Perfect Forward Security
An extra layer of security and privacy

Pratik Guha Sarkar — psarkar@isecpartners.com

iSEC Partners, Inc
123 Mission Street, Suite 1020
San Francisco, CA 94105
https://www.isecpartners.com

August 31, 2014

Abstract

Disclosure of state sponsored monitoring of electronic communications and the threat of retroactive decryption of traffic of millions of people have created an urge for an extra layer of security and privacy for all electronic communications. The purpose of this paper is to survey Perfect Forward Security — invented more than twenty years ago — as the solution to this problem.

1 Introduction

Millions of websites and billions of people rely on Transport Layer Security (TLS), IPSec, VPN software, and similar protocols to protect the electronic transmission of sensitive and personal information with the expectation that encryption guarantees security and privacy. The difficulty of cryptanalysis of the encryption algorithms used in these protocols ensures the security of the encrypted messages. The cryptographic algorithms often have a single point of failure, however. Today anybody can record an encrypted, or unreadable, communication in transit. Later, when the keys used have been compromised, or asymmetric cryptanalysis has advanced sufficiently to break them, an adversary can retroactively decrypt today's traffic.

The terms 'Forward Security' and 'Perfect Forward Security' have evolved slightly in colloquial use since Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener invented a two-party mutual authentication protocol and coined the term twenty years ago.[1] Their protocol provides authenticated key exchange, focused on using asymmetric techniques. They defined PFS as: *"An authenticated key exchange protocol provides perfect forward secrecy if disclosure of long-term secret keying material does not compromise the secrecy of the exchanged keys from earlier runs."*[2]

Today, however, the terms 'Forward Security' (FS), 'Perfect Forward Security' (PFS), and even 'Future Secrecy'[3] have evolved slightly to imply something different. In addition to fully explaining the nuanced differences between FS, PFS, and Future Secrecy, this paper will cover a simple explanation, real life applications, advantages, and implementation of PFS in different protocols, including Transport Layer Security (TLS), Off-the-record (OTR) Messaging, Secure Shell (SSH), Wireless Protected Access II Protocol (WPA2 EAP-PWD), Internet Protocol Security (IPSec) and Virtual Private Networking (VPN).

[1] "Authentication and Authenticated Key Exchanges" : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.6682&rep=rep1&type=pdf

[2] Section 4: Desirable Protocol Characteristics http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.6682&rep=rep1&type=pdf

[3] Coined by Moxie Marlinspike and Trevor Perrin in https://whispersystems.org/blog/advanced-ratcheting/

Note: The terms Perfect Forward Security and Perfect Forward Secrecy are used interchangeably in practice. In this paper, we will only use "Security", but in common literature (including the original paper) you should expect to find "Secrecy" as well. The term "Future Secrecy" is not commonly used, but in this paper we will use the term to denote a specific property we will define shortly.

2 What is (Perfect) Forward Security

Forward Security is a characteristic of an authenticated key exchange protocol which ensures that the disclosure of a long term identity key (such as a SSL Certificate or a SSH Host Key) does not compromise the confidentiality of the messages encrypted in sessions prior to the compromise. The session keys, or the short term encryption keys, are independent of the long term identity keys — although these session keys are authenticated by the identity keys.

Each protocol operates differently, but in general, a session key is not used in subsequent runs of the protocol, so it is feasible to destroy it after each session. Destroying the key does not disrupt any other communication and once these short term keys are destroyed, it is not possible to decrypt any ciphertext encrypted under these keys. Thus Forward Security provides a layer of defense against the retroactive decryption of sessions in case of compromise of long term identity keys.

2.1 PFS vs. FS

Forward Security has been used as a synonym for Perfect Forward Security but there is a subtle difference between the two. The difference relates specifically to the compromise of a *session* key — not a long term identity key.

In Forward Security, compromise of a session key allows retroactive decryption of prior sessions. Perfect Forward Security has the additional property that compromise of a session key *does not* allow compromise of prior sessions.

2.2 Future Sessions

The subtle different of Forward Security vs Perfect Forward Security applies only to the compromise of *session keys*, not long term identity keys. It also applies only to attacking *prior* sessions. The term "Future Secrecy" was coined to describe what happens to *future* sessions if a *session key* is disclosed.

But even with FS, PFS, and Future Secrecy — there is nothing to describe how a protocol behaves to *future sessions* if a *long term identity key* is disclosed. We can summarize these terms with the following table:

Scenario	Protocol with Forward Security	Protocol with Perfect Forward Security	Protocol with Future Secrecy
Compromise of Identity Key Attacking Prior Sessions	**Secure (By Definition)**	**Secure (By Definition)**	**Depends on Protocol**
Compromise of Identity Key Attacking Future Sessions	**Depends on Protocol**	**Depends on Protocol**	**Depends on Protocol**
Compromise of Session Key Attacking Prior Sessions	**Insecure (By Definition)**	**Secure (By Definition)**	**Depends on Protocol**
Compromise of Session Key Attacking Future Sessions	**Depends on Protocol**	**Depends on Protocol**	**Secure (By Definition)**

3 How to achieve PFS

This section will look into how a generic protocol can achieve Perfect Forward Security and illustrate how one could build protocols that do or don't provide security from some of the attacks outlined above. It will go over the key requirements, message flows, and demonstrate the resistance or vulnerability of the protocol to key compromise. These designs are trivial examples intended to illustrate forward security properties — certain aspects are considered out of scope: for example, no care is given to long-term key authentication, revocation or replay attacks.

Because so many protocols are built on top of the Diffie-Hellman Key Exchange, to arrive at a session key, we will briefly describe this key exchange in both the ordinary and Elliptic Curve variants.

3.1 Diffie-Hellman key exchange protocol

Diffie-Hellman key exchange[4] is an implementation of public-key cryptography, designed to arrive at a shared secret among two parties, over an insecure public medium without sharing anything beforehand. In this algorithm, there are three parameters (also known as DH parameters):

- p — a very large prime number
- g — a primitive root[5] modulo p, also known as generator of the multiplicative group of integers[6] modulo p; such that $g < p$
- x — a private key

For each number n, there is a power x of g such that $n \equiv g^x \bmod p$. This is the public key. Now from the knowledge of x, g and n, each party can calculate the shared secret key s. Both parties keep their x and s as secret, while all the other values — p, g and n are shared over insecure channel.

Let's consider our very old friends Alice and Bob, who want to communicate securely. The process starts with each of them agreeing on publicly shared p and g. Alice chooses a random integer a as her private key and calculates her public key as $n_a \equiv g^a \bmod p$. Similarly, Bob chooses b and computes his public key as $n_b \equiv g^b \bmod p$. Both of them exchange their public keys n_a and n_b. Finally, to determine the shared secret key (symmetric key), Alice computes $s \equiv (n_b)^a \bmod p$ and Bob computes $s \equiv (n_a)^b \bmod p$.

Both Alice and Bob have arrived at the same value, because $(g^a)^b \bmod p$ and $(g^b)^a \bmod p$ are equal. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.

There is no efficient algorithm to solve the discrete logarithm problem,[7] which would make it easier for adversary to compute x from the knowledge of p, g, and n and to solve the Diffie–Hellman problem.[8] So the security of the protocol is as strong as the difficulty to efficiently solve discrete logarithm problem for very large prime numbers.

3.2 Elliptic curve cryptography

Elliptical curve cryptography[9] (ECC) is a public key encryption technique based on elliptic curve[10] theory that can be used to create faster, smaller, and more efficient cryptographic keys. ECC generates keys through the properties of the elliptic curve equation, instead of the traditional methods like multiplying two very large prime numbers.

[4] http://www.ietf.org/rfc/rfc2631.txt
[5] http://math.arizona.edu/~savitt/mathcamp/1999/primitive_roots.pdf
[6] http://en.wikipedia.org/wiki/Multiplicative_group_of_integers_modulo_n
[7] http://en.wikipedia.org/wiki/Discrete_logarithm_problem
[8] http://en.wikipedia.org/wiki/Diffie-Hellman_problem
[9] http://tools.ietf.org/html/rfc6090
[10] http://en.wikipedia.org/wiki/Elliptic_curve

ECC is based on properties of a particular type of equation created from the mathematical group[11] derived from points where the line intersects the axes.

An elliptic curve is a set of points (x,y) on a plane which satisfy an equation of the form $y^2 = x^3 + ax + b$ (where a and b are constants), together with an extra point o which is called the point at infinity. For applications to cryptography we consider finite field[12] or Galois field of q elements (where q is a finite set of integers modulo a prime number), which is represented as F_q or $GF(q)$.[13]

Once again, Alice and Bob want to communicate securely using ECC. They agree on a (non-secret) elliptic curve and a (non-secret) fixed curve point F_q. Alice chooses a secret random integer k_a which is her secret key, and publishes the curve point $P_a = k_a * F_q$ as her public key. Bob does the same. His secret key is k_b and curve point is $P_b = k_b * F_q$. Now to send encrypted messages, Alice can simply compute the shared secret key $s = k_a * P_b$. Bob can compute the same number s by calculating $k_b * P_a$, since $k_b * P_a = k_b * (k_a * F_q) = (k_b * k_a) * F_q = k_a * (k_b * F_q) = k_a * P_b$.

The security of the scheme is based on the assumption that it is difficult to compute k given F_q and $k * F_q$. ECC can provide a higher level of security with a smaller key, hence with lower computing power than other cryptosystems. For elliptic-curve-based protocols, it is assumed that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is infeasible — this is known as the Elliptic Curve Discrete Logarithm problem.

3.3 Toy Protocols

Any secure communication protocol has authentication, confidentiality, and integrity of the message as its properties. The implementation specifics of these properties can vary depending on the requirement of the protocol. Our toy protocols aim to achieve these properties as well.

Authentication ensures that two parties are communicating to the intended recipient over a public channel. One can achieve authentication using Public-Key Infrastructure and Certificate Authorities or by sharing public keys of both the parties beforehand. The protocol designer can achieve confidentiality by using a strong encryption algorithm. The choice of symmetric or asymmetric encryption algorithm depends on the purpose and efficiency of the implementation. Use of a message authentication code (MAC) is one way to validate the integrity of the message.

Now we have the base framework of a secure communication protocol: let's use examples to illustrate the Forward Security and Future Secrecy properties. Alice and Bob want to communicate with each other, and somehow have established trust in each other's long term identity keys.

3.3.1 No Forward Security

Alice creates a random symmetric key, encrypts it with Bob's public key and then transfers it to Bob. Bob decrypts the symmetric key, and both start using the symmetric key to encrypt communication for the rest of the session. When they close the session and make a new one, it repeats as before with a new key.

If either of their identity keys gets compromised, then all the past communications are compromised, as that key can be used to decrypt all the symmetric keys encrypted under it. Identity key compromise also allows attacks on all future sessions as well.

[11] A Group is a set of values for which operations $(*)$ can be performed on any two members of the group to produce a third member while satisfying the group axioms like closure, associativity, identity, and invertibility. More on Group — https://en.wikipedia.org/wiki/Group_(mathematics)

[12] The requirement for a set of elements to be in a field is to have operations like addition, subtraction, multiplication, and division — these operations always produce a result that is in the field, with the exception of division by zero, which is undefined. More about the Finite field: https://en.wikipedia.org/wiki/Finite_field

[13] More details of working of ECC and requirement of choosing elliptic curve — http://www.cryptoman.com/elliptic.htm

Although this trivial protocol is about as simple as you can make it (and at its core, is how non-PFS TLS works) — it does have Future Secrecy, as disclosing any individual session key does not allow you to attack any future (or prior) sessions.

3.3.2 Adding Forward Security

Now consider the same example, but this time Alice and Bob modified the protocol to have the property of Forward Security. Alice and Bob use a DH Key Exchange (authenticated by the long term identity keys) to generate a symmetric encryption key for the initial session. They use this session key in this and all future sessions.

This protocol will provide Forward Security. But disclosure of the session key allows attacking prior sessions — so this protocol does not have *Perfect* Forward Security. It also allows compromising all future sessions, so it does not have Future Security either.

Disclosing an identity key does not compromise future sessions, unless the protocol is started from scratch. In that case, the attacker can perform a man in the middle attack on the DH exchange and impersonate one of the parties to the other.

3.3.3 Adding Perfect Forward Security

As we saw, compromising the session key allowed the attacker to decrypt all prior sessions. We'd like to fix this problem. Instead of using the same symmetric key for every connection, Alice and Bob will hash the secret. For the first session, they use an agreed upon Key K, and at the conclusion of the session, they store $H(K)$. On the next session, they use $H(K)$ as the key, and at its conclusion, they store $H(H(K))$.

This proposal provides Perfect Forward Security — if an attacker compromises a session key, they cannot decrypt prior sessions because they cannot invert the hash function. But it does allow an attacker to decrypt future sessions, as they can iterate the hash function also.

As in the prior protocol, disclosing an identity key does not compromise future sessions, unless the protocol is started from scratch. In that case, the attacker can perform a man in the middle attack on the DH exchange and impersonate one of the parties to the other.

3.3.4 Achieving Future Secrecy

Let us consider our attacks again:

1. Compromise of Identity Key Attacking Prior Sessions
2. Compromise of Identity Key Attacking Future Sessions
3. Compromise of Session Key Attacking Prior Sessions
4. Compromise of Session Key Attacking Future Sessions

So far we have achieved defense against the first and third — let us add the Future *Secrecy* property to achieve security against the last. This protocol is actually noticeably simpler than the last two. Alice and Bob will perform a DH Exchange, authenticated with their long term identity keys. They use this exchange to agree upon a new symmetric key, at every run of the protocol.

That's it (and indeed, this is how TLS with PFS works, roughly). Compromise of an identity key does not allow compromise of prior sessions. Compromise of a session key does not allow compromise of prior or future sessions.

And it is possible to extend this protocol even more! By authenticating the DH Key Exchange with both the long term identity key and the *prior session key*, we can ensure an attacker who compromises *only* the identity key cannot attack future protocols — they would need to compromise both the identity key and the most recent session key.

However, keeping some keying material around from the prior session to authenticate the next is complicated. While protocols such as ZRTP[14] are able to, it is much more difficult for TLS, where there are multiple sessions active and open at any time. For more about this concept (often called 'ratcheting'), we recommend Moxie Marlinspike and Trevor Perrin's work.[15]

3.4 What can go wrong

Even in a well-designed protocol, there are several attacks that can occur and compromise the confidentiality of a connection.

- **Long Term Identity Key Compromise**: As explained, the long term identity key may get compromised, which usually allows an attacker to perform an impersonation attack.
- **Bad Random Number Generator**: Any key generated with a biased or broken random number generator will be significantly weaker than it's advertise bit-length, allowing more efficient brute force attacks to recover the key.
- **Algorithm Strength Mismatch**: In certain circumstances, the PFS algorithm (e.g. the DH handshake) may be weaker than the Identity Keys. This allows an attacker to target the weakest link in the protocol. This is poor practice, and should be avoided, but deployment may pose problems, an example being TLS today does DH in 1024-bit groups, but authenticated with 2048-bit RSA identity keys.
- **Attack Algorithm Advances**: Public-Key cryptography is based on trapdoor functions like factoring or the discrete log problem — improvements in attacking these functions leads to the weakening of any public-key cryptography, PFS algorithms included.

4 Implementation of PFS in different protocols

In this section, several commonly-used protocols and their use of Forward Security are explained. For each, the paper addresses a brief description and walk-through of the protocol — with focus on the key exchange mechanism with and without Forwards Security. Later, the paper explains the conditions, configurations, and the caveats to achieve Forward Security in these protocols and real life utility of them.

4.1 Transport Layer Security (TLS) Protocol

In a typical TLS handshake, authentication is one-way, meaning that only the server is authenticated to the client. In a non-PFS TLS handshake using RSA keys, authenticated key exchange is achieved via the following mechanism:

1. The server sends its public key, almost always contained in an x509 certificate.
2. The client, upon successfully verifying the certificate, replies with the pre-master secret, encrypted with the server's public key.
3. The server decrypts the client's key exchange message and both parties derive a shared session key from the pre-master secret.

[14] http://tools.ietf.org/html/rfc6189
[15] https://whispersystems.org/blog/advanced-ratcheting/

4. Both parties start communicating over encrypted channel using the shared session key.

The TLS handshake described above does not have Forward Security. The server's public key is used to encrypt the keying material of the individual session, which can only be decrypted by server's private key. The security of all sessions relies on a single static key (server's private key). If the server's private key is compromised, the security of all sessions established under that key is compromised.

The property of PFS ensures that no long-term key compromise can affect the security of past sessions, which is achieved in TLS via authenticated Diffie-Hellman (DH) key agreement. Contrary to RSA handshakes, in DH handshakes, the server's long-term RSA key only performs authentication: it is only used to sign the server's DH key parameters. In TLS, PFS is often referred to as Ephemeral Diffie-Hellman, and in ciphersuite names is abbreviated DHE. If Ephemeral DH is used, then both parties generate a fresh DH keys for every handshake, and Perfect Forward Security is achieved, as the security of each session depends on a different instance of the DH problem.

In an RSA handshake, the server needs to perform one decryption operation with its private key; however, in an Ephemeral DH handshake, the server needs to perform a signing operation in addition to two exponentiation operations for the DH parameters. While they can be optimized, these operations are still costly. For more efficient DH operations, TLS thus specifies an extension for elliptic curves, which achieve equivalent security with less computational cost. ECC in TLS,[16] includes two types of elliptic curve Diffie-Hellman (ECDH) key exchange — Fixed-key key exchange with ECDH certificates (which lack PFS) and ephemeral ECDH key exchange with a RSA or ECDSA certificate.

In a TLS server, PFS can be configured by enabling the server to honor the TLS cipher suite order and placing the ECDHE and DHE suites at the top of the list. The suites that need to be enabled for PFS are:

- `DHE-DSS-AES-<k>-GCM-SHA<h>`
- `DHE-RSA-AES-<k>-GCM-SHA<h>`
- `DHE-DSS-AES-<k>-CBC-SHA<h>`
- `DHE-RSA-AES-<k>-CBC-SHA<h>`
- `DHE-RSA-AES-<k>-CBC-SHA`
- `ECDHE-ECDSA-AES<k>-GCM-SHA<h>-P<c>`
- `ECDHE-ECDSA-AES<k>-CBC-SHA<h>-P<c>`
- `ECDHE-RSA-AES-<k>-CBC-SHA<h>-P<c>`
- `ECDHE-RSA-AES-<k>-CBC-SHA-P<c>`

Here possible values of key sizes (k) are 256 and 128; hash digest sizes (h) are 521, 384, and 256; and curve (c) sizes are 384 and 256. In addition, the GCM suites (supported in TLS 1.2, but not widely implemented) can be placed higher up in the hierarchy as they provide better security than Cipher Block Chaining (CBC) mode.

4.1.1 Implementation of PFS in Apache - Linux

PFS requires Apache 2.2.* or higher. Here is one example configuration for mod_ssl that will work to enable PFS for the current stable branch Apache 2.4.*:

```
SSLProtocol +TLSv1 +TLSv1.1 +TLSv1.2
SSLHonorCipherOrder On
SSLCipherSuite ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-GCM-SHA256:
    ECDHE-RSA-AES128-SHA256:ECDHE-RSA-RC4-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES256-GCM-SHA384:DHE-
    RSA-AES256-SHA256:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES128-SHA256:DHE-RSA-AES256-SHA:DHE-RSA-
    AES128-SHA:RC4-SHA:AES256-GCM-SHA384:AES256-SHA256:CAMELLIA256-SHA:ECDHE-RSA-AES128-SHA:
    AES128-GCM-SHA256:AES128-SHA256:AES128-SHA:CAMELLIA128-SHA
```

Listing 1: Configuration of mod_ssl to enable PFS

[16] http://tools.ietf.org/html/rfc4492

4.1.2 Implementation of PFS in nginx

To implement PFS in nginx, add the following ciphers suites in the configuration file.

```
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
ssl_prefer_server_ciphers on;
ssl_ciphers "EECDH+ECDSA+AESGCM EECDH+aRSA+AESGCM EECDH+ECDSA+SHA384 EECDH+ECDSA+SHA256 EECDH+
    aRSA+SHA384 EECDH+aRSA+SHA256 EECDH+aRSA+RC4 EECDH EDH+aRSA RC4 !aNULL !eNULL !LOW !3DES !MD5
     !EXP !PSK !SRP !DSS";
```

Listing 2: Configuration of nginx to enable PFS

4.1.3 What can go wrong

The client announces the list of ciphersuites and elliptic curves it supports in ClientHello message of TLS handshake. If the server chooses an ECDH or ECDHE ciphersuite, the server also chooses a curve that both the server and the client support. But if the server chooses a DH or DHE ciphersuite, the server can choose an arbitrary DH group. Some servers choose DH groups as small as 256 bits. Smaller DH groups mean that the connection can be broken open with relatively little effort. So, the implementation of PFS can be botched by using under-sized DH groups. Ideally the DH group would match or exceed the RSA key size (preferable > 1024 bits).

Moreover, there is no possible way for server to determine the security requirements of the client. If the client does not consider the group selected strong enough, the client terminates the connection. For ECC, TLS extensions allow a client to negotiate the use of specific curves and point formats. If Gillmore's proposal[17] is adopted, TLS will also support DH field negotiation.

Another way PFS can be botched is by improper session management at the server side. Session resumption[18] in TLS is performed by either session ID[19] or by session tickets. To optimize the performance, if the server store the session keys on the server side long after the session has been terminated, those session keys can be used later to decrypt the encrypted messages.

TLS' implementation is Perfectly Forward Secure (protecting prior sessions from session key compromise) and Forward Secret (protecting future sessions from session key compromise); however, disclosure of a long term identity key does allow the attacker to impersonate the server and compromise future sessions.

4.1.4 Real life application

TLS is the backbone in providing secure communication over Internet. The most common use is for securing web browser sessions, but it has widespread application to other tasks, such as securing email servers or any kind of client-server transaction. TLS can also be used to tunnel an entire network stack to create a VPN, and to authenticate and encrypt the Session Initiation Protocol (SIP) often used in VoIP (Voice over IP).

4.2 Off-the-Record (OTR) Messaging

Off-the-Record (OTR) Messaging is a cryptographic protocol that allows two parties to have private conversations over instant messaging using a combination of the AES symmetric-key algorithm, the Diffie–Hellman key exchange, and the SHA-1 hash function providing:

- **Confidentiality**: No one else can read others' instant messages.

[17] http://tools.ietf.org/html/draft-ietf-tls-negotiated-dl-dhe
[18] http://tools.ietf.org/html/rfc5077
[19] https://en.wikipedia.org/wiki/Session_ID

- **Authentication**: Users are assured the correspondent is who they think it is.
- **Deniability**: The Message Authentication Code (MAC) keys that have already been used and will not be used again are included in the subsequent outgoing messages. The idea is that since these keys are in the clear, any one could have created or used these keys (including your chat partner) and therefore forged a message. However, during a conversation, it assures that the messages are authentic and unmodified.
- **Perfect Forward Security**: Each instant message is encrypted using a different encryption key which is discarded after use. Compromising a single encryption key does not impact on the confidentiality of other messages sent or those to be sent in the future. In addition, each message is authenticated using a different MAC key.

The OTR protocol consists of two phases:

1. An authenticated key-exchange is performed obtaining a shared session key.
2. A continuous refresh of the session key (re-key) during the exchange of instant messages[20] (often called a "ratchet").

In the OTR messaging protocol, one user signals the other about their willingness to use OTR to communicate, either by using an OTR Query Message or using a whitespace-tagged plaintext message. Once the other user accepts the request, communication begins over OTR messaging protocol.

Alice picks her own DH encryption key k_A and a serial number k_id_A. Bob does the same and generates DH encryption key k_B and serial number k_id_B. Both Alice and Bob share the public component of their DH keys over the wire. Both of them sign the components for authenticity with their long term identity key.

Alice picks the most recent of Bob's DH encryption keys k_B and serial number k_id_B. Alice then uses Diffie-Hellman to compute a shared secret from the two keys k_A and k_B, and generates the AES key e_k, and the MAC key m_k. She picks a value of the counter, ctr, so that the triple (k_A, k_B, ctr) is never the same for more than one data message that Alice sends to Bob. She computes T_A = (k_id_A, k_id_B, $key_{B(j)}$, ctr, $AES - CTR_{e_k,ctr}(msg)$) and sends Bob T_A, $MAC_{m_k}(T_A)$, $oldm_k$. If the above key is Alice's most recent key, she generates a new DH key ($key_{A(i)}$), to get the serial number $k\_id_A + 1$. Alice maintains the old MAC keys that were used in previous messages, in a list, $oldm_k$.

Bob uses Diffie-Hellman to compute a shared secret from the two keys k_A and k_B, and generates the receiving AES key, e_k, and the receiving MAC key, m_k. These will be the same as the keys Alice generated above. Bob then uses m_k to verify $MAC_{m_k}(T_A)$ and e_k, and ctr to decrypt $AES - CTR_{ek,ctr}(msg)$.

All the instant messages are encrypted with a new symmetric key. Both the parties perform these key exchanges with every instant message. For every successive message, both parties create new keys, discard old keys and increment the serial number. There is no "official" protocol implementation of OTR for multi-party messaging; OTR provides Forward Security to only two parties messaging currently.

4.2.1 What can go wrong

OTR was designed with Perfect Forward Security in mind. Furthermore, the ratcheting of Diffie-Hellman keys goes even further than PFS - because nearly every individual message is encrypted with a different key, one would have to compromise a significant amount of session state to reach any more than a single message in a session. However, after a session is ended, disclosure of a long term identity key does allow an attacker to impersonate the victim and compromise future sessions.

The OTR protocol performs a DH key exchange per message and such exchange is authenticated with the previous key. The goal is to obtain a fine-grain PFS mechanism in which learning the encryption key at one point will not allow the adversary to learn even a single past message. However, if the adversary learns the current ephemeral key,

[20] More on the working of these two phases — http://www.cypherpunks.ca/otr/Protocol-v3-4.0.0.html

future messages may be completely compromised. Indeed even if the new encryption key is not computable from the old one (being the result of a fresh DH exchange), the adversary can impersonate the parties because the old key is used for authentication. In particular the attacker can hijack the session and learn/modify all future messages.

OTR uses opportunistic authentication, where users are expected to authenticate fingerprints out of band or using a Socialist's Millionaire's Protocol. If this authentication is performed, it is secure, but it is often omitted.

4.2.2 Real life application

Many chat applications use the OTR protocol to encrypt instant messages and provide Forward Security. Chat clients like TextSecure,[21] Pidgin,[22] Jabber,[23] Adium,[24] Jitsi[25] uses OTR or variants of OTR for XMPP and SMS transport protocols.

4.3 Secure Shell (SSH) Protocol

The Secure Shell (SSH) Protocol is a protocol for secure remote login and other secure network services over an insecure network. SSH is organized as three protocols that typically run on top of TCP:

- **Transport Layer Protocol:** Provides server authentication, data confidentiality, and data integrity with Forward Security; the transport layer may optionally provide compression.
- **User Authentication Protocol:** Authenticates the user to the server.
- **Connection Protocol:** Multiplexes multiple logical communication channels over a single underlying SSH connection.

Our point of interest will be in the Transport Layer Protocol, where the key exchange happens based on the server possessing a public-private key pair. As discussed earlier, the solution to have PFS is to use ephemeral key exchange; instead of always encrypting messages using the same static key, peers in a message exchange negotiate secrets through an ephemeral key exchange.

- **SSH v1:**

  After the initial TCP connection is set up, the server sends the client two keys: a host key and a server key. The host key is a persistent asymmetric key used for server identification whereas the server key is a temporary asymmetric key. The client doubly encrypts the session key to the server key and host key. The server key is discarded periodically, by default every hour.

  The server key could be discarded after every session, but the duration is decided on the requirement. The user can set the lifetime and size of ephemeral version 1 server key by setting the option *KeyRegenerationInterval* (specifies how long in seconds the server should wait before automatically regenerating its key) and *ServerKeyBits* (specifies how many bits to use in the server key) appropriately in *sshd_config* file. This server key provides PFS in SSH v1. Once the server destroys the server key, it is not possible to recover the session key.

- **SSH v2:**

  After establishing the TCP connection, both systems agree on a session key via the *SSH_MSG _KEXINIT* message, using DH key exchange.[26] SSH v2 uses the long-term host key only to authenticate the server during DH or ECDH key exchanges; it does not encrypt any data using it.

[21] https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms&hl=en
[22] http://pidgin.im/
[23] http://www.jabber.org/
[24] https://www.adium.im/
[25] https://jitsi.org/
[26] http://tools.ietf.org/html/rfc4253#section-8

In SSH v2, DH is used to set up the session keys. DH inherently provides PFS without need of a second server key as was required in SSH v1. SSH v2 destroys the information that would compromise the session key immediately after closing the session, instead of sometime later. SSH v2 also supports use of elliptic curve algorithm defined in RFC-5656.[27] The key exchange between the server starts with the *SSH_MSG_KEX_ECDH_INIT* message after the initial TCP connection.

A user can configure the DH exchange methods in *sshd_config* as:

```
KexAlgorithms ecdh-sha2-nistp256, ecdh-sha2-nistp384, ecdh-sha2-nistp521,
diffie-hellman-group-exchange-sha256,diffie-hellman-group14-sha1,
diffie-hellman-group-exchange-sha1, diffie-hellman-group1-sha1
```

Listing 3: Snippet of sshd_config listing the DH Key Exchange algorithm

The "diffie-hellman-<group_value>-exchange-<hash_value>" specifies Diffie-Hellman Group and Key Exchange with HASH. According to RFC-4419[28] the acceptable groups with minimum and maximum values of a modulus length of k bits are 1024 and 8192 respectively.

SSH sessions resulting from a key exchange using the Diffie-Hellman methods (including "diffie-hellman-group1-sha1" and "diffie-hellman-group14-sha1") are secure even if private keying/authentication material is later revealed, but as with all PFS protocols, not if the session keys are revealed.

4.3.1 What can go wrong

SSH v1 achieves Perfect Forward Security by rotating the server key regularly. If the private components of both the server and host key were compromised, then all the sessions established between the host and the client during the lifetime of the server key are also compromised. This compromise does not affect the sessions created before the lifetime of compromised server key. Disclosure of the host key allows server impersonation for future sessions.

SSH v2 achieves Perfect Forward Security by using a DH key exchange as the default key exchange mechanism. If a session key is compromised, it will not allow the adversary to learn about any previous or future sessions. But disclosure of a long term host key still allows the attacker to impersonate the server and compromise future sessions.

Generally speaking, SSH makes it difficult to have PFS go wrong. If the DH private parameters for the client or server are revealed, then the session key is revealed, but these items are thrown away periodically in SSH v1 and right after the key exchange completes in SSH v2. It is possible to alter a server to prefer a weaker group (e.g. the 1024-bit diffie-hellman-group1-sha1) before a stronger group (e.g. the 2048-bit diffie-hellman-group14-sha1) but this would require overriding a default.

4.3.2 Real life application

Secure Shell (SSH) is used for secure data communication, remote command-line login, remote command execution, and other secure network services between two networked computers. It is extensively used as a replacement for Telnet and other insecure remote shell protocols.

4.4 Wireless Connections (WPA2 EAP-PWD)

Extensible Authentication Protocol (EAP) is an authentication framework frequently used in wireless networks and Point-to-Point connections. We will mainly concentrate on the EAP-PWD[29] protocol implemented for WPA2 protocol for authentication and authorization in wireless networks. EAP-PWD uses a shared password for authentication

[27] http://tools.ietf.org/html/rfc5656
[28] http://tools.ietf.org/html/rfc4419
[29] https://tools.ietf.org/html/rfc5931

between a peer and an authenticator. The underlying key exchange is resistant to active (including dictionary attacks) and passive attacks. It is easy to configure as this does not use certificates for authentication, and can resists active attacks even in case of low entropy passwords.

EAP-PWD is a four step process:

1. Identity exchange: The peer and server use the Identity exchange to discover each other's identities and to agree upon a ciphersuite to use in the subsequent exchanges.
2. Commit exchange: The peer and server exchange information to generate a shared key.
3. Confirm exchange: The peer and server prove "liveness" and knowledge of the password by generating and verifying verification data.
4. Shared secret generation: The shared secret is generated from the information exchanged in previous messages between peer and server for a particular session.

During Commit exchange, the peer and server use each other's identities and the agreed upon ciphersuite to fix an element in the negotiated group (either an integer DH group or an ECC group) called the Password Element (PWE). The server then chooses two random numbers s_rand and s_mask (private components) to calculate $Scalar\_S$ and $Element\_S$ (public components), and send them to the peer. Similarly, the peer chooses two random numbers p_rand and p_mask (private components) to calculate $Scalar\_P$ and $Element\_P$ (public components), and send them to the server. In Confirm exchange, the server calculates ks, $Confirm\_S$ and peer calculates kp, $Confirm\_P$ from previously exchanged information. $Confirm\_S$ and $Confirm\_P$ are calculated as:

$$Confirm\_ < i >= H(k < i > |Element\_S|Scalar\_S|Element\_P|Scalar\_P|Ciphersuite)$$

where value of $< i >$ is either P or S, and H is the random function specified in the ciphersuite. Both the peer and the server exchange these values during confirm exchange stage of the protocol.[30]

In the final step, the EAP Server computes the shared secret Master Key(MK) as: $MK = H(ks|Confirm\_P|Confirm\_S)$ and the EAP Peer computes the shared secret as: $MK = H(kp|Confirm\_P|Confirm\_S)$. Master Session Key (MSK) or Extended Master Session Key (EMSK) are derived as KDF(MK, Session-ID, 1024), where KDF is the Key Derivation Function[31] and Session-ID = Type-Code | $H(Ciphersuite|Scalar\_P|Scalar\_S)$.

The MSK and EMSK are extracted from MK, which is derived from doing group operations with s_rand, p_rand, and the PWE. The peer and server choose random values with each run of the protocol. So even if an attacker is able to learn the password, the attacker will not know the random values used by either the peer or server from an earlier run and will therefore be unable to determine MK, or the MSK or EMSK. Hence the protocol provides Perfect Forward Security.

There are n possible distinct pairs of numbers that will produce $Scalar\_P$ and $Scalar\_S$, where n is the order of the chosen group. The difficulty of finding the value of MK without the knowledge of s_rand and p_rand lies in the fact that for a large n, it is computationally infeasible to guess these random numbers.

4.4.1 What can go wrong

EAP-PWD provides Perfect Forward Security by generating the random numbers $rand$ and $mask$ per session, so no future or previous sessions can be compromised if either the private components or the session key/Forward Secret (MK) are compromised. However, if the long term identity key is compromised, the peer can be tricked into authenticating to an adversary instead of the actual authentication server. All the sessions hence forward will be compromised as long as the adversary can maintain their position as the authentication server, which is limited by the characteristics of rogue access point attacks.

[30] Details of calculation of $Element\_S$, $Scalar\_S$, $Element\_P$, $Scalar\_P$, ks, and kp are present in Section 2.8.4 - Message Construction — http://tools.ietf.org/html/rfc5931#section-2.8.4

[31] http://tools.ietf.org/html/rfc5931#section-2.5

The strength of the shared secret, MK, depends on the effort needed to solve the discrete logarithm problem in the chosen group. Choosing a weak group will decrease the computational complexity and can be broken with relatively less effort. RFC 5931[32] recommends using DH Group 19 (256 bit ECC curve) for this implementation. For example, to configure EAP-PWD in a radius server, one can set the pwd_group=19 in the server configuration file.

The security of EAP-PWD relies upon both the peer and server, producing quality secret random numbers. A poor random number chosen by either side in a single exchange can compromise the shared secret from that exchange and open up the possibility of dictionary attack.

4.4.2 Real life application

EAP-PWD is supported on Android beginning with 4.0 (Ice Cream Sandwich) and is supported by the FreeRADIUS and Radiator RADIUS servers.[33] hostapd and wpa_supplicant also support EAP-PWD. It also has plug-in that provides EAP-PWD support for the Windows supplicant.

4.5 Internet Protocol Security (IPSec)

Internet Protocol Security (IPSec) is a set of protocols to provide IP security at the network layer. IPSec supports network-level data integrity, data confidentiality, data origin authentication, and replay protection. IPSec is integrated at the Internet layer (layer 3), so it provides security for almost all protocols in the TCP/IP suite, and because IPSec is applied transparently to applications, there is no need to configure separate security for each application that uses TCP/IP.

In order to setup an IPSec connection (Tunnel Mode[34]), there is a need to exchange tunnel parameters securely. The Internet Key Exchange[35] (IKE) is responsible for letting two IPSec devices negotiate tunnel parameters (Phase 1). A Certificate Authority (CA) can create a digital certificate for each device. The certificate has identifying information about the IPSec device and their public key. When a device requests an IPSec session with another device, the other device will send its certificate. Then the two devices will use asymmetric encryption to transfer the tunnel parameters to each other.

Now each device participating in the tunnel will create a public and private key for a DH key exchange, then send a copy of the public key to the peer (Phase 2). Each device will then generate a shared secret key using its private key and its peer's public key.[36] This shared secret key will be used for encrypting the communication.

The Phase 1 proposal creates the key (the `SKEYID_d` key) from which all Phase 2 keys are derived. The `SKEYID_d` key can generate Phase 2 keys with a minimum of CPU processing. Unfortunately, if an unauthorized party gains access to the `SKEYID_d` key, all the encryption keys are compromised. However, if PFS is enabled at both the devices, whenever the Phase 2 keys are regenerated after the key lifetime, the new keys generated will not be derived from any of the specific keying material used to generate any preceding keys.

To provide PFS of both keys and all identities, two parties would perform the following:

- A Main Mode Exchange (Phase 1) to protect the identities of the Internet Security Association and Key Management Protocol (ISAKMP) peers. This establishes an ISAKMP Security Association (SA).[37]

  In main mode, the initiator and recipient send three two-way exchanges to accomplish the following services:

[32] http://tools.ietf.org/html/rfc5931#section-6.5
[33] http://www.open.com.au/pipermail/radiator-announce/2012-June/000018.html
[34] http://technet.microsoft.com/en-us/library/cc737154(v=ws.10).aspx
[35] http://en.wikipedia.org/wiki/Internet_Key_Exchange
[36] For more information on DH key exchange refer:subsection 3.1 — Diffie-Hellman key exchange protocol
[37] A security association (SA) is a unidirectional agreement between the participants regarding the methods and parameters to be used in securing a communication channel. Through the SA, an IPSec tunnel can provide privacy, content integrity, sender authentication and non-repudiation (if using certificates).

  - Propose and accept the encryption and authentication algorithms.
  - Execute a Diffie-Hellman exchange, and the initiator and recipient each provide a pseudo-random number.
  - Send and verify their identities.
- A Quick Mode Exchange (Phase 2) to negotiate other security protocol protection. This establishes a SA on each end for this protocol.
- Delete the ISAKMP SA and its associated state.

An IPSec policy defines a combination of security parameters (IPSec proposals) used during IPSec negotiation. It defines PFS and the proposals required for the connection. During the IPSec negotiation, IPSec looks for a proposal that is the same on both peers. The peer that initiates the negotiation sends all its policies to the remote peer, and the remote peer tries to find a match.

A match is made when both policies from the two peers have a proposal that contains the same configured attributes. If the lifetimes are not identical, the shorter lifetime between the two policies (from the host and peer) is used.

We will look into Virtual Private Network (VPN) — an implementation of IPSec and see how PFS works.

4.5.1 Virtual Private Network (VPN)

A Virtual Private Network (VPN) connection is the extension of a private network that includes links across shared or public networks, such as the Internet. A VPN is created by establishing a virtual point-to-point connection through the use of dedicated connections, virtual tunneling protocols, or traffic encryption. VPN connections (VPNs) enable organizations to send data between two computers across the Internet in a manner that emulates the properties of a point-to-point private link.

Both sides of the VPN must be able to support PFS in order for PFS to work. When PFS is turned on, for every negotiation of a new Phase 2 SA, the two gateways must generate a new set of Phase 1 keys. This is an extra layer of protection that PFS adds, which ensures if the Phase 2 SA's have expired, the keys used for new phase 2 SA's have not been generated from the current Phase 1 keying material. Of course if PFS is not turned on, then the current keying material already established at Phase 1 will be used again to generate Phase 2 SA's. To enable PFS on a device, configure the IPSec policy to support PFS.

```
[edit services ipsec-vpn ipsec policy policy-name]
perfect-forward-secrecy {
keys (group1 | group2 | group5 | group14 | group19 | group20 | group24);
}
```

Listing 4: Example configuration for IPSec Policies to enable PFS in Junos

The groups specify the size of the Diffie-Hellman prime modulus that the IKE will use. The key can be one of the following when performing the new Diffie-Hellman exchange:

- group1: 768-bit, group2: 1024-bit, group5: 1536-bit, group14: 2048-bit,
- group19: 256-bit(ECDH), group20: 384-bit(ECDH), group24: 2048-bit(DH/DSA)

At time of writing, the minimum acceptable security level would be group14, and in some lower-security contexts, group 5.

4.5.2 What can go wrong

During the Phase 2 negotiation, both the peers exchange the policies to negotiate the encryption and algorithm to use, SA life times, PFS etc. Both peers will choose the best configuration settings which are common in their policies. Even if one peer supports PFS, it is not possible to choose PFS if the other peer does not support it. So achieving PFS over VPN depends on both the peers.

Choosing weak groups also diminishes the security advantage that PFS provides. A weak PFS group may not provide a strong enough level of security, such that it is possible to perform successful cryptanalysis.

VPNs provide Perfect Forward Security as both Phase 1 and Phase 2 keys are used for every negotiation of a Phase 2 SA. So a compromised Phase 2 key can only decrypt the content of the current session. However, compromise of a long term key allows a rogue peer to perform an impersonation attack and compromise future sessions.

4.5.3 Real life application

IPSec is used widely for host-based packet filtering to provide limited firewall capabilities for end systems, end-to-end security between specific hosts, Layer Two Tunneling Protocol (L2TP) over IPSec (L2TP/IPSec) for remote access and site-to-site virtual private network (VPN) connections, and site-to-site IPSec tunneling with IPSec gateways among others.

5 Conclusion

Perfect Forward Security provides confidentiality of past sessions even when long-term identity keys have been compromised. After a session is completed, all parties involved have destroyed the private keys, and it is computationally difficult to recover the session, hence the "Perfect" part of Perfect Forward Security. While PFS is undoubtedly a nice property to have, it does come at a cost. PFS requires DH computation, along with the computation using RSA or (EC)DSA algorithm for the authentication mechanism.

Despite the increased computation cost, PFS is seen as an important security feature by several large Internet information providers. Since late 2011, Google has provided Perfect Forward Security with TLS by default to users of its Gmail service, along with Google Docs and encrypted search among other services. Since November 2013, Twitter has provided Forward Security with TLS to users of its service. As of July 2014, 51.3% of TLS-enabled websites support some of the cipher suites which provide Forward Security.[38]

The impossibility of recovering the session depends on the fact that presently there is no effective algorithm to solve factorization or discrete logarithmic problems with modern computational power. However, PFS cannot defend against a successful cryptanalysis of the underlying ciphers or the (EC)DH exchange itself. In particular, Shor's algorithm[39] combined with the potential power of future quantum computing[40] have the ability to solve the factoring problem[41] and discrete logarithm problem, while also halving the effective key strength of symmetric ciphers. Cryptographers are working on new post-quantum era cryptographic algorithms — Ring-LWE[42] and Supersingular Isogeny Diffie-Hellman Key Exchange[43] that can support forward security. Even though quantum computing can be a threat to classical cryptographic algorithms, both quantum computing and post-quantum cryptography with PFS support require time to mature. Until then, if implemented properly, PFS will provide the extra layer of privacy on electronic communication.

[38] https://www.trustworthyinternet.org/ssl-pulse/
[39] https://en.wikipedia.org/wiki/Shor's_algorithm
[40] http://en.wikipedia.org/wiki/Quantum_computer
[41] http://en.wikipedia.org/wiki/Factoring_problem
[42] http://en.wikipedia.org/wiki/Ideal_lattice_cryptography#Ring-LWE
[43] http://en.wikipedia.org/wiki/Supersingular_Isogeny_Key_Exchange