ALPHA

Johannes Gilger, Florian Weingarten

January 16th, 2009

Adaptive and Lightweight Protocol
for Hop-By-Hop Authentication

Summary

What we did since last meeting

- packet queue
- hash chain framework (using the `openssl` library)
- alpha signature scheme (similar to initial handshake) for **every** packet
- some minor TODOs (fix some memory leaks, S1/SYN timeout-retransmit, ...)

Problems

- Deadlock situation while using alpha in full-duplex (easy to fix)

What we did since last meeting

- packet queue
- hash chain framework (using the `openssl` library)
- alpha signature scheme (similar to initial handshake) for **every** packet
- some minor TODOs (fix some memory leaks, S1/SYN timeout-retransmit, ...)

Problems

- Deadlock situation while using alpha in full-duplex (easy to fix)

What we did since last meeting

- packet queue
- hash chain framework (using the `openssl` library)
- alpha signature scheme (similar to initial handshake) for **every** packet
- some minor TODOs (fix some memory leaks, S1/SYN timeout-retransmit, ...)

Problems

- Deadlock situation while using alpha in full-duplex (easy to fix)

Why we need a queue?

- don’t want to lose packets sent before the handshake was done
- multiplex different clients (only one tun device), so we have to store packets in case a client is in a not-ready state
- we need a packet buffer for the more advanced alpha modes

Problems

- none, easy to do in C

Why we need a queue?

- don't want to lose packets sent before the handshake was done
- multiplex different clients (only one tun device), so we have to store packets in case a client is in a not-ready state
- we need a packet buffer for the more advanced alpha modes

Problems

- none, easy to do in C

Why we need a queue?

- don't want to lose packets sent before the handshake was done
- multiplex different clients (only one tun device), so we have to store packets in case a client is in a not-ready state
- we need a packet buffer for the more advanced alpha modes

Problems

- none, easy to do in C

OpenSSL

- we need some crypto, especially one-way hash functions (SHA1, MD5, ...)
- we do not want to implement them (ever read the MD5 RFC? :-))
- OpenSSL is quite widespread (portability!) and performance-optimized

How it looks

SHA1 (Secure Hash Algorithm, 160 bit output), pretty straight-forward

```
#include <openssl/sha.h>
...
char data[];
char hash[SHA_DIGEST_LENGTH];
...
SHA1(data, sizeof(data), hash);
```

OpenSSL

- we need some crypto, especially one-way hash functions (SHA1, MD5, ...)
- we do not want to implement them (ever read the MD5 RFC? :-))
- OpenSSL is quite widespread (portability!) and performance-optimized

How it looks

SHA1 (Secure Hash Algorithm, 160 bit output), pretty straight-forward

```
#include <openssl/sha.h>
...
char data[];
char hash[SHA_DIGEST_LENGTH];
...
SHA1(data, sizeof(data), hash);
```

OpenSSL

- we need some crypto, especially one-way hash functions (SHA1, MD5, ...)
- we do not want to implement them (ever read the MD5 RFC? :-))
- OpenSSL is quite widespread (portability!) and performance-optimized

How it looks

SHA1 (Secure Hash Algorithm, 160 bit output), pretty straight-forward

```
#include <openssl/sha.h>
...
char data[];
char hash[SHA_DIGEST_LENGTH];
...
SHA1(data, sizeof(data), hash);
```

Hash chains

What is a hash chain?

- cryptographic one-way hash function h (for example SHA1)
- some secret s called *seed*, some number n
- list $(h(s), h(h(s)), ..., h^n(s))$ is called *hash chain* (of length n with seed s)

Problems/Questions

- n will be large (suppose > 100000)
- each element will be 160 bit (`SHA_DIGEST_LENGTH`)
- we want to handle multiple clients (each client needs two hash chains)
- $\Rightarrow$ at least 38 megabyte (for each endpoint!) (Nokia has 128 MB total!)
- Save the whole chain or save just the seed? (performance vs. memory)
- Save every kth element as „temporary seed"?
- ...

What is a hash chain?

- cryptographic one-way hash function h (for example SHA1)
- some secret s called *seed*, some number n
- list $(h(s), h(h(s)), ..., h^n(s))$ is called *hash chain* (of length n with seed s)

Problems/Questions

- n will be large (suppose > 100000)
- each element will be 160 bit (`SHA_DIGEST_LENGTH`)
- we want to handle multiple clients (each client needs two hash chains)
- $\Rightarrow$ at least 38 megabyte (for each endpoint!) (Nokia has 128 MB total!)
- Save the whole chain or save just the seed? (performance vs. memory)
- Save every kth element as „temporary seed“?
- ...

What is a hash chain?

- cryptographic one-way hash function h (for example SHA1)
- some secret s called *seed*, some number n
- list $(h(s), h(h(s)), \dots, h^n(s))$ is called *hash chain* (of length n with seed s)

Problems/Questions

- n will be large (suppose > 100000)
- each element will be 160 bit (`SHA_DIGEST_LENGTH`)
- we want to handle multiple clients (each client needs two hash chains)
- $\Rightarrow$ at least 38 megabyte (for each endpoint!) (Nokia has 128 MB total!)
- Save the whole chain or save just the seed? (performance vs. memory)
- Save every kth element as „temporary seed“?
- ...

The simplex situation

- $\xrightarrow{S_1^p}$
- $\xleftarrow{A_1^p}$
- $\xrightarrow{S_2^p}$

The good duplex situation

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{S_2^{p1}}$
- $\xleftarrow{S_1^{p2}}$
- $\xrightarrow{A_1^{p2}}$
- $\xleftarrow{S_2^{p2}}$

... and the bad one

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{S_1^{p2}}$ **BAM!**
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{A_1^{p2}}$
- $\xrightarrow{S_2^{p1}}$
- $\xleftarrow{S_2^{p2}}$

Solution

- use **two** state variables
- one for sending, one for receiving
- $\Rightarrow$ no more deadlocks possible

The simplex situation

- $\xrightarrow{S_1^p}$
- $\xleftarrow{A_1^p}$
- $\xrightarrow{S_2^p}$

The good duplex situation

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{S_2^{p1}}$
- $\xleftarrow{S_1^{p2}}$
- $\xrightarrow{A_1^{p2}}$
- $\xleftarrow{S_2^{p2}}$

... and the bad one

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{S_1^{p2}}$ **BAM!**
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{A_1^{p2}}$
- $\xrightarrow{S_2^{p1}}$
- $\xleftarrow{S_2^{p2}}$

Solution

- use **two** state variables
- one for sending, one for receiving
- $\Rightarrow$ no more deadlocks possible

The simplex situation

- $\xrightarrow{S_1^p}$
- $\xleftarrow{A_1^p}$
- $\xrightarrow{S_2^p}$

The good duplex situation

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{S_2^{p1}}$

- $\xleftarrow{S_1^{p2}}$
- $\xrightarrow{A_1^{p2}}$
- $\xleftarrow{S_2^{p2}}$

... and the bad one

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{S_1^{p2}}$ **BAM!**
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{A_1^{p2}}$
- $\xrightarrow{S_2^{p1}}$
- $\xleftarrow{S_2^{p2}}$

Solution

- use **two** state variables
- one for sending, one for receiving
- $\Rightarrow$ no more deadlocks possible

The simplex situation

- $\xrightarrow{S_1^p}$
- $\xleftarrow{A_1^p}$
- $\xrightarrow{S_2^p}$

The good duplex situation

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{S_2^{p1}}$
- $\xleftarrow{S_1^{p2}}$
- $\xrightarrow{A_1^{p2}}$
- $\xleftarrow{S_2^{p2}}$

... and the bad one

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{S_1^{p2}}$ **BAM!**
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{A_1^{p2}}$
- $\xrightarrow{S_2^{p1}}$
- $\xleftarrow{S_2^{p2}}$

Solution

- use **two** state variables
- one for sending, one for receiving
- $\Rightarrow$ no more deadlocks possible

The simplex situation

- $\xrightarrow{S_1^p}$
- $\xleftarrow{A_1^p}$
- $\xrightarrow{S_2^p}$

The good duplex situation

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{S_2^{p1}}$

- $\xleftarrow{S_1^{p2}}$
- $\xrightarrow{A_1^{p2}}$
- $\xleftarrow{S_2^{p2}}$

... and the bad one

- $\xrightarrow{S_1^{p1}}$
- $\xleftarrow{S_1^{p2}}$ **BAM!**
- $\xleftarrow{A_1^{p1}}$
- $\xrightarrow{A_1^{p2}}$
- $\xrightarrow{S_2^{p1}}$
- $\xleftarrow{S_2^{p2}}$

Solution

- use **two** state variables
- one for sending, one for receiving
- $\Rightarrow$ no more deadlocks possible

Goals

What we want to (or could) do next

- Finish implementing all integrity checks
- Implement some tools / evil-mode for testing this integrity checks
- Start reading up (and maybe implement) the more advanced alpha modes
- Setup some more virtual machines
- Implement an `ipqueue` filter for routers between alpha nodes

What we want to (or could) do next

- Finish implementing all integrity checks
- Implement some tools / evil-mode for testing this integrity checks
- Start reading up (and maybe implement) the more advanced alpha modes
- Setup some more virtual machines
- Implement an `ipqueue` filter for routers between alpha nodes