The highly secure document & data platform for capture, extraction, and storage.

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

Concepts of Forward Secrecy Properties in Authenticated Key Exchange Protocols

A white paper by Bill Edwards

The security of cryptosystems essentially relies on the secrecy of the respective secret key. For example, if for an encryption scheme a secret key is (accidentally) leaked, the confidentiality of all the data encrypted with respect to this key so far is immediately destroyed. One simple mitigation strategy for such a secret-key leakage is to frequently change secret keys such that leaking a secret key only affects a small amount of data. Implementing this in a naıve way, for instance in context of public-key encryption, means that one either has to securely and interactively distribute copies of new public keys frequently or to have huge public keys, which is rather inconvenient in practice. Consequently, cryptographic research focused on the design of cryptosystems that inherently provide such a property, being denoted as forward secrecy (or, forward security). The goal hereby is that key leakage at some point in time does not affect the data, which was encrypted before the key leakage, while mitigating the drawbacks of the naıve solution discussed before. That is, one aims at efficient non-interactive solutions that have fixed sublinear-size public keys in the number of key switches/time periods. As we think of cryptography, we have to think about the mathematics of authentication. I won't discuss the details but we can consider ***RSA factorization of discrete logs of***

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

ECC. Their discrete logarithm problem can be stated as follows: if we know *a* and *b*, what's k such that

$$b = a^k \bmod p \ ?$$

Both these problems are "discrete" because they involve finite sets (more precisely, cyclic subgroups). And they are "logarithms" because they are analogous to ordinary logarithms.

In its purest form cryptography is a mathematical panacea of wonder and excitement. Modular arithmetic, discrete mathematics, complex prime number calculations and elliptic curve algorithms form the basis of our current cryptographic algorithms. Such mathematics relies upon one thing... Complexity. In order for an algorithm to provide security to data, it must be computationally infeasible to deduce the original message from knowledge of the algorithm and encrypted message. Such algorithms are relatively simple to compute in one direction, yet intangible in reverse without knowledge of another piece of information, typically a number or set of numbers known as the key. The private key is imperative.

Asymmetric Cryptography

There are two different parts to creating a TLS session. There is the asymmetric cryptography, portion that is an exchange of public keys between two points. This only allows the

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

exchange of asymmetric keys for asymmetric encryption/decryption. This is the **ECDHE** portion. The **RSA** portion describes the signing algorithm used to authenticate the key exchange. This is also performed with asymmetric cryptography. The idea is that you sign the data with your private key, and then the other party can verify with your public key.

Symmetric Cryptography

You encrypt symmetric encryption/decryption keys with your asymmetric key. Asymmetric encryption is very slow (relatively speaking). You don't want to have to encrypt with it constantly. This is what Symmetric Cryptography is for. So now we're at **AES_128_GCM**.

- AES is the symmetric algorithm
- 128 refers to key size in bits
- GCM is the mode of operation

So what exactly does our asymmetric key encrypt? We want to essentially encrypt the symmetric key (in this case 128 bits, 16 bytes). If anyone knew the symmetric key then they could decrypt all of our data. For TLS the symmetric key isn't sent directly. Something called the pre-master secret is encrypted and sent across. From this value the client and server can generate all the keys and IVs needed for encryption and data integrity. Detailed look at the TLS Key Exchange

Data Integrity

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

Data integrity is needed throughout this process, as well as with the encrypted channel. When looking up GCM (explained below), the encryption mode of operation itself provides for the integrity of the data being encrypted. However, the public key handshake itself must also be confirmed. If someone in the middle changed data while being transmitted then how could we know nothing was tampered with? This is what instance where the negotiated hash function is used, **SHA256**. Every piece of the handshake is hashed together, and the final hash is transmitted along with the encrypted pre-master secret. The other side verifies this hash to ensure all data that was meant to be sent was received.

A hash function $h:X \rightarrow Y$ is one-way, which on an arbitrary length input binary string $x \in \{0,1\}^*$ outputs a fixed length n binary string, say $y \in \{0,1\}^n$ such that $y = h(x)$.

Definition 2. ([(Collision-resistant one-way hash function)]) Let $Adv_{(A)}^{HASH}(t)$ ***denote the advantage that an adversary*** $\mathcal{A}$ ***has in finding a collision. Then***

$$Adv_{(A)}^{HASH}(t) = Pr[(a, a') \in_R \mathcal{A} : a \neq a' \\ h(a) = h(a')]$$

where the pair $(x, x') \in_R \mathcal{A}$ ***is selected randomly by*** $\mathcal{A}$ ***.*** $h(\cdot)$ ***is collision resistant if*** $Adv_{(A)}^{HASH}(t)$ ***is negligible, that is, if*** $Adv_{(A)}^{HASH}(t) \leq \psi$ ***, for any sufficiently small number*** $\psi > 0$ ***.***

SHA256 is also used for the Pseudo-Random Function (PRF). This is what expands the pre-master secret sent

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

between the two parties into the session keys we need for encryption.

For other modes of operation, each message would be hashed with this integrity algorithm as well. When the data is decrypted the hash is verified before using the plaintext.

For SSL, there is SSL 3.0, TLS 1.0, TLS 1.1 and TLS 1.2.

In SSL 3.0, an internal Pseudorandom Function (PRF) is used to extend the shared secret (obtained from the key exchange mechanism) into symmetric keys for the subsequent symmetric encryption. This PRF always uses both MD5 and SHA-1.

In TLS 1.0 and TLS 1.1, the PRF is replaced by a new construction, which still uses both MD5 and SHA-1, in an arguably "better" way (cryptographically speaking).

MD5 and SHA-1 are also used in SSL 3.0, TLS 1.0 and TLS 1.1 to compute the signatures in the handshake (a signature from the client when using certificate-based client authentication, a signature from the server when using a DHE cipher suite); that which is signed will be hashed with either SHA-1 (for DSA/ECDSA) or with *both* MD5 and SHA-1 (for RSA).

Also, exchanged handshake messages are hashed, again with both MD5 and SHA-1, to compute the Finished messages which conclude the handshake; in the

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

case of SSL 3.0, the two hashes are used "as is", while with TLS 1.0 and 1.1, an extra PRF invocation is involved.

Beyond the handshake, the application data will be encrypted with some symmetric algorithm *with an integrity check* and that integrity check will normally use HMAC (in the case of SSL 3.0, a HMAC-derivative), which will be based on a hash function specified in the negotiated cipher suite. Cipher suites, which end with “MD5”, use MD5 at that point, while cipher suites, which end with “SHA”, use SHA-1.

With TLS 1.2, things change. The PRF and is still there, but uses only *one* hash function, and that hash function is the one negotiated in the cipher suite. The same hash function will be used to hash the handshake messages for the Finished message (by the way, that's cumbersome for embedded RAM-starved implementations, because implementations must then either buffer the ClientHello and the start of the ServerHello, or start a bunch of hash functions, because the hash function which will be used is not known until the ServerHello is parsed). When the cipher suite ends with "MD5", the hash function is MD5. When it ends with "SHA", that's SHA-1. When it ends with "SHA256", that's SHA-256.

When GCM is used, there is no per-record HMAC; integrity is obtained from the GCM mode itself. So the hash function specified in the cipher suite is used only for the PRF and other handshake-related usages.

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

Now the tricky point: **there is no standard GCM cipher suite for TLS before TLS 1.2**. GCM cipher suites for TLS begin with RFC 5288; others are defined in RFC 5289 and some subsequent RFC. All currently defined cipher suites are published in the IANA registry. All the GCM cipher suites are for TLS 1.2+ only, and, moreover, they all use either SHA-256 and SHA-384, not SHA-1. Correspondingly, **as of July 2013, there is no standard way to use both SHA-1 and GCM in TLS**. If you want GCM (with AES or some other algorithm), you will need TLS 1.2, and the hash function (for handshake purposes) will be SHA-256 or SHA-384. Presumably, cipher suites with SHA-512 may be defined as well, but this has not occurred yet.

Of course, every rule has exceptions. Public keys, in SSL, are exchanged through X.509 certificates, which are complex objects using a lot of cryptography, and you will often find some SHA-1 or even MD5 lurking there. But that's mostly out of scope for TLS, and, in particular, is not impacted by the negotiated cipher suite. Theoretically, client and server may/should negotiate supported hash functions for that, but in practice the server will send *the* certificate it has obtained from its CA, and has little choice in the matter.

For SSH, the algorithms are negotiated independently of each other; there is no real notion of an all-encompassing cipher suite. See RFC 4253, section 7.1. RFC 5647 defines use of AES/GCM with SSH, but does not care about the hash function which will be used internally; of course, there will be

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

no hash-based MAC for bulk data encryption if GCM is used (that's the point of GCM: it comes with its own MAC), but a hash function is still used internally for, at least, key derivation (similarly to what is called the "PRF" in SSL).

RFC 6239 is more thorough in that it tries to define *sets* of algorithms for SSH, following the so-called NSA Suite B. This means elliptic-curve Diffie-Hellman ("ECDHE" in SSL terminology: with SSH, the key exchange always uses "ephemeral keys"), AES/GCM... and SHA-256 or SHA-384, not SHA-1. Also, when strictly following RFC 6239, the SSH server public key (the permanent one, used for signatures) should use ECDSA, not RSA.

Let’s introduce a theorem with proof for such an implementation.

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

Theorem: Let $\varphi : G \to H$ be a group homomorphism. Then the quotient $G/\ker\varphi$ is isomorphic to the image of φ. That is,

$$G/\ker\varphi \cong \varphi(G)$$

As a quick corollary before the proof, if φ is surjective then $H \cong G/\ker\varphi$.

Proof. We define an explicit map $f : G/\ker\varphi \to \varphi(G)$ and prove it is an isomorphism. Let $g\ker\varphi$ be an arbitrary coset and set $f(g\ker\varphi) = \varphi(g)$. First of all, we need to prove that this definition does not depend on the choice of a coset representative. That is, if $g\ker\varphi = g'\ker\varphi$, then $f(g) = f(g')$. But indeed, $f(g)^{-1}f(g') = f(g^{-1}g') = 1$, since for any coset N we have by definition $gN = g'N$ if and only if $g^{-1}g' \in N$.

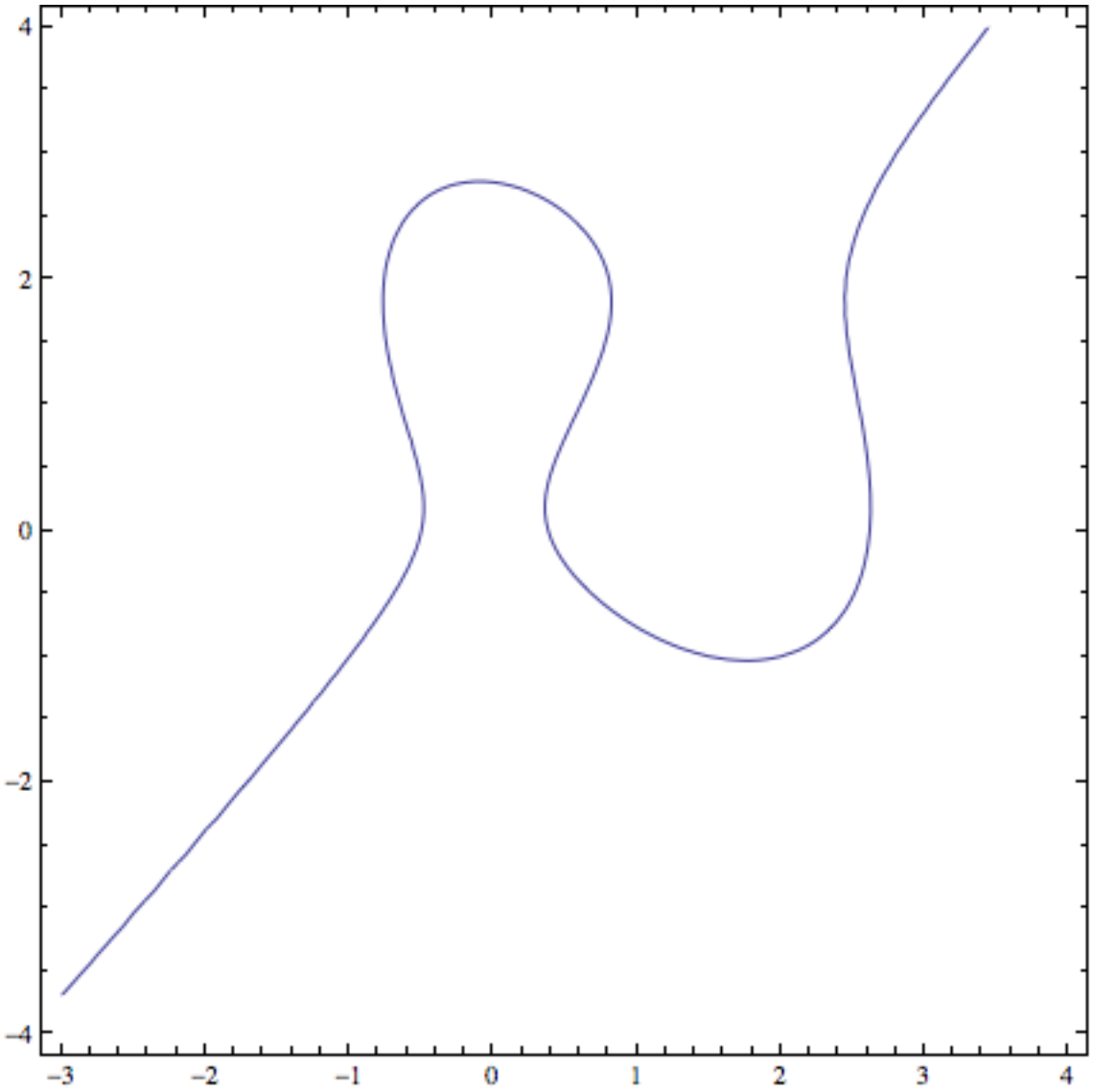
It is similarly easy to verify that f is a homomorphism:

$$f((g\ker\varphi)(g'\ker\varphi)) = \varphi(gg') = \varphi(g)\varphi(g') = f(g\ker\varphi)f(g'\ker\varphi)$$

It suffices now to show that f is a bijection. It is trivially surjective (since anything in the image of φ is in a coset). It is injective since if $f(g\ker\varphi) = 1$, then $\varphi(g) = 1$ and hence $g \in \ker\varphi$, so the coset $g\ker\varphi = 1\ker\varphi$ is the identity element. So f is an isomorphism. □

$$y^2z = x^3 + axz^2 + bz^3$$

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005



So you *can*, at least theoretically, used SSH with ECDHE, RSA signatures, AES/GCM *and* SHA-1, but when you use GCM you are encouraged to also switch to SHA-256 or SHA-384. What SSH implementations actually do is prone to depend on the exact software version, compilation options and local configuration, and needs testing.

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

Key lengths can be confusing because you encounter different numbers and recommendations. For example, when creating an SSH RSA key, you need to use at least a 2048-bit key. This is way larger than 128 bits.

AES is a symmetric cipher. You can't compare it with RSA, an asymmetric cipher. AES supports only 3 key lengths - 128, 192, and 256. Choosing a larger key isn't always a good idea because of performance reasons. In the case of AES, 128 is secure enough. It will take several decades to break an AES 128-bit key in the absence of quantum computers.

It is interesting to note that key lengths for symmetric ciphers only matters if a brute-force attack are the best-known attack. If an analytical attack exists, a large keyspace does not help at all. In the case of AES, no such attack exists and that's why it's the currently most used symmetric cipher today.

When you connect to a website (server) securely, you generally do so over HTTPS. In the first few milliseconds of the connection between your browser and the server, your browser sends the server some information about what kind of encryption it supports, and the server replies back with a verification and the encrypted connection begins.

The relevance to ‘Perfect Forward Security’ (PFS) is that the mechanism of using asymmetric encryption to agree a symmetric encryption key, which is then used for the remainder of the SSL/TLS session, is dependent entirely on

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

the secrecy of the private key. If the private key is ever disclosed, even in the future, it is possible to go back and decrypt entire key exchange conversation and from that obtain the encryption key used for the symmetric encryption used for the remainder of the 'secure' session.

This holds true of every SSL/TLS session using that private key. Even though each secure session will have used a separate and quite different symmetric encryption key, they will all have agreed and set that unique key using the single private key and so possession of that single private key allows discovery of all the separate symmetric encryption keys and with them the decryption of every secure communication exchange with that server.

Perfect Forward Security is designed to overcome this inherent weakness and is so-called as it protects a secure message exchange from future disclosure by breaking of the encryption due to loss or disclosure of the private key used in the key exchange. It achieves this by extending the key exchange mechanism and using an intermediate temporary encryption to protect the exchange of the symmetric encryption key. This means a future loss or disclosure of the private key can no longer be used to decrypt the secure message, e.g. perfect forward security.

As we mentioned earlier, asymmetric cryptography is a mechanism where one key is used to encrypt a message, but cannot then be used to reverse the encryption, which can only be achieved, by use of the other key. Typically a party keeps

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

one key secret, the so-called private key, and makes the other key freely available, the so-called public key. This allows a message to be encrypted with the party's public key and transmitted to the party who is then able to decrypt the message with the secret private key.

A downside to asymmetric encryption is that it is much more computing-intensive than symmetric encryption where the same key is used to both encrypt and decrypt a message, consequently in SSL and TLS, asymmetric encryption is used by the two parties – the client machine and server – to exchange and set-up how they are going to communicate with each other, part of which is the agreement over a symmetric encryption key they will use for the session that follows.

To enable PFS, the client and the server have to be capable of using a cipher suite that utilizes the Diffie-Hellman key exchange. Importantly, the key exchange has to be ephemeral. This means that the client and the server will generate a new set of Diffie-Hellman parameters for each session. These parameters can never be re-used and should never be stored, the ephemeral part, and that's what offers the protection going forwards. Because of the magic behind Diffie-Hellman, the exchange of key material can take place in clear text without compromising the generation of a shared secret.

Because the shared secret, the session key, is derived from complex mathematical operations carried out on the numbers

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

exchanged between the client and the server, that are too difficult for an attacker to brute force, the attacker can at no point record data that is used to derive the session key. Even if the private key of the server is compromised, it doesn't aid the attacker in decrypting any data because the associated public key was never used to protect anything. What's even better is that with Perfect Forward Secrecy, the server generates a new set of Diffie-Hellman parameters for each session and both parties create a new-shared secret that is unique and unknown to an attacker. Even if the attacker managed to compromise this shared secret somehow, it would only compromise that particular session. No previous or future sessions would be compromised.

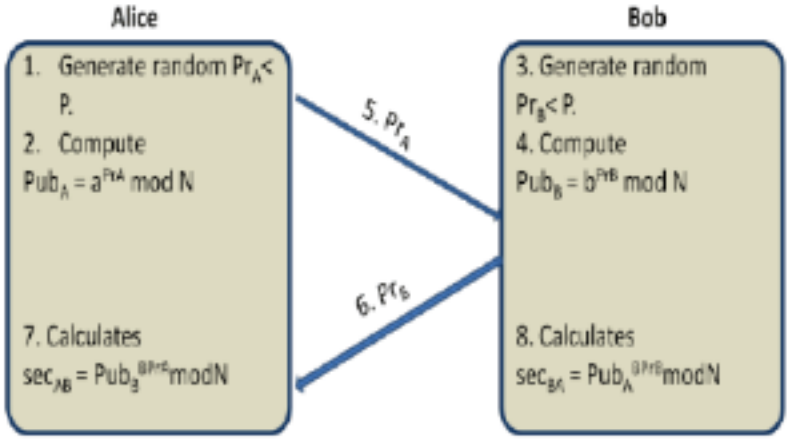
1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

1. Alice and Bob agree to use a prime number $p = 23$ and base $g = 5$.
2. Alice chooses a secret integer $a = 6$, then sends Bob $A = g^a \bmod p$
   - $A = 5^6 \bmod 23$
   - $A = 15{,}625 \bmod 23$
   - $A = 8$
3. Bob chooses a secret integer $b = 15$, then sends Alice $B = g^b \bmod p$
   - $B = 5^{15} \bmod 23$
   - $B = 30{,}517{,}578{,}125 \bmod 23$
   - $B = 19$
4. Alice computes $s = B^a \bmod p$
   - $s = 19^6 \bmod 23$
   - $s = 47{,}045{,}881 \bmod 23$
   - $s = 2$
5. Bob computes $s = A^b \bmod p$
   - $s = 8^{15} \bmod 23$
   - $s = 35{,}184{,}372{,}088{,}832 \bmod 23$
   - $s = 2$
6. Alice and Bob now share a secret (the number **2**).

Enabling support for **Perfect Forward Secrecy** on your server is actually fairly straightforward. Whilst the appropriate ciphers have been available since SSLv3, it's best to ensure

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

you have support for the latest TLSv1.2 protocol. After that, it's just a case of picking the correct cipher suites and ensuring that they are ordered correctly. Any Diffie-Hellman key exchange will provide you with Forward Secrecy, but you should only select Ephemeral key exchange to obtain Perfect Forward Secrecy (a brand new session key for every session). This is usually displayed in the cipher suite in the form of DHE or EDH. You should also include Elliptic Curve DHE suites, as they are faster than their DHE counterparts and should be prioritized above them where possible. You can opt to exclude DHE suites and just stick with ECDHE suites.



PFS in practice

Server overhead

The use of more advanced encryption techniques demands additional computing resource as both ECDHE_RSA and DHE_RSA place greater strain on a server supporting and using these ciphers. Of the two, ECDHE_RSA is believed to be more efficient, adding an approximate 20% overhead of

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

SSL/TLS processing, while DHE-RSA is reported as being much less efficient.

Browser support

PFS cipher suites are being adopted by newer versions of browsers. Google's Chrome browser is generally considered to be at the front of PFS support although current versions of Firefox, Internet Explorer, Safari and other browsers also now support PFS.

Users of Windows XP and Internet Explorer cannot use PFS or indeed most high-grade encryption ciphers, e.g. AES, due to restrictions in the Microsoft encryption libraries made available to Windows XP, however, PFS is available to Windows XP users using Google Chrome, Firefox and other modern browsers.

Adoption and usage of PFS

Google have been a major proponent of PFS and have offered PFS cipher suites on Gmail and Google Docs for some considerably time. If you therefore use these services via a current browser, you will be making use of PFS.

Twitter has also supported PFS since late 2013, as have a few other US-based services.

Cipher suites

Typical cipher suites using both high-grade encryption ciphers (AES256, AES128 and 3DES) and also using ECDHE_RSA taken from Firefox v28 are:

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

Cipher suite	Encryption key size	Key exchange mechanism
ECDHE-RSA-AES128-GCM-SHA256	128 Bit	ECDH, encryption: AES, MAC: SHA256
ECDHE-RSA-AES128-SHA	128 Bit	ECDH, encryption: AES, MAC: SHA1
ECDHE-RSA-AES256-SHA	256 Bit	ECDH, encryption: AES, MAC: SHA1
ECDHE-RSA-3DES-EDE-SHA	168 Bit	ECDH, encryption: 3DES, MAC: SHA1
ECDHE-RSA-RC4128-SHA	128 Bit	ECDH, encryption: RC4, MAC: SHA1

Table 1 – Example ECDHE_RSA high-grade encryption cipher suites

Example

Messaging Terminology – WhatsApp

Modern chat protocols also aim at protecting users from a different kind of attack: the skimming of the session state of an ongoing conversation. Note that the session state information is orthogonal to the long-term secrets discussed above and, intuitively, an artifact of exclusively the second (symmetric) phase of communication. The necessity of being able to recover from session state leakage is usually motivated with two observations: messaging sessions are often long-lived, e.g., kept alive for weeks or months once established, so that state exposure is more damaging, more easily provoked, and more likely to happen by accident; and

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

leaking state information is sometimes impossible to defend against (state information held in computer memory might eventually be swapped to disk and stolen from there, and in cloud computing it is standard to move virtual machine memory images around the world from one host to the other).

There are three keys that are used for encrypting messages sent between two users:

Root Key

32byte key derived from `master secret`

Purpose:Deriving `ChainKeys`

Chain Key

32byte key derived from `RootKey`

Purpose:Deriving `MessageKeys`

Message Key

80byte key derived from `ChainKey`
32byte for AES256 key (message encryption)
32byte for HMACSHA256 key (message authentication)

16byte for IV (random initialization)
Purpose:Encrypting and authenticating a message (onetime use)

```
Message Key = HMAC-SHA256 (Chain Key, 0x01)
```

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

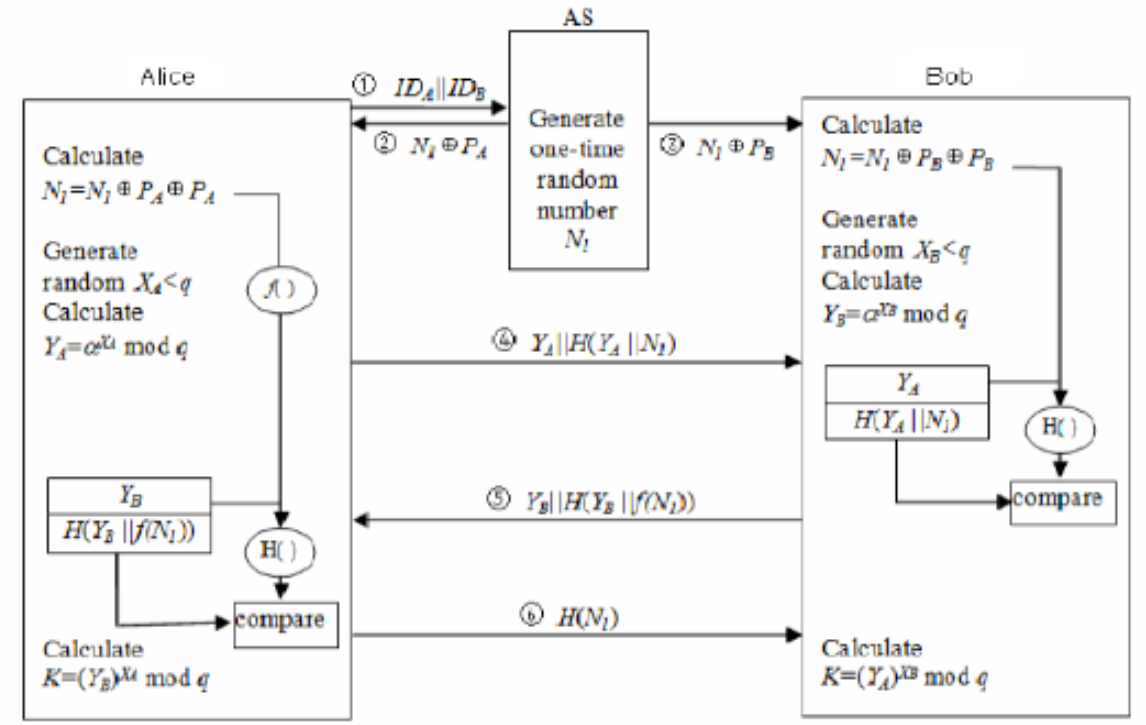
```
Chain Key = HMAC-SHA256 (Chain Key, 0x02)
```

Other examples – SSL and TLS

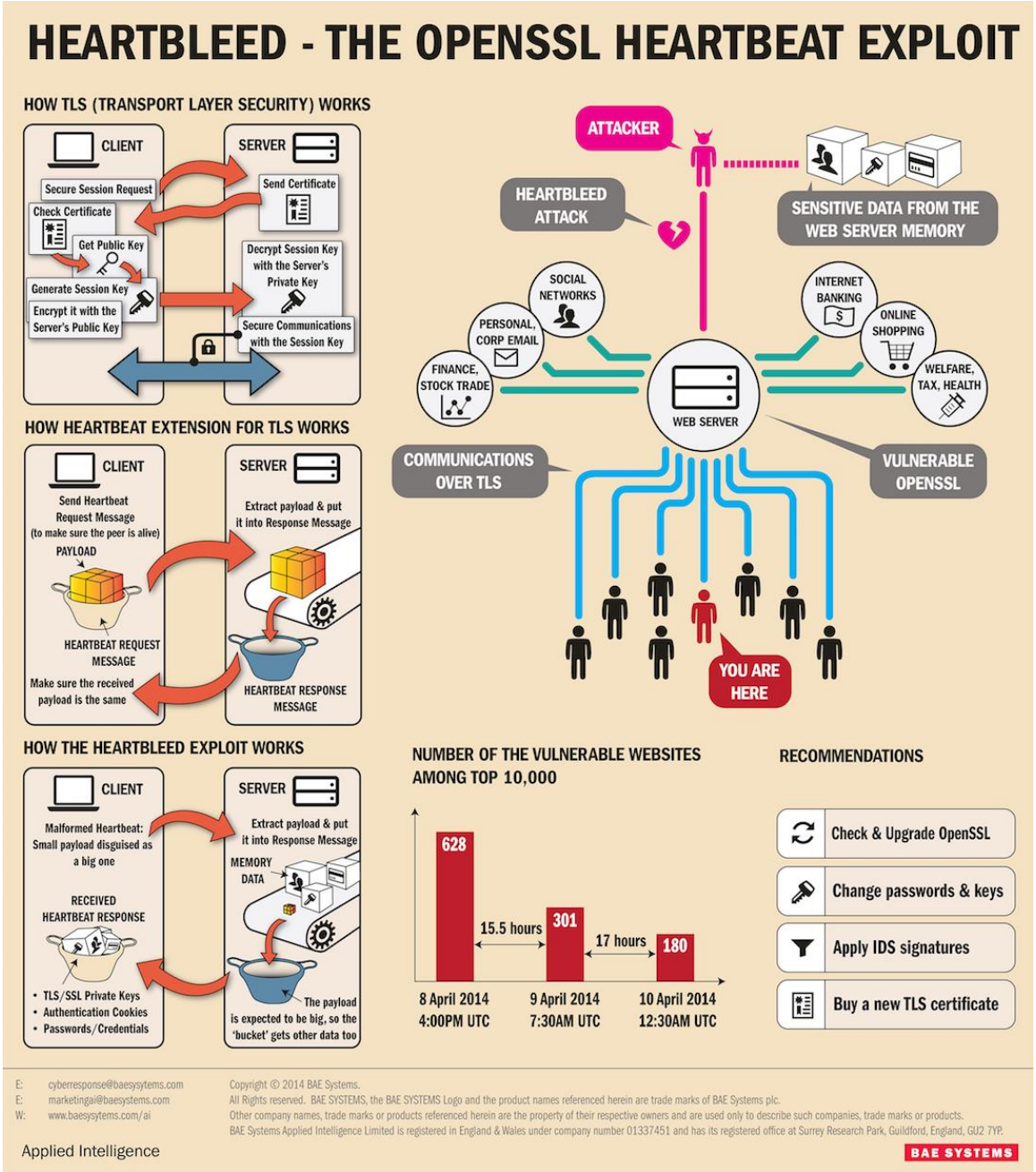
One of the best examples of the need for each online transaction to have its own encryption was the Heartbleed bug, which was discovered in 2012 and officially announced in 2014 along with the release of a patch that fixed the vulnerability. The bug didn't directly exploit sessions; instead, it took advantage of a security flaw in servers running OpenSSL.

The attack allowed hackers to download 64 kilobytes of private memory on a server. Repeatedly running the attack allowed hackers to download a variety of sensitive data from these servers, including cookies, email addresses, and passwords. Additionally, session keys could also leak. The infographic below from Creately shows how to Heartbleed bug worked in more detail.

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005



1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005



Utilizing Perfect Forward Secrecy

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

Implementing Perfect Forward Secrecy is reasonably simple since it works on sites that use either SSL or TLS. Both SSL and TLS are cryptographic protocols that allow secure connections to be created. Neither of which mandate the actual key exchange nor determine the encryption cipher to be used.

Instead, the server and the user's machine must agree upon a type of encryption. To make sure that Perfect Forward Secrecy is used, you simply need to set your servers up to make the suitable cipher suites available. Currently, key exchanges that are compliant are:

- Ephemeral Diffie-Hellman (DHE) and
- Ephemeral Elliptic Curve Diffie-Hellman (ECDHE).

These key exchanges need to be set up to be ephemeral, meaning that the keys will only be used once, and after the transaction is complete, the encryption related to the exchange is deleted from the server. This is what ensures that obtaining a session key will be of little to no use to hackers.

A secure connection requires the exchange of keys. But the keys themselves would need to be transferred on a secure connection. That's an important distinction: **You're not *sharing information* during the key exchange, you're *creating a key* together.**

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

There are two possible solutions:

1. Exchange the key by physically meeting and sharing the keys.
2. Somehow established a shared secret on a public unsecure channel. This is easier said than done, and the first such implementation of this is the Diffie-Hellman Scheme.

Properties

Diffie-Hellman makes use of a mathematical function with the following properties:

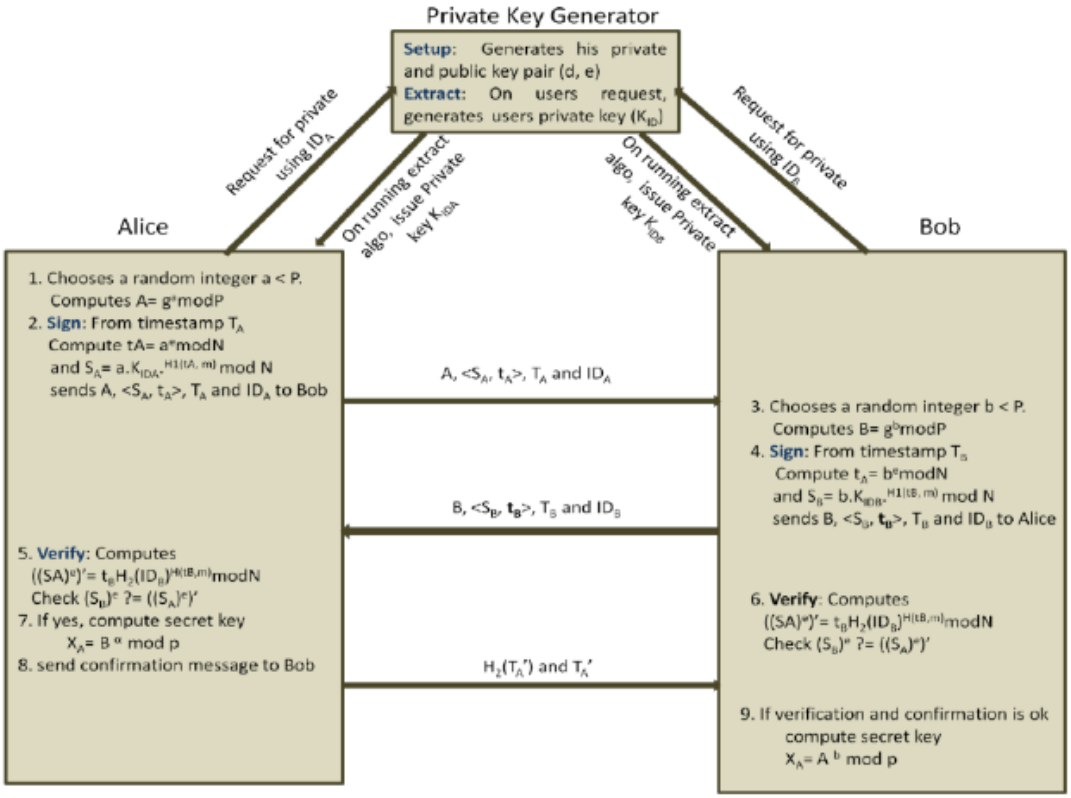
1. It is EASY to compute f[x] (from x)
2. It is HARD to invert f[x] to get x
3. It is EASY to calculate S from A and f[B]
4. It is EASY to calculate S from B and f[A]
5. It is HARD to calculate S without either A or B (even with f[A] and f[B])

How DH scheme works

1. Alice comes out with a random number A. She computes f[A], and sends f[A] to Bob. Alice never discloses her A, not even to Bob.

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

2. Bob comes out with another random number B. He computes f[B], and sends f[B] to Alice. Bob never discloses his B, not even to Alice.
3. Alice computes S using A and f[B]. Bob computes S using B and f[A]
4. Mallory, who is Eavesdropping, has only f[A] and f[B], and so it is HARD for her to calculate S.
5. Alice and Bob now share a common secret which can be used as (or to come up with) a key to establish a secure connection.



Example: Google

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

Google uses cryptographic authentication and authorization at the application layer for inter-service communication. This provides strong access control at an abstraction level and granularity that administrators and services can naturally understand.

They do not rely on internal network segmentation or firewalling as our primary security mechanisms, though they do use ingress and egress filtering at various points in our network to prevent IP spoofing as a further security layer. This approach also helps us to maximize our network's performance and availability.

Each service that runs on the infrastructure has an associated service account identity. A service is provided cryptographic credentials that it can use to prove its identity when making or receiving remote procedure calls (RPCs) to other services. These identities are used by clients to ensure that they are talking to the correct intended server, and by servers to limit access to methods and data to particular clients.

Google's source code is stored in a central repository where both current and past versions of the service are auditable. The infrastructure can additionally be configured to require that a service's binaries be built from specific reviewed, checked in, and tested source code. Such code reviews require inspection and approval from at least one engineer other than the author, and the system enforces that code modifications to any system must be approved by the owners

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

of that system. These requirements limit the ability of an insider or adversary to make malicious modifications to source code and also provide a forensic trail from a service back to its source.

They have a variety of isolation and sandboxing techniques for protecting a service from other services running on the same machine. These techniques include normal Linux user separation, language and kernel-based sandboxes, and hardware virtualization. In general, we use more layers of isolation for riskier workloads; for example, when running complex file format converters on user-supplied data or when running user supplied code for products like Google App Engine or Google Compute Engine. As an extra security boundary, they enable very sensitive services, such as the cluster orchestration service and some key management services, to run exclusively on dedicated machines.

Example – What I prefer at the application layer

The application layer transport (ALT) handshake protocol is a Diffie-Hellman-based authenticated key exchange protocol that supports both Perfect Forward Secrecy (PFS) and session resumption. The ALT infrastructure ensures that each client and server have a certificate with their respective identities and an Elliptic Curve Diffie-Hellman (ECDH) key that chains to a trusted Signing Service verification key. In ALT, PFS is not enabled by default because these static ECDH keys are frequently updated to renew forward secrecy even if PFS is not used on a handshake. During a handshake, the

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

client and server securely negotiate a shared transit encryption key, and the Record protocol the encryption key will be used to protect. For example, the client and server might agree to a 128-bit key that will be used to protect an RPC session using AES-GCM. use the Handshake protocol to negotiate a Record protocol secret. This protocol secret is used to encrypt and authenticate network traffic. The layer of the stack that performs these operations is called the ALT Record Protocol (ALTRP). ALTRP contains a suite of encryption schemes with varying key sizes and security features. During the handshake, the client sends its list of preferred schemes, sorted by preference. The server chooses the first protocol in the client list that matches the server's local configuration. This method of scheme selection allows both clients and servers to have different encryption preferences and allows us to phase in (or remove) encryption schemes. What about **framing**?

Frames are the smallest data unit in ALT. Depending on its size; each ALTRP message can consist of one or more frames. Each frame contains the following fields:

• Length: a 32-bit unsigned value indicating the length of the frame, in bytes.
This 4-byte length field is not included as part of the total frame length.
• Type: a 32-bit value specifying the frame type, e.g., data frame.
• Payload: the actual authenticated and optionally encrypted data being sent.

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

The maximum length of a frame is 1MB plus 4 length bytes. For current RPC protocols, we further limit the frame length, as shorter frames require less memory for buffering.

Now for the **Payload**. In ALT each frame contains a payload that is integrity protected and optionally encrypted. As of the publication of this paper, ALT supports the following modes:

- AES-128-GCM, AES-128-VCM: AES-GCM and AES-VCM modes, respectively, with 128-bit keys. These modes protect the confidentiality and integrity of the payload using the GCM, and the VCM schemes, respectively.
- AES-128-GMAC, AES-128-VMAC: these modes support integrity-only protection using GMAC and VMAC, respectively, for tag computation. The payload is transferred in plaintext with a cryptographic tag that protects its integrity.

We can support PFS. We instead use frequent certificate rotation to establish forward secrecy for most applications. With TLS 1.2 (and its prior versions), session resumption is not protected with PFS. When PFS is enabled with ALT, PFS is also enabled for resumed sessions. The 128-bit AES-GCM scheme is based on NIST 800-38D, and AES-VCM is discussed in details in AES-VCM, An AES-GCM Construction Using an Integer-Based Universal Hash Function. Parts of these concepts are a function of the trust model developed by policy. Still on going.

ALT performs authentication primarily by identity rather than host. Every network entity (e.g., a corporate user, a physical machine, or a production service) has an associated identity. These identities are embedded in ALT certificates and used

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

for peer authentication during secure connection establishment. I still have to work on the certificate issuance portion. An idea might be ALT performs authentication primarily by identity rather than host. These entities on the network need to be provisioned with handshake certificates. First, the issuer obtains a master certificate signed by the signing service and optionally passes it down to the entity. Then, a handshake certificate is created and signed by the associated master private key.

Ideas for the Handshake Certificate. Handshake certificate is created and signed locally by the master private key. This certificate contains the parameters used during the ALT handshake (secure connection establishment), for example, static Diffie-Hellman (DH) parameters and the handshake ciphers. Also, the handshake certificate contains the master certificate that it is derived from, i.e., the one associated with the master private key that signs the handshake certificate.

I also need to address CRL. More to follow . . . . ..

References

https://tools.ietf.org/html/draft-arkko-eap-aka-pfs-01

Haverinen, H., Ed. and J. Salowey, Ed., "Extensible Authentication Protocol Method for Global System for Mobile Communications (GSM) Subscriber Identity Modules (EAP-SIM)", RFC 4186, DOI 10.17487/RFC4186, January 2006, <https://www.rfc-editor.org/info/rfc4186>.

1177 Branham Lane #345
San Jose, CA 95118
Web: p3idtech.com
Email: security@p3idtech.com
Main: 408-785-2005

https://www.sans.org/reading-room/whitepapers/bestprac/correctly-implementing-secrecy-35842

http://www.office.xerox.com/latest/SECGD-01UU.PDF

https://manuals.konicaminolta.eu/bizhub-C554-C454-C364-C284-C224/EN/contents/id08-0464.html
https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security/