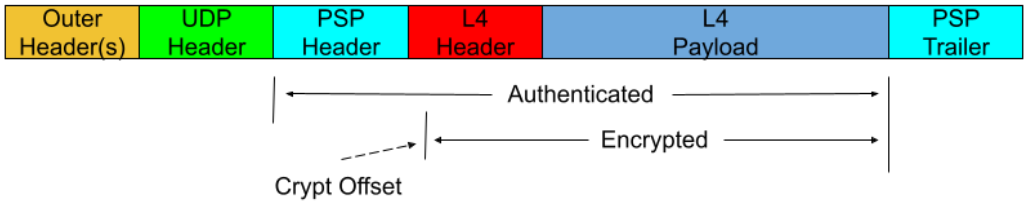


The format of the PSP trailer is shown in the figure above. The fields are described in the table below.

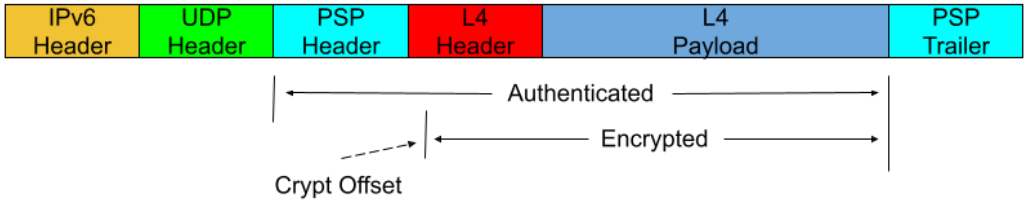
Field	Description
Integrity Checksum Value (128 bits)	An AES-GCM Authentication Tag authenticating the PSP from the start of the PSP header to the end of the payload.

PSP Transport Mode Packet Format

The general format of a PSP packet in transport mode is shown in the figure below. The PSP payload carries a layer 4 packet. The PSP packet along with the UDP encapsulation header is carried as the payload of an outer encapsulation. The entire PSP packet up to the PSP trailer is authenticated. The crypt offset specifies what part of the PSP payload is encrypted.



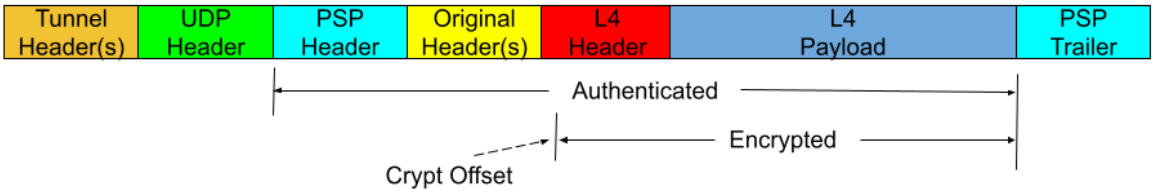
A more specific example of the most common format of a PSP packet in transport mode is shown in the figure below. In the specific example, the outer header is IPv6. Support must also be provided for encapsulations where the outer header is IPv4. Flexibility to support additional outer encapsulation formats is desired (examples include GRE and MPLS).



The transport mode packet format is typically used in non-virtualized environments.

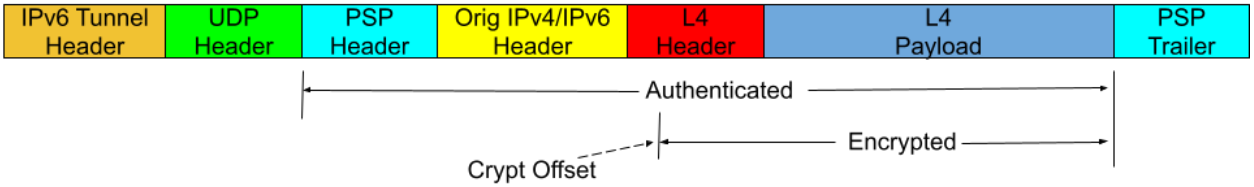
PSP Tunnel Mode Packet Format

The general format of a PSP packet in tunnel mode is shown in the figure below. The PSP payload carries the original packet. The PSP packet along with the UDP encapsulation header is carried as the payload of the tunnel encapsulation. As was the case for transport mode, the entire PSP packet up to the PSP trailer is authenticated and the crypt offset specifies what part of the PSP payload is encrypted.



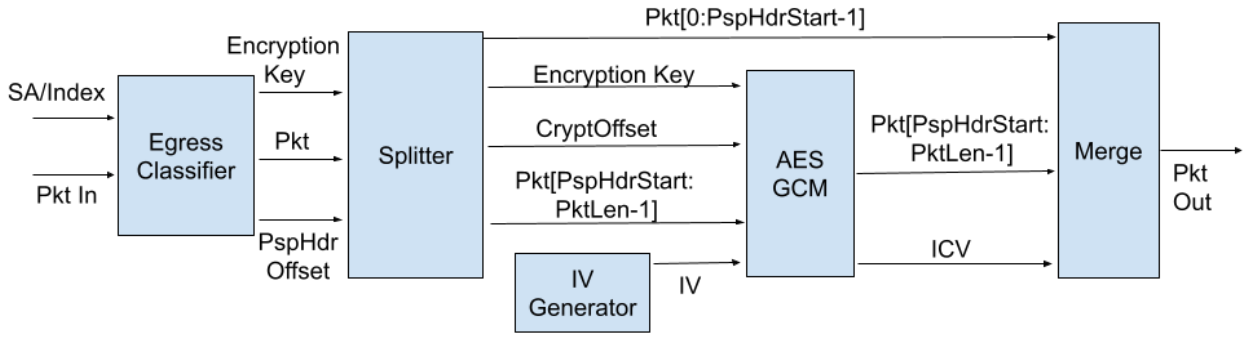
A more specific example of the most common format of a PSP packet in tunnel mode is shown in the figure below. In the specific example, the tunnel header is IPv6, and the original packet is IPv4/IPv6. Flexibility to support additional encapsulation formats is desired.

.



The tunnel mode packet format is typically used in virtualized environments.

NIC Transmit Processing



On transmit, PSP packet processing can be implemented as shown by the example block diagram in the figure above. The egress classifier is responsible for outputting the encryption key and building the PSP encapsulation (including the PSP and UDP headers). The encryption key can be produced in multiple ways:

1. By being provided as an input
2. By lookup based on an input index
3. By lookup in a flow table based using packet header fields as the key

The egress classifier must also communicate the PSP header offset (from the start of the packet) to the Splitter block.

The Splitter block determines the crypt offset from the PSP header and divides the packet into 2 portions:

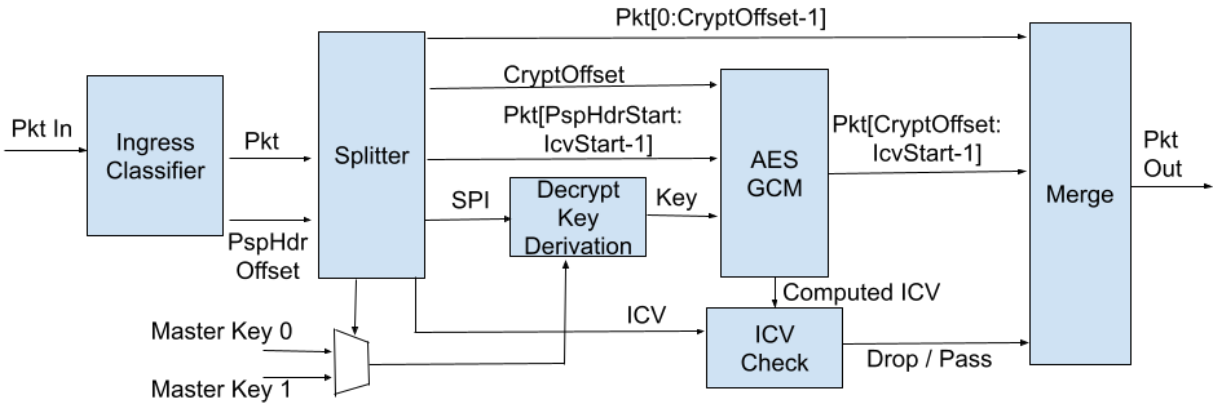
1. The cleartext portion from the start of the packet to the crypt offset, and
2. The PSP packet from the start of the PSP header to the end of the packet.

The PSP packet along with the Crypt Offset is passed to the AES-GCM block. The AES-GCM block gets the IV from the IV Generator block and outputs the encrypted portion of the packet from Crypt Offset to the end of the packet along with the authentication tag (ICV) computed over the entire PSP packet.

The Merge block puts together the 3 pieces of the packet to create the outgoing encrypted packet. Those 3 pieces are:

1. The cleartext portion from the Splitter block,
2. The ciphertext portion from the AES-GCM block, and
3. The ICV from the AES-GCM block.

NIC Receive Processing



On receive, PSP packet processing can be implemented as shown by the example block diagram in the figure above. The ingress classifier is responsible for identifying the incoming packet as a PSP packet and determining the offset of the PSP header from the start of the packet. A simple TCAM-based classifier can be used to identify the PSP packet formats and output the PSP header offset. The ingress classifier must communicate the PSP header offset (from the start of the packet) to the Splitter block.

The Splitter block breaks up the incoming packet into several pieces and does the following:

1. The cleartext portion of the packet from the start of the packet to the crypt offset is passed to the Merge block,
2. The PSP packet including the PSP header but not including the ICV is passed to the AES-GCM block,
3. The Crypt Offset is also passed to the AES-GCM block,
4. The SPI is passed to the Decryption Key Derivation block.
5. The MSB of the SPI is used to select the active master key used by the Decryption Key Derivation block, and
6. The ICV is passed to the ICV Check block.

The Decryption Key Derivation block derives the decryption key via an approved key derivation algorithm. The decryption key is then passed to the AES-GCM block.

The AES-GCM block authenticates and decrypts the packet. The cleartext portion of the packet from the crypt offset to the start of the ICV is passed to the Merge block, and the computed authentication tag (ICV) is passed to the ICV Check block.

The ICV Check block compares the computed ICV with the ICV in the received PSP trailer. If there is a mismatch, an authentication failure is signaled by dropping the packet and incrementing the corresponding error counter.

The Merge block puts together the pieces of the packet to create the outgoing cleartext packet. Those pieces are the cleartext portion from the Splitter block, and the decrypted ciphertext portion from the AES-GCM block.

Implementation Requirements

The NIC must provide packet counters for authentication and encryption failures, and should provide byte counters for these failures. The NIC must also optionally provide a mechanism to capture a packet that fails authentication or encryption and make it available for inspection by the host. It must be possible to rate limit the packets that are made available to the host for inspection to prevent denial of service attacks.

If the NIC contains packet forwarding logic, such as virtual switch functionality, that operates before PSP encryption, then it must be possible for the forwarding rules to override the transmit descriptor such that selected packets are transmitted without encryption.

RX packet handling requirements include:

- An option to provide the SPI, IV, and VC values from the PSP header to upper-layer software,
- An option to provide a flag indicating that the packet was decrypted/authenticated properly to upper-layer software, and
- An option to provide packets that fail decryption/authentication to upper-layer software without any modifications to the packet.

On TX, an option must be provided that enables upper-level software to provide packets that are already formatted to include the headers required for the PSP encapsulation. In this case, the NIC will modify the contents of the headers appropriately and add the PSP trailer to the packet.

Bad-Data-Injection Defense

This section describes a defense against a “bad-data-injection” attack. More specifically, in the context of PSP, a “misuse of credentials” attack. Consider the case of a PSP-protected TCP socket, S, where a bad actor attempts to inject forged packets into S using a valid SA that is different from the SAs associated with S.

Since the SA is valid, the receiving NIC will decrypt the forged packets. If the NIC does not maintain per-SA state on receive, the NIC cannot directly defend against the attack by checking whether the SA used for decryption is associated with the receiving socket. However, the NIC can participate in the defense by providing the following information, in addition to the decrypted packet, to upper-layer software:

- A flag indicating that the packet was decrypted/authenticated properly, and
- The SPI value from the received packet that was used to perform the decryption/authentication.

This information allows the upper-layer software to drop the packet if the SPI is not on the "approved list” for the receiving socket (the approved list might be initialized when the socket’s TCP connection is established). If the NIC does not remove the PSP header from the packet, then the SPI value is available to upper-layer software in the PSP header; otherwise, the NIC can provide the SPI value as metadata associated with the decrypted packet.

Appendix A - PSP Transmit Key Management

The transmit keys for the various SAs being used at any given time may be provided to the egress classifier in any of a number of ways. There are three primary methods in use today, and others are certainly possible:

1. Keys may be stored in an on-chip egress flow table. In this approach, the NIC's egress classifier is responsible for matching the packet header against a table of known flows, finding the correct flow, and extracting the necessary key from the flow information.
2. Keys may be stored in on-chip, chip-connected, or PCIe-connected "Secure Association" RAM tables. In this approach, the NIC's egress classifier is responsible for fetching the required key from this RAM, based on a key index provided in the packet transmit descriptor.
3. The host may provide a key value on a per-packet basis, by embedding the value of the key into the packet transmit descriptor.

Keys Stored in an On-Chip Per-Flow Table

The first approach is suitable for handling relatively small numbers of flows, limited by the size of the on-chip flow table. Its performance can be quite high, as no off-chip lookups are required per packet. It does not require embedding a key into each packet descriptor, and hence the software constructing the descriptor may not even need to be aware that the connection is encrypted.

This method has been used successfully for carrying PSP traffic over encapsulated tunnels between client virtual machines. A single tunnel (and hence a single key per direction) can carry all of the encapsulated traffic between a pair of client VMs, and the flow table must be updated only infrequently.

Keys Stored in a "Secure Associations" Database in RAM

The second approach is suitable for handling a moderate number of simultaneous flows, limited by the size of the RAM used for the secure associations table. The database must be large enough to hold the keys (and perhaps the SPIs) of every active transmit connection in use at a given time. PSP can have as many as 2^{32} valid SPI/key associations per NIC at a time (worst case) so this can in theory require up to 4 billion table entries. It seems unlikely that NICs will choose to dedicate this much RAM for key storage, and limiting a NIC's "maximum simultaneous connections" specification may be required.

This approach can introduce a significant amount of latency and overhead into the process of setting up and deleting connections, as it is necessary to provision each new connection's key

into the RAM database before transmission can begin, and to reclaim database entries when a connection is torn down and the "time to live" for the connection's packets has expired.

Loading a key from the secure associations database into the egress classifier will take time. The amount of time depends on where the database is stored. This may limit packet transmission throughput. If the secure associations database is split between a fast on-chip cache, and slower off-chip or PCIe-connected memory (e.g., host DRAM), transmit performance may become difficult to predict if the pattern of packet transmission causes "thrashing" in the key cache.

It is possible to amortize the per-connection overhead by sharing SAs; for example, a single SA could be established for traffic sent from $Host_A$ to $Host_B$, and all $Host_A$->$Host_B$ connections could share that SA.

Keys Provided in the Transmit Descriptor

Transmit keys can be provided by the host on a per-packet basis, embedded in or referenced by the packet descriptor. This is the preferred approach for handling large numbers of simultaneous transmit connections, as it requires little or no on-NIC storage for transmit keys - the key for each packet arrives with the packet, is used, and then discarded. There is no specific transmit-key setup/tear-down protocol between host and NIC, hence no overhead or latency when a new PSP connection is established or an old one is retired.

This approach requires that the NIC's transmit-descriptor processing and format must be flexible enough to allow a key to be embedded in the packet descriptor (16 bytes for an AES-GCM-128 key, 32 bytes for an AES-GCM-256 key). The descriptor-processing architecture in some NICs may not be capable of handling such large auxiliary descriptor information.

Alternatively, this approach can also work if the transmit descriptor includes the host memory address of the required key, and a separate DMA operation is used to fetch the key. This would result in a smaller descriptor (e.g., 8 bytes for a 64-bit memory address), but would probably come at the expense of greater latency due to the need for a separate NIC-to-host DMA read.