Ephemeral Key Exchange in Vita, part two

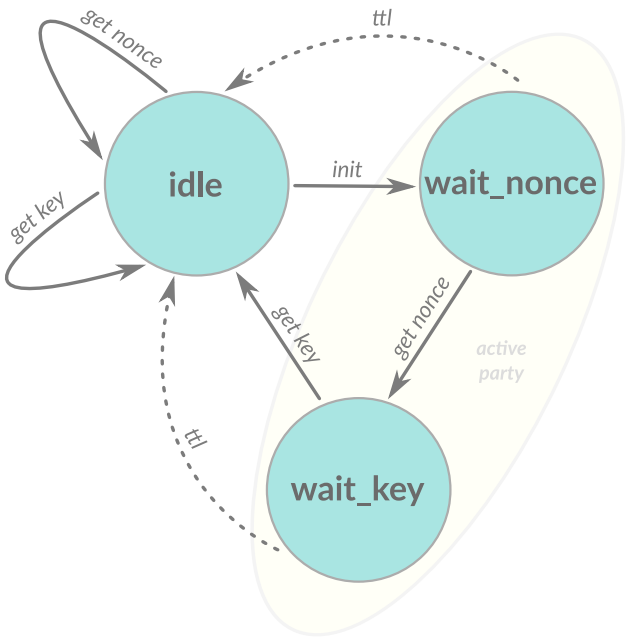
Max Rottenkolber <max@mr.gy>

Monday, 13 May 2019

This article is a follow up to *Ephemeral Key Exchange in Vita, part one* (https://mr.gy/blog/ephemeral-key-exchange.html). As hinted at in the precursor, Vita's authenticated key exchange (AKE) protocol was far from done when the article was written in August 2018. Since then I worked on numerous improvements, and fixed design flaws and bugs. The following paragraphs retrace the steps that lead to the current design, which is based on a *Noise* (http://noiseprotocol.org/) instance.

FSM split into initiator and responder

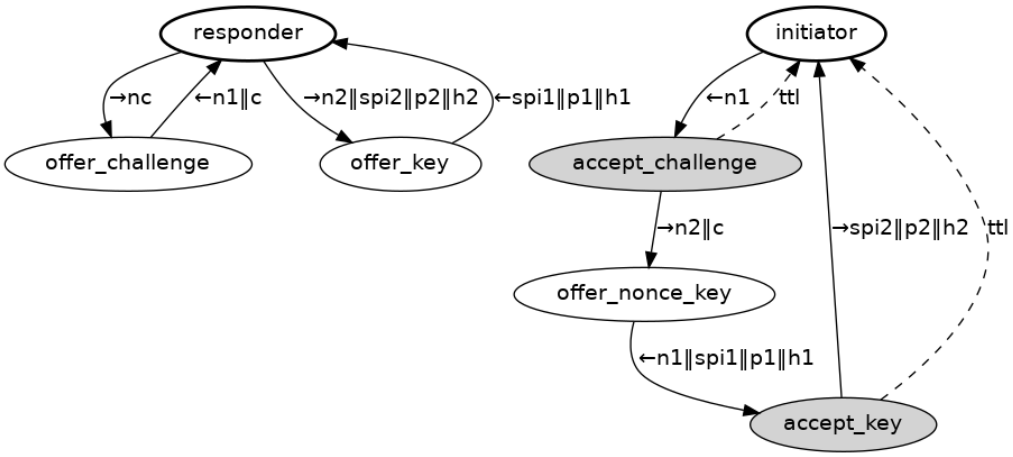
Perhaps the most glaring design defect in the previous AKE protocol iteration was that it had been stubbornly modeled as a single, symmetric finite state machine (FSM). The idea was to allow two nodes to initiate an exchange at the same time, and for them to complete the exchange to produce a single security association (SA) while both parties acted as initiators. This design turned out to be overly ambitious, and led to an avoidable defect. Turns out it also allowed two nodes to interact as responders by accident.



Previous, subtly broken version of the finite state machine (FSM) for Vita's native AKE protocol.

The defect was that if you managed to inject a fake nonce message, you could trick two "idle" participants of the protocol to exchange nonces indefinitely, with each party in the belief that they are acting as the responder in the exchange. Awkward. This was possible due to the fact that a pair of gateways that negotiate SAs for a route would use a hybrid FSM that contains the logic for both initiator and responder, and that the *nonce* message was identical when sent by either an initiating or a responding peer.

The lesson here is that perceived minimalism can easily back-fire. Different state machines should be separate, and different messages should be represented differently. Simplicity in representation does not necessarily equal simplicity in reasoning.



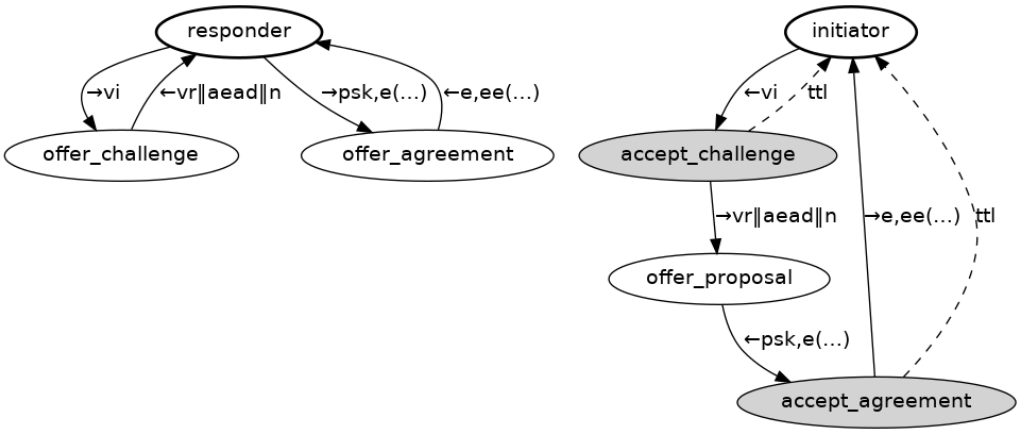
Next iteration of the protocol split into separate initiator and responder FSMs.

Eventually, this was addressed by *splitting the protocol FSM into separate FSMs for initiator and responder* (https://github.com/inters/vita/pull/60). By making sure that the sets of states and messages that form the interaction between initiator and responder are strictly disjoint, the total set of possible FSM interactions is reduced, and the faulty behavior of a responder interacting with another responder is no longer possibile.

Collaborate using a common language

It took me some time to understand the *Noise* (http://noiseprotocol.org/) protocol framework well enough to wield it with confidence, but I eventually managed to *replace the cryptographic core of Vita's AKE protocol with a Noise instance* (https://github.com/inters/vita/pull/76). Shouts to *Justin Cormack* (https://www.cloudatomiclab.com/) for letting me bounce off ideas off him, and answering my questions regarding Noise.

In addition to being a great specification to work with, Noise provides us with a pattern language to express cryptographic protocols. Having a common, shared language allows us to discuss the properties and mechanisms of our protocol's cryptographic core with a wider audience of information security practitioners. Hopefully, this will speed up development, and bolster confidence in our design.



Noise-based protocol FSMs.

Swapping out the cryptographic core with a Noise instance was quite straight forward once I realized that `NNpsk0` was the functional equivalent of the previous core. A `NNpsk0` instance could act as an almost drop-in replacement of our *protocol* (https://github.com/inters/vita/blob/b5f2f577b5a067768a9a0a0ece00d8bddf4cb12c/src/program/vita/README.exchange) core by passing parameters into the Noise prologue. These parameters include static information such as the protocol version, a route identifier, and the identifier of the AEAD to be used with the SA, as well as a random nonce (exchanged beforehand) that protects the protocol from being replayed. The independently chosen security parameter indices (SPI) for the negotiated SAs are protected as payloads of the Noise messages.

Implementing Noise itself was helped by the excellent *Noise Explorer* (https://noiseexplorer.com/), which provides a valuable resource for understanding Noise protocols as well as runnable and readable reference implementations in Go in addition to *ProVerif* (https://prosecco.gforge.inria.fr/personal/bblanche/proverif/) proofs. While I am not too familiar with the Go programming language, for me reading code has always been the surest way to understand how something works.

One initial oversight in the design was brought to light when *Selfie: reflections on TLS 1.3 with PSK* (https://eprint.iacr.org/2019/347) was published not long after. Turns out you could trick our protocol participants to negotiate successfully with themselves, a property used knowingly in Vita benchmarking and test scripts. In order to prevent it from being exploited, *the network addresses of the intended parties of an exchange were added into the prologue* (https://github.com/inters/vita/pull/96).

A minimal set of cryptographic primitives

Sodium (https://github.com/jedisct1/libsodium) maintains a comprehensive set of implementations of cryptographic primitives for a wide range of CPU architectures, and exposes a high-level interface on top of that. While the Sodium library of cryptographic primitives can only be praised, its build-time complexity, and added executable bloat were becoming a drag.

Vita only used a small subset of the provided primitives, only supports `x86_64`, and bypasses the high-level API provided by the Sodium library. Sodium will detect optimal implementations of primitives to build and link with at compile-time, but has no easily accessible means to build only the subset used by Vita. Getting a Sodium object to statically link with required some messing around as well.

After compiling an inventory of primitives needed to implement Noise, and hunting down candidate implementations of these primitives on x86, I replaced the dependency on Sodium by statically linking to a minimal set of primitive implementations of *Curve25519* and *BLAKE2*. The results is two megabytes less of executable size, greatly reduced build time, and a much more transparent build (i.e., visibility into what code actually goes into the artifact.)

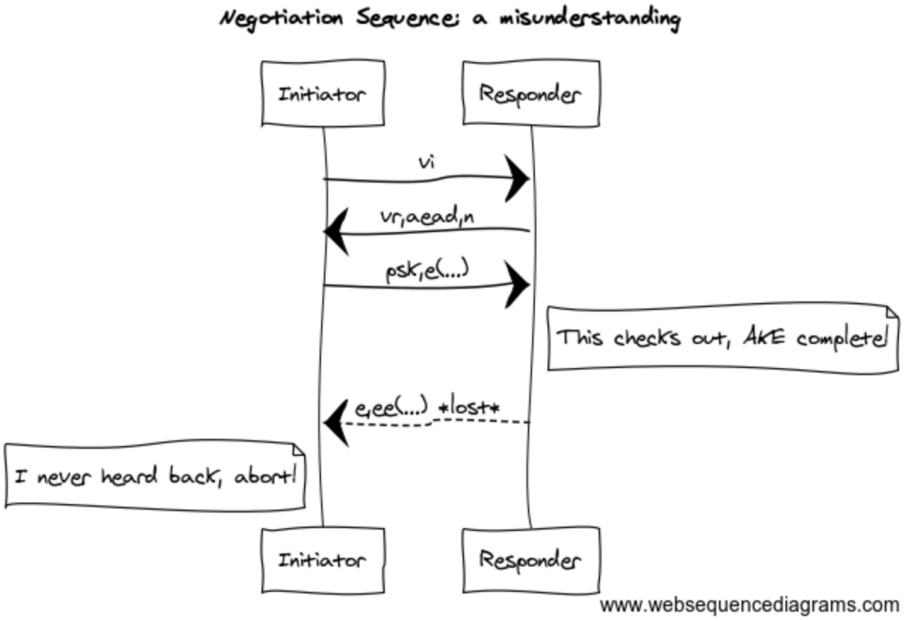
While *replacing Sodium with the official BLAKE2 and the "sandy2x" Curve25519 implementations* (https://github.com/inters/vita/pull/74), I added a ~200 lines of code Lua module (heavily using the LuaJIT FFI) that implements the *HMAC* (https://tools.ietf.org/html/rfc2104) and *HKDF* (https://tools.ietf.org/html/rfc5869) functions based on *BLAKE2s*, in addition to providing a friendly API for using the cryptographic primitives. To provide an *AEAD* (https://en.wikipedia.org/wiki/Authenticated_encryption) suitable for Noise, I *extended our AES-GCM implementation to support AES256 and extended (>12-byte) AAD data* (https://github.com/inters/vita/pull/75).

Seamless SA cycles

Another area in which work has happened is the mechanism by which Vita gateways transition from one SA to the next. How do we cycle the transport credentials without loosing packets during the switch? After all, packets—including the packets that make up the AKE protocol negotiation—can be reordered in transit or lost altogether. One end of the route might consider an exchange to have completed while to other end it timed out. And how long does it take for the other end to activate a

newly negotiated SA? If we transmit packets using the a newly created SA right away the other end might not be ready to decrypt them yet.

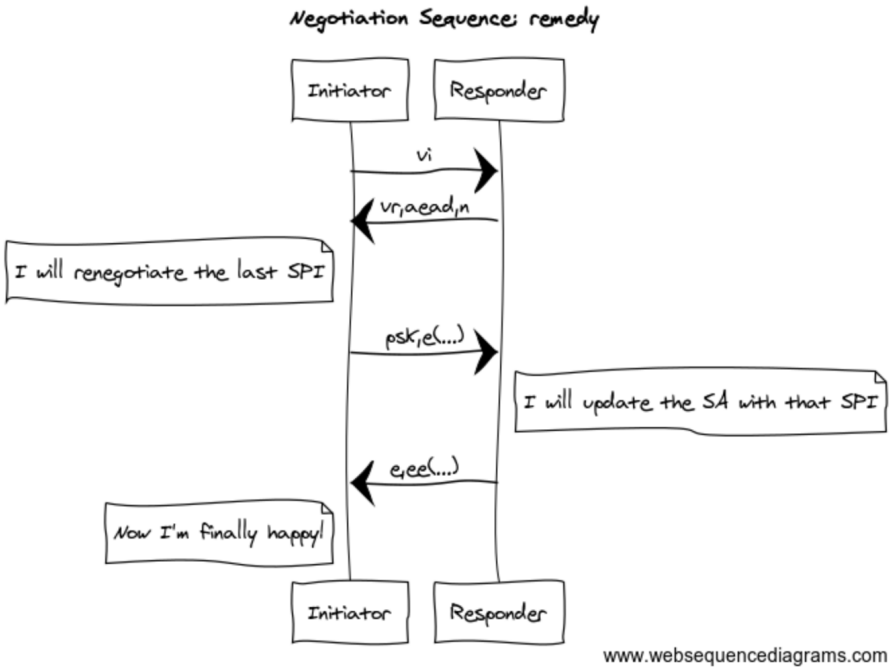
Furthermore, this situation gets slightly more complicated in light of the FSM split into initiator and responder. After all, a node might be the initiator and responder of two AKE negotiations for the same route at the same time, and intern multiple SAs in quick succession only for one of them to be superseded by the other right after.



Sequence diagram of an incomplete negotiation, leaving initiator and responder with different ideas about what happened. The responder ends up installing a new pair of SAs as per the initators request. The initiator, on the other hand, never received the remote part of the *Diffie-Hellman* (https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange) (DH) handshake, and is unable to install the negotiated SA pair.

Part of the solution is to support multiple *concurrent inbound SAs* (https://github.com/inters/vita/pull/47). While there can be only one active outbound SA per route at any given time (we would have no way to tell which one to use if we had more than one), we keep multiple inbound SAs active since we can not always be sure which SA a remote gateway will use to encapsulate the next packets. As a result we maintain a bounded list of active inbound SAs. Newly established inbound SAs get appended to the list and push out old SAs in *FIFO* (https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)) order.

To give a remote gateway a chance to setup the inbound SA matching a newly established outbound SA before we start transmitting on it, there is now a staging area for outbound SAs to be activated. The staged SA replaces the active outbound SA after a delay of `1.5 * negotiation_ttl`. This delay is chosen so that in cases where the final acknowledgment message of an exchange gets lost, the initiating gateway has a chance to initiate a renegotiation and replace the staged outbound SA before it gets activated.



Upon failure, the initiator restarts the negotiation for the previous SPI. The responder will replace the stale outbound SA with the updated SA denoted by the same SPI. For the pair's other SA (i.e., the one from initiator to responder) we just create and activate a new one, leaving both initator and responder with functioning outbound SAs.

What's next

SWITCH engineer Alexander Gall has been working on *integrating the StrongSWAN IKE daemon* (https://github.com/inters/vita/issues/68) for their Layer-2 VPN application. In the future, I hope to support using *IKEv2* with Vita in addition to its existing native AKE protocol and third-party SA management. Might be that we can support an alley for grad-

ual adoption: first replace parts of existing IPsec/IKE infrastructure with Vita, and eventually switch over to a simpler, modern AKE protocol. Stay tuned!