Ephemeral Key Exchange in Vita, part one

Max Rottenkolber <max@mr.gy>

Thursday, 2 August 2018

"Now, the *only* thing that's missing is a mechanism to negotiate and cycle ephemeral keys," I thought to myself, "and I'll have myself a lean, mean IPSec ESP based VPN endpoint." Naiveté can be a blessing—it lets you dive head-first into a subject matter just deep enough that, once you realize the terrors of its subtleties, you are already in too far to turn back.

This article describes my work-in-progress on the research and implementation of an authenticated key exchange (AKE) protocol for use in *Vita* (https://github.com/inters/vita#-) in form of a loose collection of notes. It is prescriptively labeled "part one" in order to signify it as such.

"Don't write your own crypto"

I really thought the type of key exchange issue I had in front of me would be a solved problem, and in a way it is. The existing solutions are just less boring than I expected them to be. In full disbelief mode, I started out by combining cryptographic primitives and pitting the results against requirements. What more could you possibly need than an *AEAD* (https://en.wikipedia.org/wiki/Authenticated_encryption) and a *CPRNG* (https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator)? (Turns out you do not necessarily need any encryption at all in an AKE!)

Needless to say, I came up with a bunch of really broken stuff. During my course, I became familiar with different types of cryptographic primitives and weaknesses, of which I lacked awareness initially. I even caught myself missing weaknesses I was *already aware of*. For instance, having implemented a *protocol* (https://tools.ietf.org/html/rfc4303) with prominent mechanisms against *replay attacks* (https://en.wikipedia.org/wiki/Replay_attack) did not prevent me from devising a protocol vulnerable to such attacks.

My takeaway is that I seem to learn more through missteps than by good guidance. Stepping into each pitfall individually at least once

provided me with invaluable experience and subsequently intuition for the dangers that await when applying and implementing cryptography. To illustrate my point, consider the experiences that led to the *SIGMA* (http://webee.technion.ac.il/~hugo/sigma.html) protocol.

Krawczyk writes, "having discovered the misbinding attack on the 'natural' authenticated *DH* (https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange) protocol BADH, Diffie et al. designed the STS protocol intended to solve this problem" (*SIGMA: the 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols, Hugo Krawczyk, 2003*), and then showcases insecurities to consider in the STS protocol (1992). Notably, SIGMA appears to not have been the last word in key exchange protocols either.

I feel the canonical warning phrase helps create a climate of exclusivity and mysticism. Applied cryptography appears to be a young, vibrant, bleeding-edge field with lots of activity. Encouraging programmers to get some first-hand experience in designing and implementing cryptographic applications, and learning from the inevitable mistakes, seems to improve the state of software security, not impair it. Hence, I would rephrase the mantra as "don't entrust your information security with immature technology."

Problem statement

In order to establish a secure channel via an AEAD, two nodes typically need to exchange a shared key. Naturally, that key then needs to remain confidential between the two nodes exclusively. Further, the exchange must be authenticated. That is, a node needs to ensure that it established a channel to the intended peer, and not an imposter. Authenticity is rather subtle, as will hopefully become clear soon.

The key exchange needs to be automated. In order to limit exposure, best practice mandates that we change keys frequently. I.e., when an ephemeral key is compromised, it should impact past and future communications as little as possible. *ESP with AES-GCM* (snabb-esp.html) specifically, as used by Vita, presents us with the danger of nonce-reuse. That is, regardless of exposure, we need a fresh key whenever we loose track of the counter that sources our nonces (for instance when a node reboots.)

In the context of automation, the subtleties of authenticity surface. For instance, a common defect is a so-called replay-attack. Imagine a message-based key exchange protocol, and that an attacker can record genuine protocol messages and resend them at a later time. If a node

can be made to believe these duplicate messages to be genuine then authenticity is broken.

The SIGMA paper (Krawczyk, 2003) referenced before describes more of such subtle requirements (loosely quoted):

- **aliveness requirement** if A completes a session with peer B then A has a proof that B was "alive" during the execution of the protocol (e.g., by obtaining B's unique authentication on some nonce freshly generated by A)
- **peer awareness** a protocol provides peer awareness for A if when A completes a session with peer B, A has a guarantee that (not only is B alive but) B has initiated a corresponding session with peer A

Another requirement expected of a modern key exchange protocol is that it provides *forward secrecy* (https://en.wikipedia.org/wiki/Forward_secrecy). Assuming a protocol provides confidentiality and authenticity by means of a (a)symmetric preshared keys that were exchanged out-of-band, what happens if any of these mother keys is compromised? In order for forward secrecy to be provided, an attacker should not be able to recover past ephemeral keys from recorded traffic even if they compromise any preshared keys.

Finally, an an aspect often left unmentioned is resilience against *denial of service attacks* (https://en.wikipedia.org/wiki/Denial-of-service_attack). Generally, you want your service to be as robust as possible, but perfect resilience is hard to achieve. While this trait is sometimes not covered by the cryptographic core of a key exchange protocol, it is without doubt a prominent issue in real-world protocol implementations.

(There are additional, desirable traits that are outside the current scope of Vita such as identity protection and *plausible deniability* (https://en.wikipedia.org/wiki/Plausible_deniability). I leave these undiscussed.)

All these requirements only become meaningful in light of an *adversarial model* (also *security model*). I will not go much into formalisms to reason about such models, but let me note that I found *TLS, PACE, and EAC: A Cryptographic View at Modern Key Exchange Protocols, Brzuska et al., 2012* to be a friendly introduction. Basically, an adversarial model describes all the things that could go wrong, or the things an attacker should be considered capable of, that could affect a protocol. With such a model in mind you can then reason about whether a protocol is robust in light of the model's conditions.

A common model is to assume that the communication links over which a message-based protocol is executed are controlled by an attacker. They can thus inspect, drop, and inject messages arbitrarily, but remain bound to the laws of time. E.g., they can not replay a message before a node has sent it. Additionally, it is usually considered that an attacker can reveal any secret key at any point in time. By considering key compromise in the adversarial model, it is possible to reason about forward secrecy, among other things.

One way to think about this is to imagine executing a protocol over secure communication links (i.e., links not influenced by an attacker), and then consider whether it provides equivalent guarantees when executed over links controlled by an attacker. If it is, then the protocol can be considered robust to link compromise. *Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels, Canetti & Krawczyk, 2001* describes this approach in depth.

Bearing gifts

I talked about a vibrant field earlier, let me back that up with some of the neat things available to implementers today. This (incomplete) list includes both useful references for learning, as well as tools to use for implementation and design.

libsodium (https://download.libsodium.org/doc/) provides well documented implementations of curated modern cryptographic primitives. The APIs are totally pleasant to work with via the LuaJIT FFI. The Sodium library encourages you to use wrappers around the actual primitives, but it is trivial to call out to the actual underlying primitives directly, which is useful if you want to be explicit about which primitive to use such as when implementing a protocol. The only complaint I might voice is that it could be made easier to link with statically, maybe even vendor specific parts of the library you might use.

The Sodium library is itself a fork of *NaCl* (http://nacl.cr.yp.to/), and there is also *LibHydrogen* (https://github.com/jedisct1/libhydrogen) which targets constrained environments. Regardless of which option you choose, having a community maintained library of the nitty gritty primitives is a real boon.

The *Noise Protocol Framework* (http://noiseprotocol.org/) is a state-of-the-art framework for secure transport protocols with a beautifully written specification. It is versatile enough to describe a wide range of protocols, called Noise instances, and includes constructs for authenticated key exchange (handshake). Its flexibility can be a overwhelming for a

novice though, and I think it requires the use of a transport layer protocol that provides sessions.

IKEv2 (https://tools.ietf.org/html/rfc7296) is the IPsec key exchange protocol. It is based on the before-mentioned *SIGMA* (http://webee.technion.ac.il/~hugo/sigma.html) and by no means antiquated, having received support for *ECDH* (https://en.wikipedia.org/wiki/Elliptic_curve_Diffie-Hellman) over the course of its long history. It would be the natural choice for Vita for the sake of compatibility with other IPsec implementations. The downside is that it is a rather large specification and hence a bit daunting to implement.

The *spiped protocol* (https://github.com/Tarsnap/spiped/blob/master/DESIGN.md) is a neat little protocol that performs AKE with preshared symmetric keys. It is specific and minimal, which makes it a great starting point for learning about the types of cryptographic primitives it uses. Its author, Colin Percival, was even kind enough to answer the silly *questions* (https://github.com/Tarsnap/spiped/issues/151) I asked on spiped's issue tracker.

Design and implementation

I tend to work in a very iterative fashion. Starting with a stub, each iteration adds new features or satisfies more requirements, and grows the design and implementation in lock-step towards a usable state. If at any point either implementation or design turn out to be intractable I cut my losses and start fresh, armed with the experience gathered in course of the previous failures.

When I first looked at IKEv2 (the natural choice for interoperability) I felt overwhelmed, and was doubtful if I could implement a specification this large without loosing oversight. The next contender was the Noise framework, but it did not support the stubborn design I had in mind (using long-lived, preshared, symmetric keys per route), and presented me with choices I did not feel qualified to consider with my limited experience. Still, the Noise specification provided invaluable ideas for how I might tackle the eventual implementation of such a protocol, and as such proved a great resource regardless.

I learn by doing, so the minimal spiped protocol came in handy as its AKE phase did most of what I thought I needed, and adapting it to use primitives from libsodium seemed doable. A good starting point to study and learn from.

The current iteration will not be the last, but here is how Vita's AKE looks right now. I wanted the operation of Vita to be as sim-

ple as possible, and decided to go with one symmetric, preshared key per route as opposed to asymmetric *Curve25519* (https://en.wikipedia.org/wiki/Curve25519) key pairs which seems to be the current recommended way to do things.

```
route {
  id site2;
  net-cidr4 172.16.2.0/24;
  gw-ip4 203.0.113.2;
  preshared-key 91440DA06376A668AC4959A840A125D75FB544E8A...;
  spi 1001;
}
```

Configuration (https://github.com/inters/vita/blob/203971b/src/program/vita/README.config) of a route to `site2`: packets destined to the subnet `172.16.2.0/24` will be routed to the Vita node at `203.0.113.2` via a tunnel authenticated using the preshared, symmetric 256-bit key identified by the security parameter index 1001. The Vita node at `site2` would configure a matching route with identical `preshared-key` and `spi` values.

The choice of symmetric, preshared keys warrants an explanation. The obvious benefit of asymmetric key pairs is that they require less keys in total (a key pair per node, as opposed to a key per route.) Another benefit is that you can transmit public keys in the clear as long as they are authenticated. In the scope of Vita as a site-to-site VPN endpoint, I reasoned that routes will be limited and static in nature (hence, no particular use for a *PKI* (https://en.wikipedia.org/wiki/Public_key_infrastructure)). One key affects one route (think door/gate) seems like a familiar mental model.

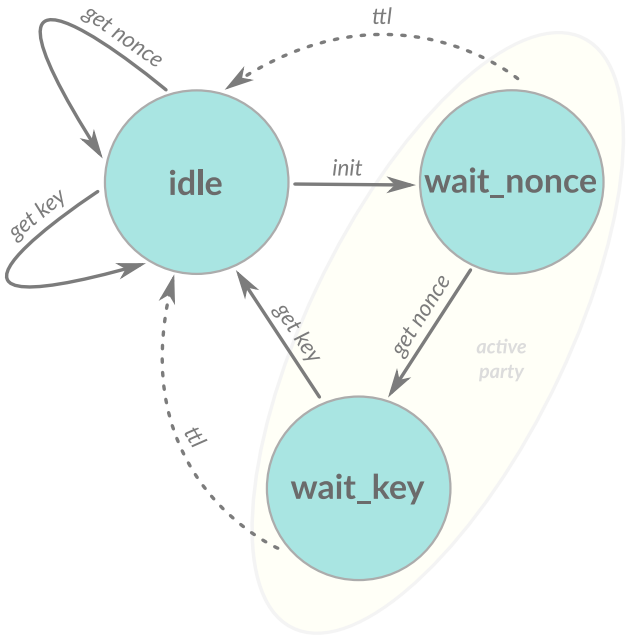
Second, I believe secrecy as a concept is better understood by the average layperson than the often subtle concept of authenticity. If they understand that they need to distribute a valuable secret between two sites that want to establish a secure route, they might fall back to sending a courier in person. Simple. With public keys there is always the temptation to exchange them via email. Tricky. I am still torn on this subject. The final iteration on Vita's AKE may very well use asymmetric key pairs. For now, I enjoy the simplicity of implementation and ease of key generation: read 256 bits from `/dev/urandom`.

I managed to derive a *cryptographic core* (https://github.com/inters/vita/blob/203971b/src/program/vita/README.exchange) for Vita's AKE from the spiped protocol that uses just three primitives: an *HMAC* (https://en.wikipedia.org/wiki/HMAC), a *DH* (https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange), and a cryptographic *HASH*

(https://en.wikipedia.org/wiki/Cryptographic_hash_function) function. Its two round trips can be summarized as follows:

- The parties start by exchanging sufficiently big and hence unique nonces (these are later used to ensure that messages from prior executions of the protocol can not be replayed or mixed up into this execution.)
- They then exchange the parameters required for a DH exchange (providing forward secrecy because the exchanged secret is never transmitted.) This second message is authenticated with the preshared key via a HMAC over the SPI for the route (to prevent misbinding), the nonces (to prove that the message belongs to this execution instance of the protocol), and the DH parameters themselves.
- Finally, each node derives the required key material from the shared secret using the cryptographic HASH function.

What I like about this protocol is that it is minimal (every operation has a single specific function), and that it can be reasoned about based on the properties of the individual primitives. At the expense of interoperability, each version of this protocol as used in Vita will be instantiated using an explicit set of primitives (currently from libsodium).



What is still missing is how this protocol actually gets transmitted over the wire, and how we protect it against denial of service attacks.

The neat way to build upon a cryptographic core is to compose it with additional layers of protocols. *Principles of Protocol Design* by Sharp is a really neat resource on protocol algebra that I, regrettably, still grok too little about.

Being anxious about the exploitability of lower protocol layers, I decided to keep it really minimal for the start. At the moment, Vita maintains a single state machine for each route, and encapsulates the cryptographic core with the bare minimum of a transport header that encodes a message type (is this a nonce, or is it a DH offer?) and an SPI to match messages to the correct state machine. I made sure than an attacker has no vector to trigger allocations, or any other meaningful state change in a node.

What remains though, is the ability for an attacker to inhibit key exchanges by spoofing bogus nonces with the right timing. I felt this was an acceptable trade-off between simplicity and resilience, reasonably close to the unavoidable vulnerability to a brute force DoS or an active attacker blocking link connectivity. Though, I think you can do better.

As to the *implementation* (https://github.com/inters/vita/blob/203971b/src/program/vita/exchange.lua#L5-L128), I took some hints from the Noise protocol framework to ensure a robust design. The cryptographic core is implemented as a finite state machine (FSM) with a black box interface designed for misuse resistance. The idea is that as long as you stick to its API it should be impossible to cause fatal failure.

```
initiate_exchange(nonce_message) → error?, [nonce_message]
receive_nonce(nonce_message) → error?, [nonce_response]
exchange_key(key_message) → error?, [key_message]
receive_key(key_message) → error?, [key_response]
derive_ephemeral_keys() → error?, [inbound-sa, outbound-sa]
reset_if_expired() → reset?
```

Misuse-resistant interface of the cryptographic core FSM. The caller allocates the objects operated on by the core, which only returns them on success along with an error indicator that signifies what, if anything, went wrong.

This technique appears to be a really effective way to maintain separation of concerns and to avoid bugs. The caller needs to know about the FSM transitions and what to do in error situations, but does not need to bother about invariants the FSM is designed to uphold. If an API call returns an object then it is safe to use it. Otherwise, the type of failure will be indicated by the error code.

One remaining danger is memory corruption. Since we are using the LuaJIT FFI to call to libsodium and to fiddle with packet headers there is always some uncertainty about what might go wrong if write a byte too much. I am hopeful what this situation can be *improved* (https://github.com/raptorjit/raptorjit/issues/156) in RaptorJIT by harnessing the powers of runtime type information on foreign pointers. "Details to be determined!"

What is next? Obviously, work on this is far from done, and I am certain there are plenty of bugs left to discover. I am *currently working on* (https://github.com/inters/vita/pull/47) concurrent security associations (SA) per route in order to support seamless rekeying (so packets in-flight during a rekeying of the SA can still be decapsulated once the new SA is swapped in.) Due to experience gained from implementing the current state of affairs, I also feel more confident that I could pull of implementing IKEv2 for Vita, or alternatively instantiate a Noise protocol instead. Though, I have not made up my mind on which way is the way to go, yet.

All help appreciated

I am always looking for interested people who would like to take a critical look at Vita, or who would want to contribute in any way, be it via code or discussion. If that person is you, I am looking forward to hearing from you!