Notes on implementing IPsec ESP for Snabb

Max Rottenkolber <max@mr.gy>

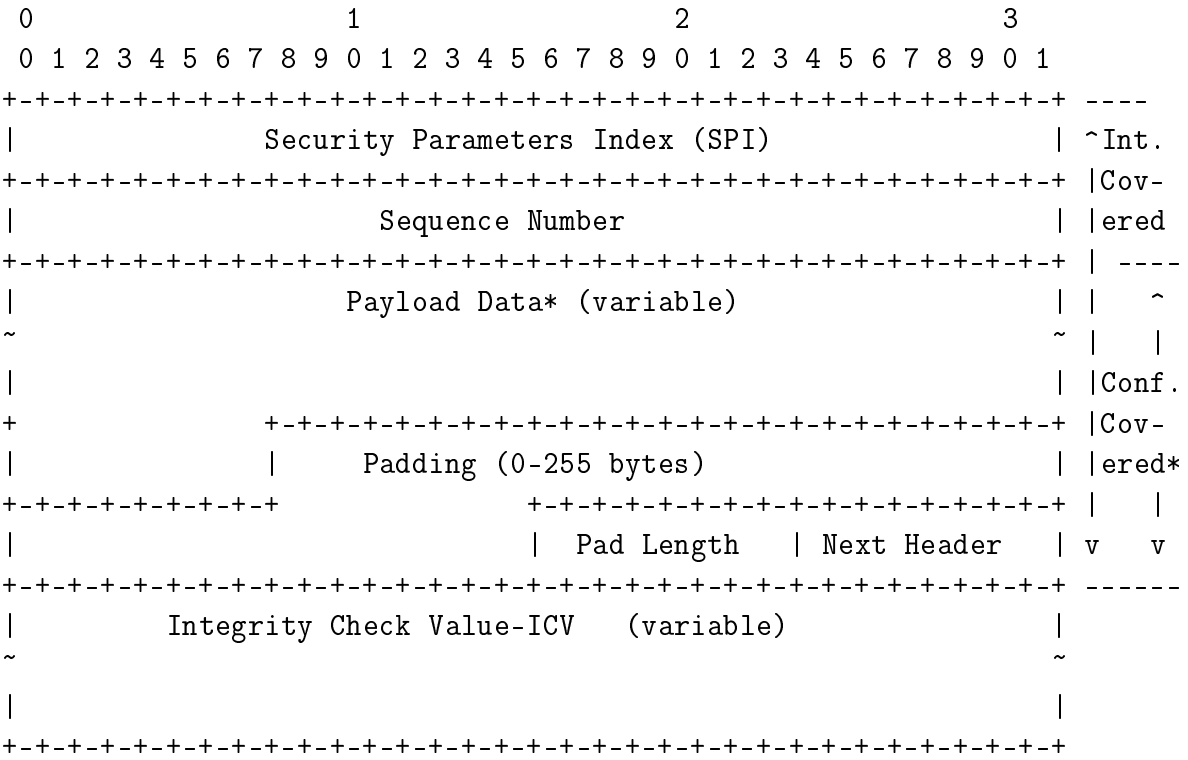
Friday, 27 April 2018

In November 2016, an implementation of *RFC 4303* (https://tools.ietf.org/html/rfc4303) *IP Encapsulating Security Payload* and *RFC 4106* (https://tools.ietf.org/html/rfc4106) *The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)* landed in *Snabb* (https://github.com/snabbco/snabb) for use with Snabb NFV. Since then, the implementation has received many improvements, but the original design is essentially unchanged. Nowadays, it also serves as one of the core pieces of *Vita* (https://github.com/inters/vita#-). In fact, the majority of CPU cycles used by Vita are spent in the ESP implementation, which makes it a prominent target for optimization.

Overview

IP Encapsulating Security Payload (ESP) is a wrapper protocol that provides confidentiality and optional integrity protection over a payload. ESP comes in two variants, *tunnel* mode and *transport* mode. The protected payload can either be an IP frame when used in tunnel mode or

an upper level protocol frame when used in transport mode. In transport mode, the IP source and destination addresses are not protected. Ignoring packet overhead, the two variants are otherwise equivalent.



From RFC 4303: *Figure 1. Top-Level Format of an ESP Packet*

ESP does not mandate a specific encryption mechanism, but instead acts as a container for arbitrary encryption and authentication schemes. The leading *SPI* field is set by the sender to indicate which *Security Association* (SA) a given packet belongs to. The receiver maps the SPI value to a tuple that includes the cipher, key and any other parameters that are required to decrypt the payload data. SAs are ephemeral, and their negotiation is outside of the scope of ESP.

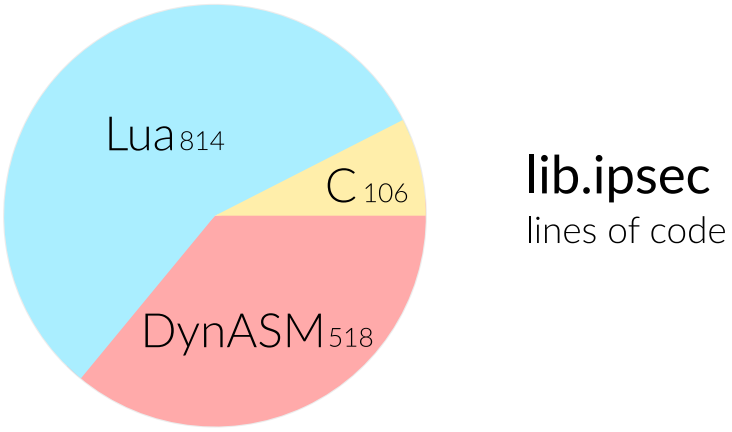
Our
particular implementation provides AES-GCM as the only encryption mechanism, the use of which is described in RFC 4106. AES-GCM is an *AEAD* (https://en.wikipedia.org/wiki/Authenticated_encryption); i.e., it provides *authenticated encryption with associated data*. This is useful because it allows the trailing *ICV* field to become a *message authentication code* (https://en.wikipedia.org/wiki/Message_authentication_code) (MAC). With AES-GCM, the portion of the packet that is indicated to be "Int. Covered" (as in "covered by the ICV") by the diagram above is covered by

a MAC, and can therefore be authenticated by the receiver. By using an AEAD, ESP does not only provide confidentiality but also data origin authentication, rendering a dedicated *Authentication Header* (https://en.wikipedia.org/wiki/IPsec#Authentication_Header) redundant.

The *Sequence Number* field is used to disambiguate packets that are received out-of-order, and as a means to defend against replay attacks (more on that later). The remaining fields, *Padding*, *Pad Length* and *Next Header*, are used to align the beginning of the ESP trailer and indicate the first protocol header that appears in the plaintext respectively.

LuaJIT and its FFI Library

Characteristic of this implementation is that its written in a highly dynamic *LuaJIT* (http://luajit.org/) world, while other IPsec implementations are usually written in and surrounded by C or C++.



LuaJIT's high-quality compiler enables the bulk of the protocol logic to be written in Lua. The ESP header and trailer mangling is performed using LuaJIT's excellent *FFI library* (http://luajit.org/ext_ffi.html). Notably, the FFI library is not used to call out to C code in this particular case. Instead, dealing with protocol headers is made simple by the FFI's language extensions that make handling C structure types in Lua natural.

```
local esp_header_t = ffi.typeof(
   [[struct {
      uint32_t spi;
      uint32_t seq_no;
   } __attribute__((packed))]]
)
local esp_header_ptr_t = ffi.typeof("$ *", esp_header_t)
```

The ESP header can be defined in terms of C structure syntax and semantics.

```
-- ...
local esp_header = ffi.cast(esp_header_ptr_t, ptr)
esp_header.spi = htonl(self.spi)
esp_header.seq_no = htonl(self.seq:low())
```

ESP fields are populated by casting an address in the packet buffer to a structure pointer type.

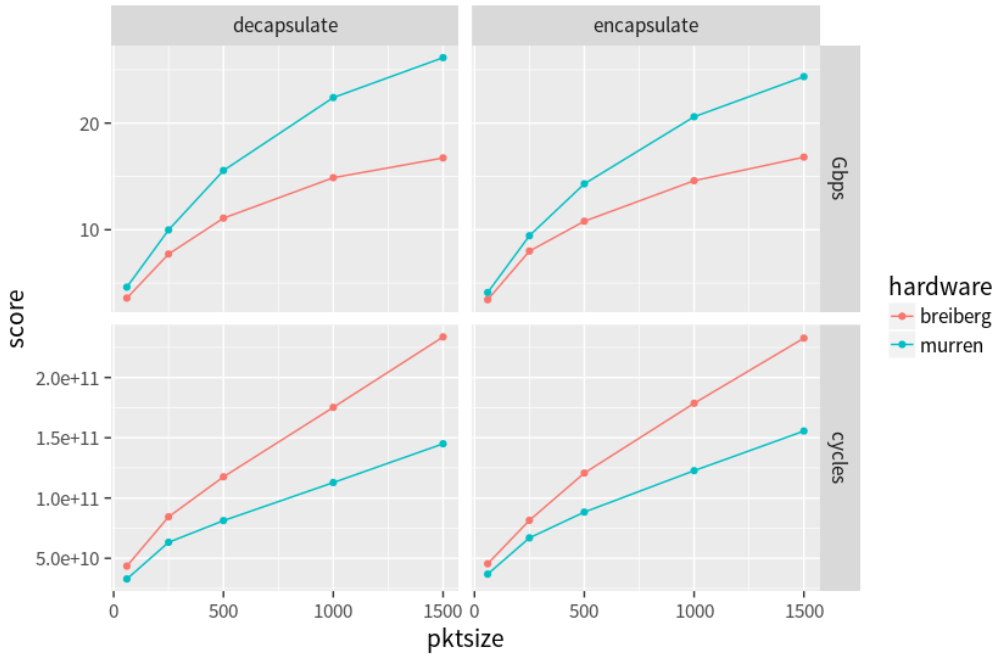
The slice of C in the pie chart above accounts for the code that deals with ESP's anti-replay window (more on that later) written, among other things, by Timo Buhrmester. The anti-replay window is essentially a shifting bit field, and poking around in it seemed most natural in C. In this case, the FFI library is used to call the related C functions.

Embedded Assembler routines using DynASM

The AES-GCM code is based on a reference implementation from Intel, heavily optimized for their fourth generation of *AVX-2* (https://en.wikipedia.org/wiki/Advanced_Vector_Extensions#Advanced_Vector_Extensions_2) capable processors. It uses AVX and *AES-NI* (https://en.wikipedia.org/wiki/AES_instruction_set) extensions and achieves optimal latency by aggressively unrolling loops, encrypting and computing the MAC up to eight blocks at a time. Pete Cawley *translated* (https://github.com/inters/vita/blob/4990df6cb13339e10863d4d0efcc60b6e969b639/src/lib/ipsec/aes_128_gcm_avx.dasl#L3) this implementation to DynASM for use in Snabb, and avoided much of the repetition in the original code using DynASM's "Lua mode", which lets you dynamically assemble code using Lua.

Intel's AES-GCM implementation is really fast, possibly the fastest available for current x86 CPUs. Still, it is an expensive operation, especially on smallish payloads such as network packets. The fact that

AES-GCM throughput gets significantly better with larger payloads only helps to some extent, since our maximum packet size will usually not exceed around 1500 bytes. The good news is that new CPUs seem are still getting better at handling this workload. While Haswell processors ran our ESP encapsulation code at ~1.55 instructions/cycle, the newer Skylake cores do the same work at ~2.33 instructions/cycle (with 1500 byte packets).



Throughput and CPU cycles taken for encapsulation routines over 100 million packets at various packet sizes. *Breiberg*: Xeon E3-1246 v3 (3.5 GHz Haswell) / *Murren*: i7-6700 (3.4 GHz Skylake)

Using DynASM from Lua is neat, it lets you spin out low-level assembler routines that need to be really fast. As an example, consider the assembly code used to verify the ICV. Since the ICV is also a MAC in our case, it is important that the comparison to the should-be value executes in constant time so that it does not leak a timing side-channel. Our implementation only supports a full 16-byte ICV, and can get away with a small piece of simple code.

```
local function auth16_equal(Dst)
   | mov rax, [arg1]
   | mov rdx, [arg1 + 8]
   | xor rax, [arg2]
   | xor rdx, [arg2 + 8]
   | or rax, rdx
   | ret
end

-- ...later, the generated code is cast to a foreign function:
auth16_equal = ffi.cast("uint64_t(*)(uint8_t[16], uint8_t[16])",
                        entry.auth16_equal)
```

X86 code to compare two vectors of 16 bytes in constant time (LuaJIT expects `rax` to contain the first return value.)

Extended sequence numbers

As can be seen in its protocol header, ESP originally specified only 32-bit sequence numbers, which turned out to be too little in the world of today. As a result, conforming ESP implementations must deduce the actual 64-bit "extended sequence number" of each packet from its lower 32 bits, stored in the *Sequence Number* field, and the last sequence number the decapsulator has seen.

Running out of sequence numbers is in itself not much of an issue because SAs should be renewed frequently, but at 100-Gigabit networking the original 32-bit sequence number leaves just short of 30 seconds to do so. Extended sequence numbers alleviate that particular concern definitively (though, one should still change SAs often), but complicate the implementation of ESP significantly, as will hopefully become clear in the remainder of this article.

Replay protection

The primary purpose of the *Sequence Number* field is to ensure that packets are decapsulated only once. If the decapsulator were not to reject previously received sequence numbers an attacker could replay any packet they have captured, and subsequently inject unauthentic traffic. Hence, the encapsulator needs to ensure that each packet within the SA has a unique, monotonically increasing sequence number. (Side note: since

AES-GCM requires an initialization vector (IV) that only needs to satisfy the invariant that it be unique, our implementation uses the sequence number to construct the IV.)

Because packets on the network may be reordered, it is not enough to keep track of the last sequence number that was successfully decapsulated. Instead, the decapsulator maintains the *Anti-replay Window*, a sliding bit field that tracks the received-state of a fixed number of previous sequence numbers. Authentic packets beyond the upper bound of the window cause the bit field to shift upwards while zeroing any invalidated bits. This way it is possible to accept out-of-order packets with some tolerance, while rejecting packets that already had their window bit set or whose sequence number is beyond the lower boundary of the window.

This anti-replay machinery presents, among other things, a challenge when trying to parallelize ESP. Since sequence number verification is a strictly serial process, it would have to be done separately from the decryption step in order to parallelize the latter. Our implementation has no interface to support such a scheme at the time.

Re-synchronization

Given that the ESP header only contains the lower 32 bits of a packet's extended sequence number, the decapsulator needs to guess its upper half based on the next sequence number it is expecting in order to perform the anti-replay check (well, that and to verify the ICV given that the ESN is part of the *additional authentication data* supplied to AES-GCM). Perhaps obviously, this is only possible if both encapsulator and decapsulator are somewhat on the same page as to what sequence number should be next.

Given extended packet loss between encapsulator and decapsulator, and a client protocol that does not detect lack of connectivity (e.g., one-directional UDP) there can arise a situation where the encapsulator's lower half of the sequence number has wrapped around multiple times without the decapsulator noticing. The result is that encapsulator and decapsulator are "out of sync", leaving the decapsulator unable to decrypt any further packets for the SA.

RFC 4303 resolves this issue by introducing a re-synchronization protocol, in which the decapsulator will try to catch up with the encapsulator after it fails to decrypt consecutive packets beyond a given threshold. When it detects such a situation, the decapsulator will attempt to decrypt the next packet using a potential sequence number candidates

by incrementing the guessed half of the number. If decapsulation succeeds it updates its last-seen sequence number accordingly, and voila, encapsulator and decapsulator are back in sync. Fun fact: after failing to decrypt a payload in-place, it is possible to recover the original payload in-place by re-encrypting it with the same (wrong) parameters.

The ugly bit about this is that it leaves us with a tricky code path that will rarely, if ever, be taken in real-world circumstances. Given sufficiently short SA lifespans, it might even be safe to skip this sub-protocol completely. For now, our implementation supports re-synchronization, and suggests defaults for threshold and retries that should guarantee SA durability and at the same time avoid load amplification attacks.

Testing

Besides exhaustive unit tests that exercise the numerous edge cases of RFC 4303 and verify AES-GCM implementation against test vectors, the implementation is tested for interoperability with the Linux networking stack. This is done by hooking it up to a Qemu VM running Linux via *Vhost-User* (http://www.virtualopensystems.com/en/solutions/guides/snabbswitch-qemu/) and talking to its IPsec stack using our implementation of ESP. Like much of the integration testing in Snabb, these tests are orchestrated using *Nix* (https://nixos.org/nix/), which makes managing test dependencies a bliss.

Nix, along with its excellent CI service, Hydra, also drives most of the performance benchmarking, which mostly happens "downstream" in Vita and Snabb NFV. Nix allows us to automate arbitrarily large benchmark campaigns, complete with report generation to quantify performance differences in between software versions and runtime enviri-onments. For instance, the data used to compare ESP performance on Haswell vs Skylake was generated using such an automated campaign.

Interested? Check out Vita (https://github.com/inters/vita#-)*, a high-performance VPN gateway.*