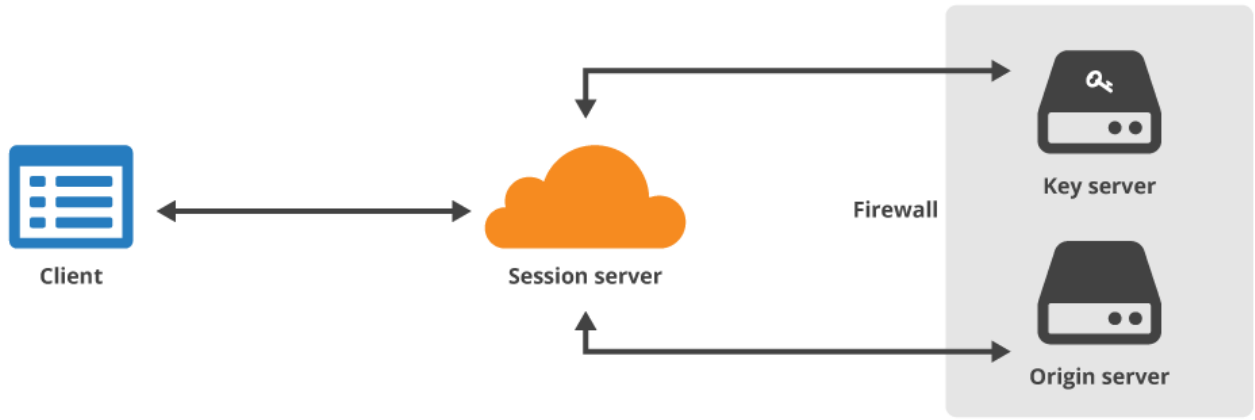
Keyless SSL: The Nitty Gritty Technical Details

19 Sep 2014 by Nick Sullivan.



We announced Keyless SSL (https://blog.cloudflare.com/announcing-keyless-ssl-all-the-benefits-of-cloudflare-without-having-to-turn-over-your-private-ssl-keys/) yesterday to an overwhelmingly positive response. We read through the comments on this blog, Reddit (http://www.reddit.com/r/programming/comments/2grd1d/cloudflare_annouces_keyless_ssl/), Hacker News (https://news.ycombinator.com/item?id=8334933), and people seem interested in knowing more and getting deeper into the technical details. In this blog post we go into extraordinary detail to answer questions about how Keyless SSL was designed, how it works, and why it's secure. Before we do so, we need some background about how encryption works on the Internet. If you're already familiar, feel free to skip ahead (#makingitkeyless).

TLS

Transport Layer Security (TLS) is the workhorse of web security. It lets websites prove their identity to web browsers, and protects all information exchanged from prying eyes using encryption. The TLS protocol has been around for years, but it's still mysterious to even hardcore tech enthusiasts. Understanding the fundamentals of TLS is the key to understanding Keyless SSL.

Dual goals

TLS has two main goals: confidentiality and authentication. Both are critically important to securely communicating on the Internet.

Communication is considered confidential when two parties are confident that nobody else can understand their conversation. Confidentiality can be achieved using symmetric encryption: use a key known only to the two parties involved to encrypt messages before sending them. In TLS, this symmetric encryption is typically done using a strong block cipher like AES (http://en.wikipedia.org/wiki/Advanced_Encryption_Standard). Older browsers and platforms might use a cipher like Triple DES (http://en.wikipedia.org/wiki/Triple_DES) or the stream cipher RC4 (http://en.wikipedia.org/wiki/RC4), which is now considered insecure (http://blog.cloudflare.com/killing-rc4-the-long-goodbye/).

The other crucial goal of TLS is authentication. Authentication is a way to ensure the person on the other end is who they say they are. This is accomplished with public keys. Websites use certificates and public key cryptography to prove their identity to web browsers. And browsers need two things to trust a certificate: proof that the other party is the owner of the certificate, and proof that the certificate is trusted.

A website certificate contains a public key, and if the website can prove that it controls the associated private key, that's proof that they are the owner of the certificate. A browser considers a certificate trusted if the certificate was granted by a trusted certificate authority, and contains the site's domain name. More technical details of how trust works with web certificates is described in a previous blog post (http://blog.cloudflare.com/introducing-cfssl/) about our open source SSL toolkit, CFSSL.

In the context of the web, confidentiality and authentication are achieved through the process of establishing a shared key and proving ownership of a certificate. TLS does this through a series of messages called a "handshake".

What's in a handshake?

The TLS protocol evolved from the Secure Sockets Layer (SSL) protocol which was developed by Netscape in the mid-1990s. In 1999, the Internet Engineering Task Force (IETF) standardized a new protocol called TLS, which is an updated version of SSL. In fact, TLS is so similar to SSL that TLS 1.0 uses the SSL protocol version number 3.1. This may seem confusing at first, but makes sense since TLS is just a minor update to SSL 3.0. Subsequent versions of TLS have followed this pattern. Since TLS is an evolution of the SSL protocol, people still use the terms TLS and SSL somewhat interchangeably.

There are two main types of handshakes in TLS: one based on RSA (http://en.wikipedia.org/wiki/RSA_(cryptosystem)), and one based on Diffie-Hellman (http://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange). RSA and Diffie-Hellman were the two algorithms which ushered in the era of modern cryptography, and brought cryptography to the masses. These two handshakes differ only in how the two goals of key establishment and authentication are achieved:

	Key establishment	Authentication
RSA handshake	RSA	RSA
DH handshake	DH	RSA/DSA

The RSA and DH handshakes both have their advantages and disadvantages. The RSA handshake only uses one public key algorithm operation, RSA. A DH handshake with an RSA certificate requires the same RSA operation, but with an additional DH operation. Given that the certificate is RSA, the RSA handshake is faster to compute. Public key algorithms like RSA and DH use a lot of CPU and are the slowest part of the TLS handshake. A laptop can only perform a couple hundred RSA encryptions a second versus around ten million per second of the symmetric cipher AES.

The DH handshake requires two algorithms to run, but the advantage it brings is that it allows key establishment to happen independently of the server's private key. This gives the connection forward secrecy (http://blog.cloudflare.com/staying-on-top-of-tls-attacks/), a useful property that protects conversations from being decrypted after the fact if the private key is somehow exposed. The DH version of the handshake also opens up the possibility of using non-RSA certificates that can improve performance, including ECDSA keys (http://blog.cloudflare.com/ecdsa-the-digital-signature-algorithm-of-a-better-internet/). Elliptic curves provide the same security with less computational overhead. A DH handshake with and elliptic curve DSA certificate and elliptic curve Diffie-Hellman key agreement can be faster than a one-operation RSA handshake.

CloudFlare supports both handshakes, but, as we will describe later, the type of handshake used is chosen by the server. CloudFlare will choose a DH handshake whenever we can.

TLS Glossary

Before we walk through the steps of the handshake, here are a couple definitions.

1. Session key

This is the end result of a handshake. It's a key for a symmetric cipher, and allows the client and server to encrypt messages to each other.

2. Client random

This is a sequence of 32 bytes created by the client. It's unique for each connection, and is supposed to contain a four-byte timestamp followed by 28 random bytes. Recently, Google Chrome switched to using 32 bytes of random in order to prevent client fingerprinting. These random values are often called a nonce (http://en.wikipedia.org/wiki/Cryptographic_nonce).

3. Server random

A server random is the same as the client random except generated by the server.

4. Pre-master secret

This is a 48-byte blob of data. It can be combined with both the client random and the server random to create the session key using a "pseudorandom function" (PRF).

5. Cipher suite

This is a unique identifier for combining algorithms making up a TLS connection. It defines one algorithm for each of the following:

- key establishment (typically a Diffie-Hellman variant or RSA)
- authentication (the certificate type)
- confidentiality (a symmetric cipher)
- integrity (a hash function)

For example "AES128-SHA" defines a session that uses:

- RSA for key establishment (implied)
- RSA for authentication (implied)
- 128-bit Advanced Encryption Standard (http://en.wikipedia.org/wiki/Advanced_Encryption_Standard) in Cipher Block Chaining (CBC) mode (https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation) for confidentiality
- 160-bit Secure Hashing Algorithm (SHA) (http://en.wikipedia.org/wiki/SHA-1) for integrity

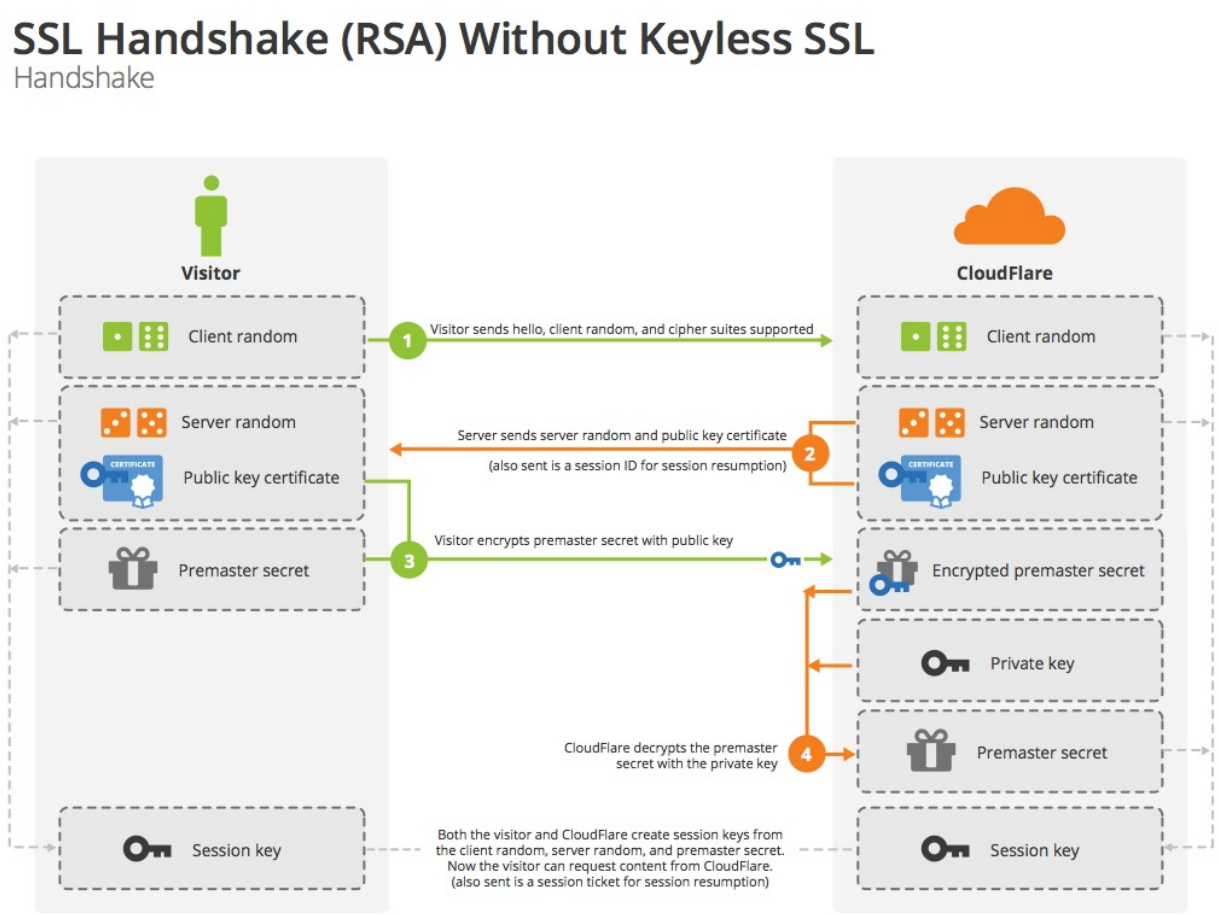
A more daunting, but valid cipher suite is "ECDHE-ECDSA-AES256-GCM-SHA384" which defines a session that uses:

- Elliptic Curve Diffie-Hellman Ephemeral (ECDHE (http://en.wikipedia.org/wiki/Elliptic_curve_Diffie%E2%80%93Hellman)) key exchange for key establishment
- Elliptic Curve Digital Signature Algorithms (ECDSA (http://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm)) for authentication
- 256-bit Advanced Encryption Standard (http://en.wikipedia.org/wiki/Advanced_Encryption_Standard) in Galois/Counter mode (GCM) (http://en.wikipedia.org/wiki/Galois/Counter_Mode) for confidentiality
- 384-bit Secure Hashing Algorithm (http://en.wikipedia.org/wiki/SHA-2) for integrity

With these definitions in hand, let's walk through an RSA handshake.

RSA handshake

Note that none of the messages in the handshake are encrypted with a session key; they are all sent in the clear.



(/content/images/2014/Sep/ssl_handshake_rsa.jpg)

Message 1: "Client Hello"

The client hello contains the protocol version that the client wants to use, and some other information to get the handshake started including the client random and a list of cipher suites. Modern browsers also include the hostname they are looking for, called the Server Name Indication (SNI) (http://en.wikipedia.org/wiki/Server_Name_Indication). SNI lets the web server host multiple domains on the same IP address.

Message 2: "Server Hello"

After receiving the client hello, the server picks the parameters for the handshake going forward. The choice of cipher suite determines what type of handshake is performed. The server "hello" message contains the server random, the server's chosen cipher suite, and the server's certificate. The certificate contains the server's public key and domain name.

Note: CloudFlare's cipher suite preferences are posted publicly on our Github page (https://github.com/cloudflare/sslconfig).

Message 3: "Client Key Exchange"

After validating that the certificate is trusted and belongs to the site they are trying to reach, the client creates a random pre-master secret. This secret is encrypted with the public key from the certificate, and sent to the server.

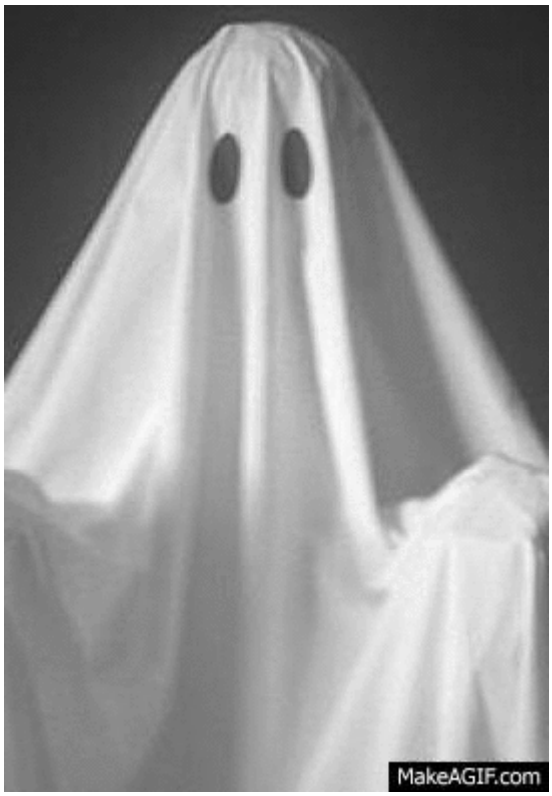
Upon receiving this message, the server uses its private key to decrypt this pre-master secret. Now that both sides have the pre-master secret, and both client and server randoms, they can both derive the same session key. Then they exchange a short message to indicate that the next message they send will be encrypted.

The handshake is officially complete when the client and server exchange "Finished" messages. The actual text is literally: "client finished" or "server finished" encrypted with the session key. Any subsequent communication between the two parties are encrypted with the session key.

This handshake is elegant because it combines key exchange and authentication in one step. The logic is that if the server can correctly derive the session key, then they must have access to the private key, and, therefore, be the owner of the certificate.

The downside of this handshake is that the messages secured by it are only as safe as the private key. Suppose a third party has recorded the handshake and the subsequent communication. If that party gets access to the private key in the future, they will be able to decrypt the premaster secret and derive the session key. With that they can decrypt the entire message. This is true even if the certificate is expired or revoked. This leads us to another form of handshake that can provide confidentiality even if the private key is compromised.

Ephemeral Diffie-Hellman handshake



The ephemeral Diffie-Hellman handshake is an alternative form of the TLS handshake. It uses two different mechanisms: one for establishing a shared pre-master secret, and one for authenticating the server. The key feature that this relies on is the Diffie-Hellman key agreement algorithm.

In Diffie-Hellman, two parties with different secrets exchange messages to obtain a shared secret. This handshake relies on the simple fact that exponents are commutative. Specifically that taking a number to the power of a, and the result to the power of b, is the same as taking the same number to the power of b, and the result to the power of a.

The algorithm works like this:

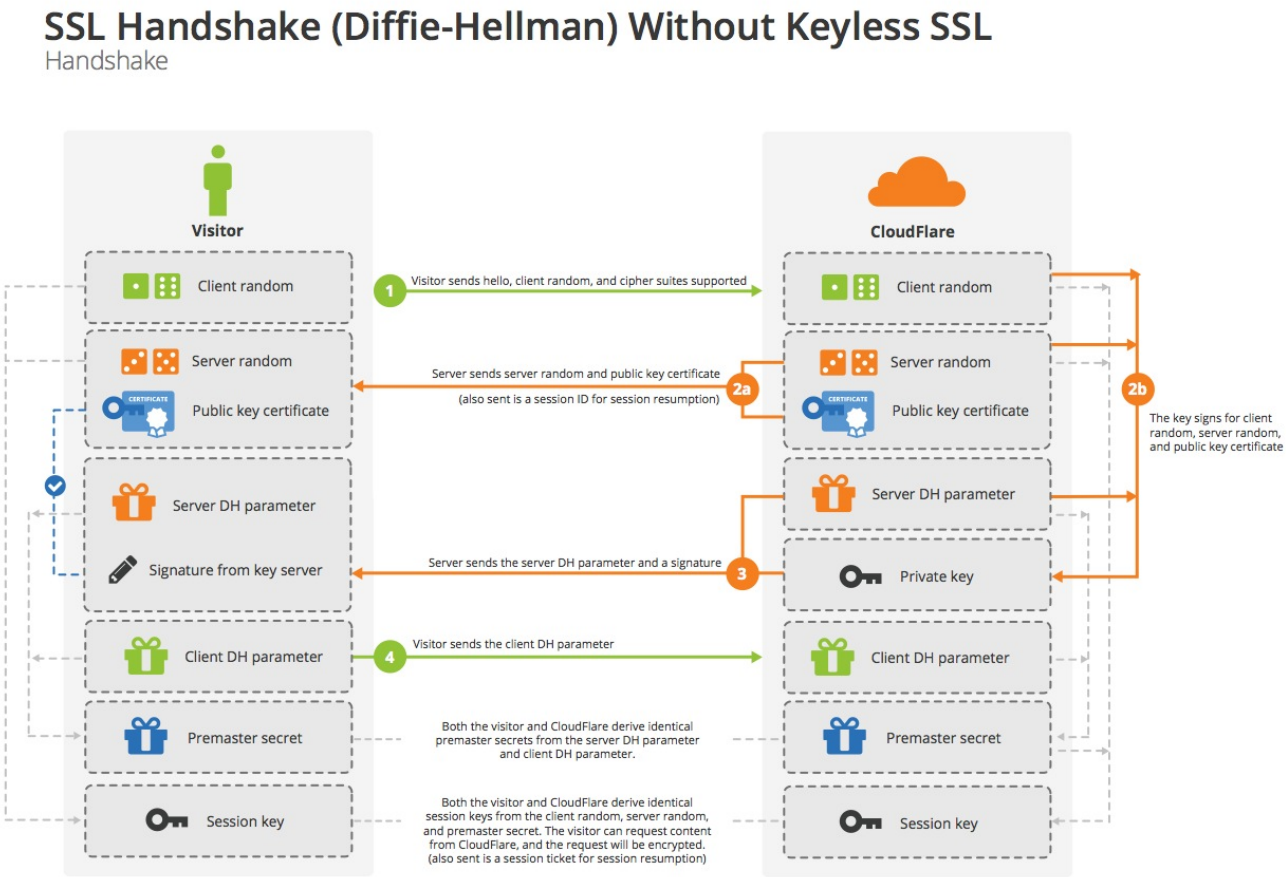
- person a has secret a, sends g^a to person b
- person b has secret b, sends g^b to person a
- person a computes $(g^b)^a$
- person b computes $(g^a)^b$
- Both person a and b end up with g^{ab}, which is their shared secret

This doesn't work well with regular numbers because g^{ab} can get really large, and there are efficient ways to take the nth root of a number. However, we can change the problem space and make it work. This is done by restricting the computation to numbers of a fixed size by always dividing the result of a computation by big prime number and taking the remainder. This is called modular arithmetic. Taking an nth root in modular arithmetic is called the

discrete logarithm problem (http://en.wikipedia.org/wiki/Discrete_logarithm) and is considered a hard problem.

Another variant of the Diffie-Hellman key agreement uses Elliptic Curves, ECDHE. For more information on Elliptic Curves, check out this primer (http://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/) we published last year. A shared secret can be derived using either of these fixed-size Diffie-Hellman key agreement algorithms.

Now let's go through a Diffie-Hellman handshake:



(/content/images/2014/Sep/ssl_handshake_diffie_hellman.jpg)

Message 1: "Client Hello"

Just like in the RSA case, the client hello contains the protocol version, the client random, a list of cipher suites, and, optionally, the SNI extension. If the client speaks ECDHE, they include the list of curves they support. If this is omitted, or there is a mismatch, it can be tricky to debug (http://terinstock.com/blog/2014/07/02/tls-with-erlang.html) .

Message 2: "Server Hello"

After receiving the client hello, the server picks the parameters for the handshake going forward, including the curve for ECDHE. The server "hello" message contains the server random, the server's chosen cipher suite, and the server's certificate.

The RSA and Diffie-Hellman handshakes start to differ at this point with a new message type.

Message 3: "Server Key Exchange"

In order to start the Diffie-Hellman key exchange, the server needs to pick some starting parameters and send them to the client---this corresponds to the g^a we described above. The server also needs a way to prove that it has control of the private key, so the server computes a digital signature of all the messages up to this point. Both the Diffie-Hellman parameters and the signature are sent in this message.

Message 4: "Client Key Exchange"

After validating that the certificate is trusted, and belongs to the site they are trying to reach, the client validates the digital signature sent from the server. They also send the client half of the Diffie-Hellman handshake (corresponding to g^b above).

At this point, both sides can compute the pre-master secret from the Diffie-Hellman parameters (corresponding to g^{ab} above). With the pre-master secret and both client and server randoms, they can derive the same session key. They then exchange a short message to indicate that they the next message they send will be encrypted.

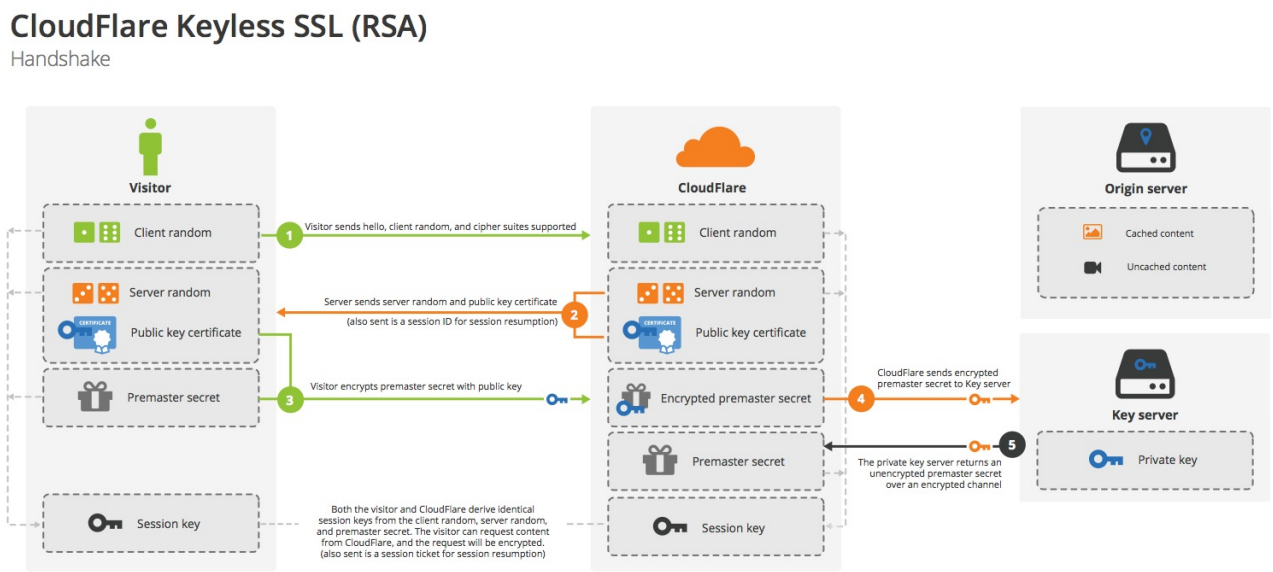
Just like in the RSA handshake, this handshake is officially complete when the client and server exchange "Finished" messages. Any subsequent communication between the two parties are encrypted with the session key.

Making it keyless

Yesterday we announced Keyless SSL (http://blog.cloudflare.com/announcing-keyless-ssl-all-the-benefits-of-cloudflare-without-having-to-turn-over-your-private-ssl-keys/), CloudFlare's solution that allows sites to use CloudFlare without requiring them to give up custody of their private keys.

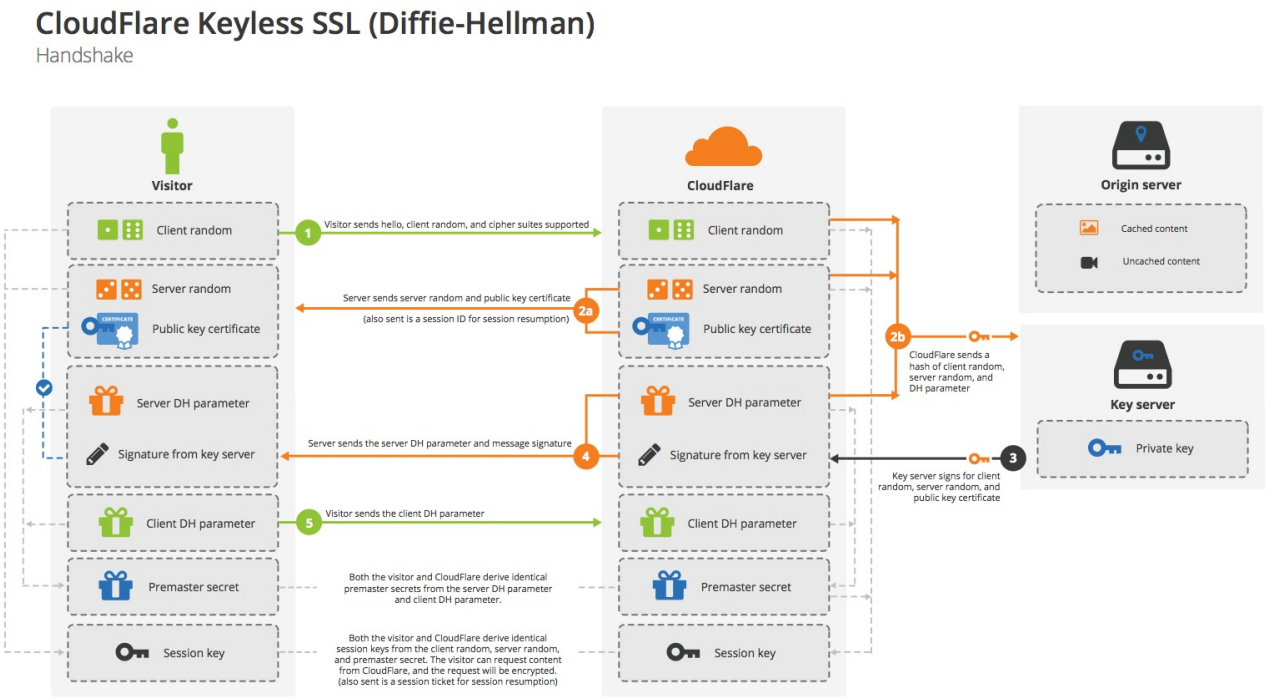
One takeaway from the handshake diagrams above is that the private key is only used once in each handshake. This allows us to split the TLS handshake geographically, with most of the handshake happening at CloudFlare's edge while moving the private key operations to a remote key server. This key server can be put on the customer's infrastructure, giving them exclusive access to the private key.

Once the secure tunnel is established, the RSA handshake looks like this:



(/content/images/2014/Sep/cloudflare_keyless_ssl_handshake_rsa.jpg)

The DH handshake looks like this:



(/content/images/2014/Sep/cloudflare_keyless_ssl_handshake_diffie_hellman.jpg)

Extending the TLS handshake in this way required changes to the NGINX server and OpenSSL to make the private key operation both remote and non-blocking (so NGINX can continue with other requests while waiting for the key server). Both the NGINX/OpenSSL changes, the protocol between the CloudFlare's server, and the key server were audited by iSEC Partners and Matasano Security. They found the security of Keyless SSL equivalent to on-premise SSL. Keyless SSL has also been studied by academic researchers from both provable security and performance angles.

The key server can run on Linux (packaged for Red Hat/CentOS, Debian and Ubuntu, and others), other UNIX operating systems (including FreeBSD), and Microsoft Windows Server. Customers also get access to a reference implementation (https://github.com/cloudflare/keyless) written in C, so they can build their own compatible key server.

The key server will soon be integrated with hardware security module (HSM) vendors and key management solutions (such as Venafi) to provide customers with additional ways to control how the keys are managed in their infrastructure.

Keyless SSL supports multiple key servers for the same certificate. Key servers are stateless, allowing customers to use off-the-shelf hardware and scale the deployment of key servers linearly with traffic. By running multiple key servers and load balancing via DNS, the customer's site can be kept highly available.

Protecting the oracle

For Keyless SSL to be secure, the connection from CloudFlare's edge to the key server also needs to be secure. The key server can act as a cryptographic oracle by performing private key operations for anyone who can contact it. Ensuring that only CloudFlare can ask the key server to perform operations is crucial to the security of Keyless SSL.

We secure the connection from CloudFlare to the key server with mutually authenticated TLS. Previously, we described TLS handshakes that were only authenticated in one direction: the client validated the server. In mutually authenticated TLS, both client and server have certificates and authenticate each other. The key server authenticates CloudFlare and CloudFlare authenticates the key server.

In Keyless SSL, the key server only allows connections from clients with a certificate signed by a CloudFlare internal certificate authority. We use certificates granted by our own certificate authority for both sides of this connection. We have strict controls over how these certificates are granted and use the X.509 Extended Key Usage (http://en.wikipedia.org/wiki/X.509#Extensions_informing_a_specific_usage_of_a_certificate) option to ensure that certificates are only used as intended. This prevents any party that doesn't have a CloudFlare granted certificate from communicating with the key server. Customers also have the option to add firewall rules to limit incoming connections to those from CloudFlare's IP space (https://www.cloudflare.com/ips).

Additionally, we restrict the cipher suite for this connection to one of the following:

- ECDHE-ECDSA-AES256-GCM-SHA384
- ECDHE-RSA-AES256-GCM-SHA384

These are two of the strongest ciphers available in OpenSSL and guarantee the connection between CloudFlare and the keyserver has perfect forward secrecy.

Other security considerations

The key server itself can be modified to work with hardware security module (HSM), providing additional hardware security for customers who want to protect the key server from undiscovered software vulnerabilities similar to Heartbleed (http://blog.cloudflare.com/searching-for-the-prime-suspect-how-heartbleed-leaked-private-keys/).

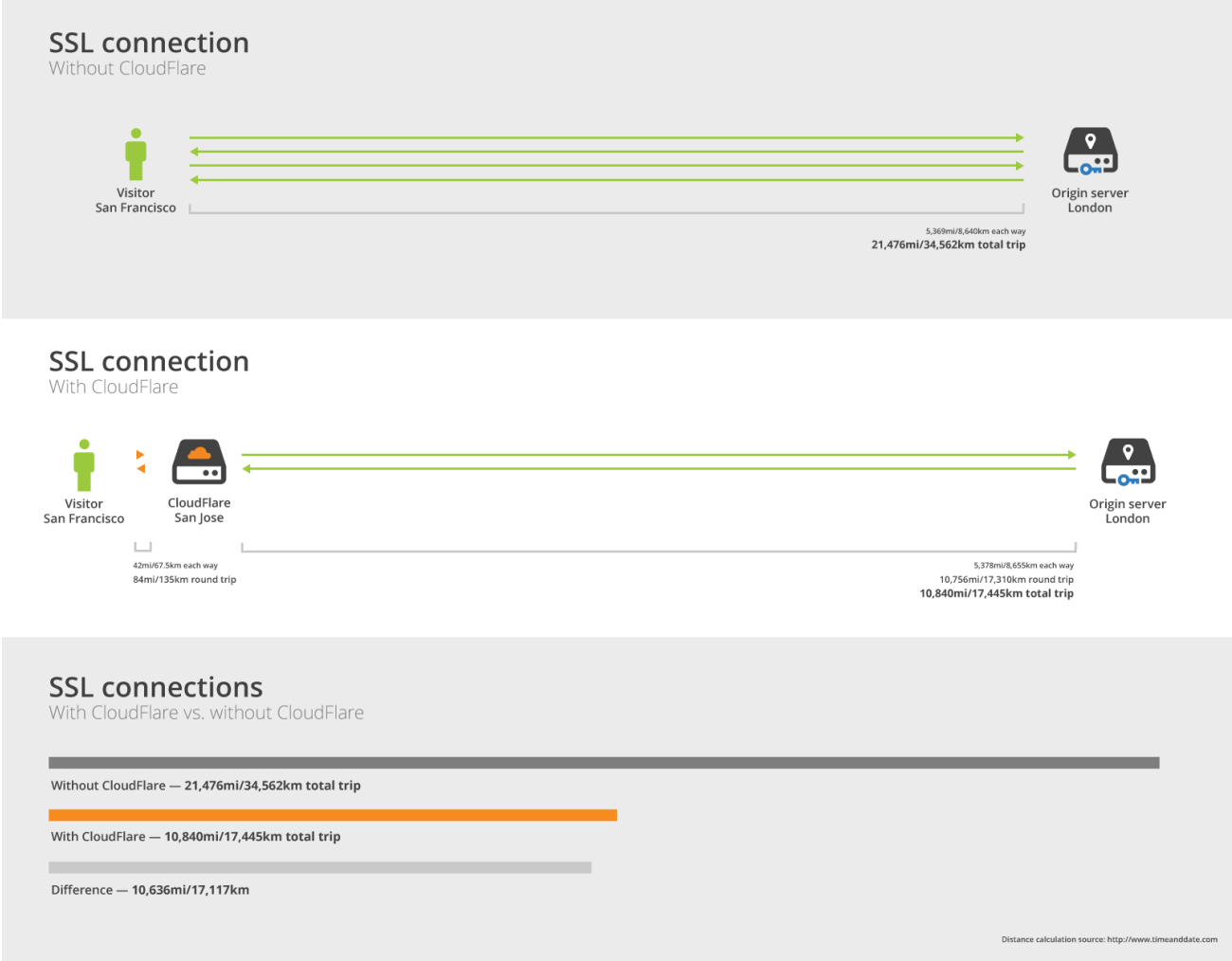
The key server is not subject to padding oracle attacks like that of Bleichenbacher (http://en.wikipedia.org/wiki/Adaptive_chosen-ciphertext_attack) because it uses constant size responses. Side-channel attacks such as timing attacks are ineffective as long as the underlying cryptographic library used on the key server is immune. We use OpenSSL in our reference implementation which has been hardened against such attacks.

Performance enhancements

CloudFlare is designed to make sites faster: it should take less time to connect to a site on CloudFlare than the same site off CloudFlare. This is also the case with Keyless SSL. Connecting to a site with Keyless SSL should be faster than connecting to the same site with CloudFlare disabled. People have asked how can that be, given that the Keyless SSL requires an additional connection to the key server. The answer lies in geography.

CloudFlare's data centers are geographically distributed in 20 countries around the world and are located within less than 20ms of 95% of the Internet's active population. This allows visitors to communicate with a CloudFlare server that is closest to them on the network. Messages sent between visitor and CloudFlare don't have to travel far, so the connection latency is smaller. This proximity effect is one of the ways that CloudFlare accelerates websites.

In the Keyless SSL diagrams above, all the messages except one are traveling over the short link between CloudFlare and the visitor. The only long round trip is the one to the key server.



(/content/images/2014/Sep/illustration-ssl-with-cf-and-without.png)

Consider the scenario where a visitor in San Francisco wants to visit a site hosted in London over TLS. Without CloudFlare, the TLS handshake requires two round-trips from San Francisco to London. With Keyless SSL and a keyserver hosted in London, the visitor will end up in CloudFlare's nearby San Jose (http://blog.cloudflare.com/and-then-there-were-threecloudflares-new-data/) data center. In this scenario, only one of the messages has to travel to London and back. Messages have to travel a shorter distance, resulting in a faster handshake.

The reason we only require one round-trip to the key server is persistent connections. Once CloudFlare has connected to a key server, it keeps the connection ready for any new visitors to the site. The first connection to a Keyless SSL powered site is fast, but the major performance improvement comes when a visitor returns to the site.

Abbreviated handshake

TLS provides an excellent performance feature called "session resumption". If a client has previously established a session with the server, and is trying to

connect again, they can use an abbreviated handshake. There are two mechanisms to do so: session IDs and session tickets.

Session IDs require the server to keep the session state (i.e. the session key) ready in case a previous session needs to be resumed. In the case of session tickets, the server sends a session ticket (consisting of the session key encrypted with a ticket key) to the client during the initial handshake. When resuming a session, the client sends the encrypted key back to the server who decrypts it and resumes the session. There is no need to use the private key for session resumption.

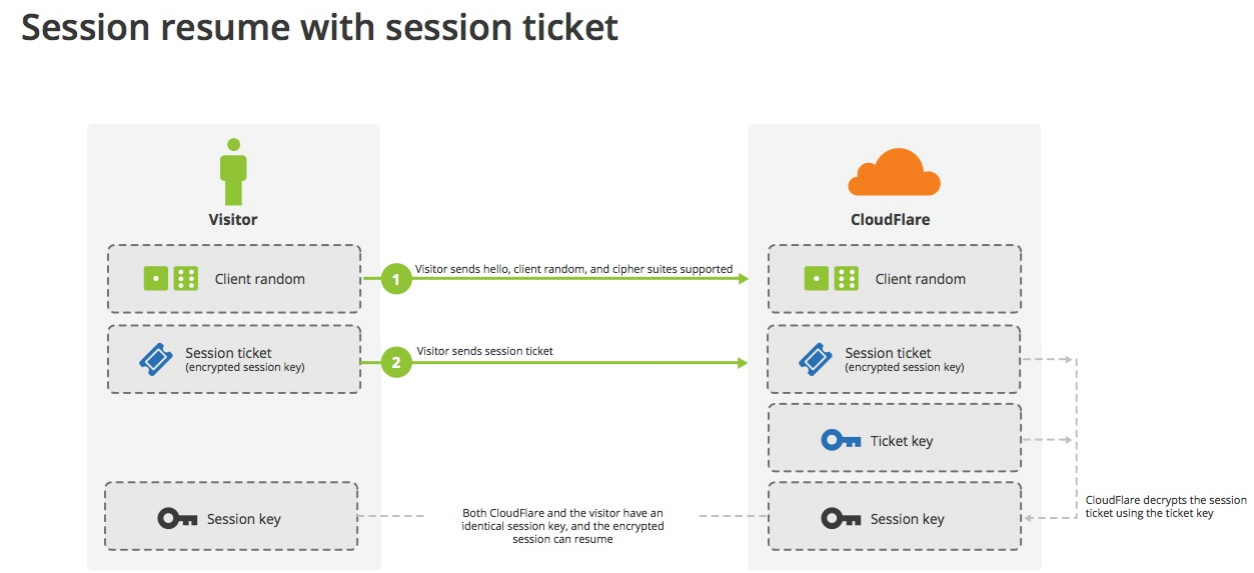
Firefox and Chrome are the major browsers that support session tickets. All other modern browsers support resumption via session IDs. One of the challenges faced when using these techniques at scale is load balancing. In order for a server to resume a connection, it needs to have the previously established session key. If the visitor tries to resume a connection with a new server, that server needs to get the original session key somehow.

The main problem with session resumption is that is was not meant to scale to load-balanced servers. If a client starts a session on one server, it cannot resume that session on another server. This is not a failing of the protocol, just a missing feature in open source web servers.

With Keyless SSL, we are introducing advanced session resumption capabilities to solve this problem. This includes worldwide session resumption via session tickets and session resumption within a data center via session IDs. Session resumption allows repeat visitors to have lightning fast connection times because there is no need to go back to the key server to resume a connection.

Session ticket resumption

With session tickets, we can resume a session from any machine on our network. This required significant engineering work that we are opening up to the community.



(/content/images/2014/Sep/session_resumption_with_session_ticket.jpg)

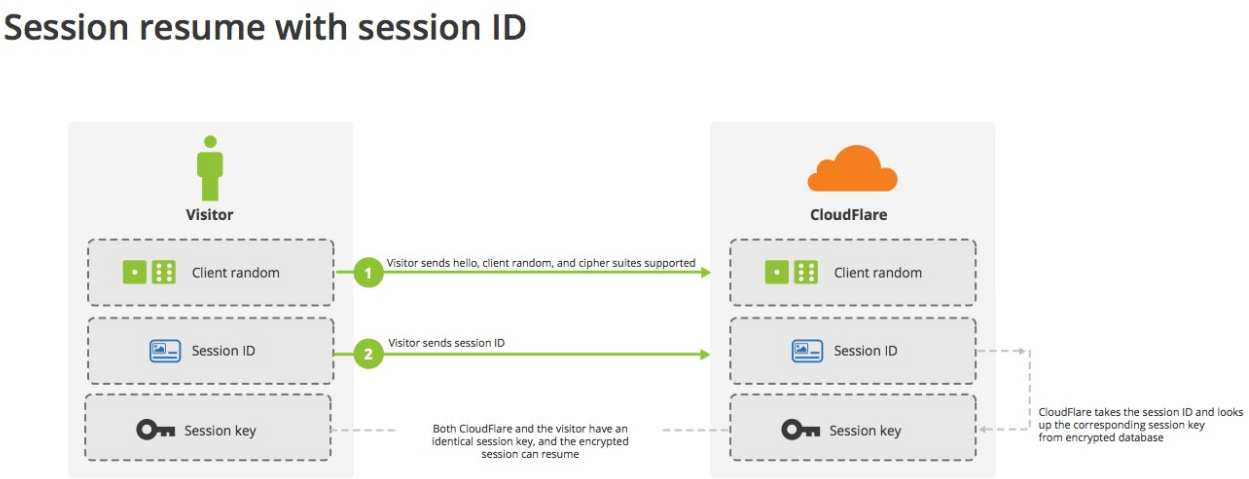
Twitter recently announced (https://blog.twitter.com/2013/forward-secrecy-at-twitter) that they are using session ticket keys rotated every 12 hours. We are upping the ante by rotating session ticket keys every hour. We built a centralized session ticket key generator that issues new keys every hour for distribution across our global network. Each key persists for a user-configurable amount of time (defaulting to 96 hours), after which it is permanently deleted. To distribute the keys, we added a TLS layer to the key-value store Kyoto Tycoon (http://blog.cloudflare.com/kyoto_tycoon_with_postgresql/) so that replication is fully encrypted with mutually authenticated TLS and pinned to CloudFlare's CA. With Kyoto Tycoon, ticket keys are replicated globally within seconds to every one of our edge machines. In keeping with our open source philosophy (http://blog.cloudflare.com/keeping-our-open-source-promise/), we plan on open sourcing our changes to Kyoto Tycoon. With the ticket keys available on every server, we can resume any connection on any machine in our entire network.

The rotation of ticket keys helps us maintain perfect forward secrecy for our customers while reducing latency for returning visitors using Firefox and Chrome.

Session ID resumption

We can also resume sessions across multiple machines with a session ID. Unlike with session tickets, we can only resume sessions within a datacenter. This turns out to be good enough for 99.99% of users because CloudFlare's Anycast network directs requests to the nearest data center. Browsers don't typically move between cities very often, so most resumption requests end

up in the same place. And, in the worst case, such as if you access a site from your phone in one city and then fly to another, your client will simply setup a new session.



(/content/images/2014/Sep/session_resumption_with_session_id.jpg)

We can resume sessions using an ID by caching session keys within a datacenter. For each new connection, we cache an encrypted versions of the session key in a centralized location, indexed by session ID. If a new request come in with a session ID that has been seen before, we look it up in the central store that can be accessed from all the servers in any data center. These session keys do not leave the datacenter and persist for a user-configurable amount of time, again defaulting to 96 hours. Session ID caching lets us use an abbreviated handshake for almost all resumed connection attempts in browsers other than Chrome or Firefox.

Other technically sophisticated organizations also use session resumption. Google, for instance, uses a similar technique to resume sessions across its infrastructure. **Note: All CloudFlare customers with SSL enabled now get the benefits of this advanced session resumption, even if they are not using Keyless SSL.**

Open Source

CloudFlare developed a lot of code when building Keyless SSL and have contributed major portions of it back to the community:

Strict SSL: this code allows upstream connections from NGINX to validate TLS connections, needed for validating the identity of the key server. This change (http://mailman.nginx.org/pipermail/nginx-announce/2014.txt) was merged into NGINX.

Session tickets: we added support for session ticket in NGINX. This change (http://mailman.nginx.org/pipermail/nginx-devel/2013-October/004370.html) was merged into NGINX.

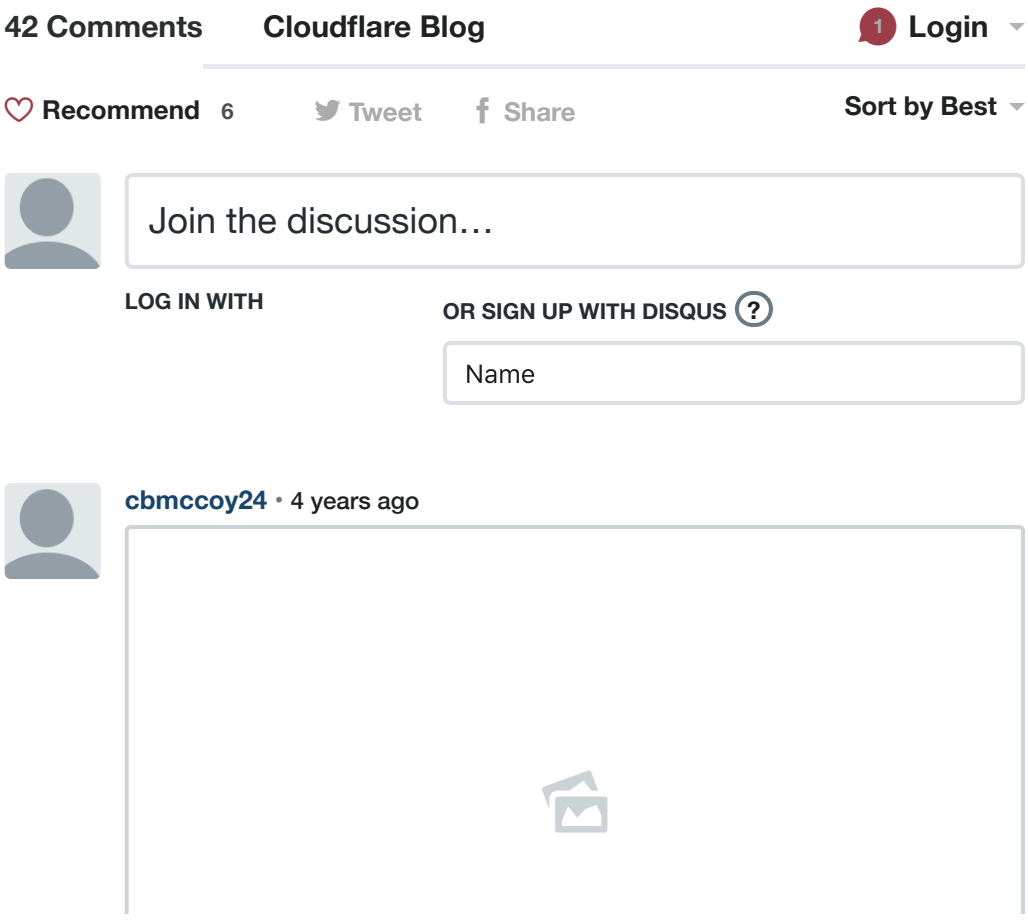
CFSSL: we recently open sourced the tool we use for our internal certificate authority. It is available on GitHub (http://blog.cloudflare.com/introducing-cfssl/) .

Kyoto Tycoon: we are soon open sourcing our changes to Kyoto Tycoon, a high performance key value store we use extensively, to allow mutually authenticated replication.

Key Server: The reference implementation of the Keyless SSL key server is now available on Github (https://github.com/cloudflare/keyless) .

Why it matters

Keyless SSL is a big advancement allowing website owners to use a service like CloudFlare to make their website faster and more secure, while retaining control of their private keys. As we said in the previous post announcing it (https://blog.cloudflare.com/announcing-keyless-ssl-all-the-benefits-of-cloudflare-without-having-to-turn-over-your-private-ssl-keys/) , Sebastien was able to build the initial Keyless SSL prototype overnight. Making sure it was secure, fast, and could scale is what took us two years of engineering. Now, with persistent connections and advanced session resumption techniques, using Keyless SSL is not only safe, it's blazing fast!



I loved reading this. Good job!

18 ^ v • Reply • Share ›

Luca Moser • 4 years ago

With Keyless SSL you are only preventing the theft of private keys in case a server gets hacked, period. However, CloudFlare is able to read all the encrypted traffic. I wouldn't trust CloudFlare with my bank data at all. I wonder if those banks really understand how Keyless SSL really works, since I can't imagine them to be so stupid to allow CloudFlare (or really any other third party) being able to read my private bank data.

6 ^ v • Reply • Share ›

Carl ➚ Luca Moser • 3 years ago

LOL. Tons of small banks in the US use the same white label software and app. If you trust your bank not to outsource, you are being foolish. The banks do only as much as they are legally obligated to do.

Edit: I looked up who my last two banks have been outsourcing to. It's https://en.wikipedia.org/wi...

^ v • Reply • Share ›

Robert ➚ Luca Moser • 3 years ago

This isn't so much an issue with keyless SSL as simply using a third-party to provide your infrastructure, keyless or not. If your business is so confidential you can't trust this third party then you cant, but if it's something less confidential and yet you still don't want to submit your private keys then it's fine.

^ v • Reply • Share ›

Csaba Vegso ➚ Luca Moser • 4 years ago

I am trying to look at this solution from the customers' point of view .... Correct me if I am wrong but in my opinion CloudFlare's Keyless SSL = forcing bank's customers to implicitly trust in an entity (CloudFlare) they normally do not trust (even they do not know about)

^ v • Reply • Share ›

Luca Moser ➚ Csaba Vegso • 4 years ago

Yes, this is what would happen if banks would use CloudFlare's Keyless SSL. In my opinion, it's a much higher security risk having a third party being able to read encrypted bank data, than having the old

system where I directly communicate with the bank servers. It's CloudFlare's infrastructure and I have absolutely no transparency what they will do with my data. The question is, why can't banks not just do the same thing, without having CloudFlare? I am sure banks can afford a group of system engineers being capable of building the same system.

Reply • Share ›

wayne • 4 years ago

"Extending the TLS handshake in this way required changes to the NGINX server and OpenSSL to make the private key operation both remote and non-blocking (so NGINX can continue with other requests while waiting for the key server)."

Could you elaborate the changes to NGINX and OpenSSL?

3 Reply • Share ›

Misiek ➚ wayne • 2 years ago

Am I missing something or it's still closed-source?

Reply • Share ›

Ryan Mod ➚ Misiek • 2 years ago

The changes to OpenSSL and nginx are still closed source, but we have a Go implementation available.

https://github.com/cloudfla...

Reply • Share ›

Kibet • 4 years ago

Slightly off-topic, Which tool did you use to create those diagrams?

3 Reply • Share ›

misterparker • 4 years ago

Loved reading this. Very informative, and clearly explained.

3 Reply • Share ›

Andrew • 4 years ago

In Keyless SSL, even though Cloudflare doesn't know the private key, won't they know the session key, thus allowing them to decrypt and read traffic to and from the client?

2 Reply • Share ›

robertjpayne ➚ Andrew • 4 years ago

That's definitely true but the security scope is limited only to the sessions the key server hands out session keys for.

The goal here is that if there is a security breach the private

key is not exposed or at risk. While someone could use the existing session keys until they expire they could not spoof or intercept sessions forever which is the biggest risk of handing over private keys.

The goal for CloudFlare was to provide enterprise customers with an option to outsource their protection without risking private key leaks which have huge implications in the banking industry as noted by their previous post.

2 ^ v • Reply • Share ›

Matt Vlasach ➚ robertjpayne • 4 years ago

This is a very cool engineering feat, and seems to fully achieve the goal of keeping the private key within the enterprise! And great write up to boot.

However, doesn't this just open another security can of worms for enterprise customers? Sure, their private key is safe, and arguably the encryption methods are now better than what they would be otherwise (ECDHE, DH, session keys, etc.), but now they have to worry about a 3rd party provider having the ability to decrypt sessions between their end-users and the origin server. Even though a lot of work has been put into reducing the security scope (a la session keys) on CloudFlare's infrastructure, a compromised CloudFlare node could result in the ability for a nefarious party to intercept and exfiltrate otherwise private data.

So in other words, what is more risky: the compromise/loss of a private SSL key (keeping in mind they can turn on DH on their web servers to achieve PFS themselves), or introducing a 3rd party into an otherwise private 2-party conversation?

I'd love to hear your thoughts on this one, as I'm sure you will hear it from enterprise customers!

2 ^ v • Reply • Share ›

Mark Hammond ➚ robertjpayne • 4 years ago

To shield sensitive data from prying eyes it would make sense to advocate use of another layer of encryption, e.g. javascript based, during login.

1 ^ v • Reply • Share ›

anon • 4 years ago

Wasn't it demonstrated that Elliptic Curve cryptography had been compromised by the NSA via NIST by choosing hidden constants in the algorithms? Why would you be advocating it's use?

in the algorithms? Why would you be advocating it's use?

1 · Reply · Share ›

Eric Lawrence ➚ anon • 4 years ago

No. What was shown is that using *a specific curve* as a random number generator *might* be dangerous as it's not as random as it looks.

6 · Reply · Share ›

paolo • a month ago

Regarding the RSA handshake:

"After validating that the certificate is trusted and belongs to the site
they are trying to reach, the client creates a random pre-master secret. This secret is encrypted with the public key from the certificate, and sent to the server."

I believe this is in the wrong order, since you later clarify:

"The logic is that if the server can correctly derive the session key, then they must have access to the private key, and, therefore, be the owner of the certificate."

If I understand correctly, the client first creates the pre-master secret, encrypts and sends it to the server and only after the server replies it can be authenticated.

· Reply · Share ›

Ashvind Nadar • a year ago

Does this also support outgoing connections? Say if we are hosting an API which has webhook and does call backs ? And we need to have mutual tls between API consumer and provider.

· Reply · Share ›

Bin Ni • 2 years ago

As Luca Moser pointed out, the only problem it solves is preventing customer's private key loss when cloudflare server is compromised. A problem which can be solved by customers asking their CA to revoke the old cert and issue a new one. However, in such scenario, the bigger loss is the hacker sniffing the end users' traffic and stealing the private data such as credit card number, which is the whole idea of even using SSL. In addition, instead of getting customer priv key, the hacker can get CF's priv key to communicate to customer's key server. CF will need to contact its CA to revoke and redeploy anyway. (And CF can potentially do this without notifying their customer regarding the hack...)

So when the CF server is hacked, this solution does not give any better security or result in less loss. And customers need to setup

this key server and it is slower compared to CF hosting the private key. The only thing I agree is that this is an engineering marvel.

Reply • Share ›

Ekta Khandelwal • 2 years ago

Very informative article ..loved reading it

Reply • Share ›

Wen Xie • 3 years ago

The key server is not subject to padding oracle attacks like that of Bleichenbacher because it uses constant size responses. What does "constant size responses" means? Would you mind to give me a more detailed explain? Thanks very much! :)

Reply • Share ›

grant • 3 years ago

with DH style without keyless, there seems to be no encryption at all during the generating of session key, while with RSA style, the pre secret will be encrypted with public key. so how to ensure the security of generating of session key in DH stype?

Reply • Share ›

Jay • 4 years ago

First, one of the best tech articles I have read in years. Excellent job Nick and Cloudflare team.

Curious, would like to know a good test case from CloudFlare as to what kinda SSL
customers they intend to sign up, given that they have gone this extent
to come with such a nifty way to alleviate concerns about hosting Private keys.

If corporations really care about their end to end encryption, they would prefer to keep everything in-house (physical/virtual/cloud). If they don't care or understand how this works, they might as well have
Cloudflare store the priv keys !!

Reply • Share ›

Sumit Vij • 4 years ago

I'm curious can CloudFlare just let the traffic pass through and let Key server handle the session, session key, etc. Is that even possible or makes sense? This way CloudFlare won't be able to decrypt traffic. I understand the key server (or similar entity running on the customer infra) has to manage the sessions, but you get the data encryption end-to-end.

Reply • Share ›

Jay ➦ Sumit Vij • 4 years ago

In that case, would defeat the entire purpose of using CloudFlare as it optimizes/caches "a website" contents, they very basic of features it offers. Hence it would need the Session key to encrypt/decrypt response/request streams. Like Luca mentioned, not suitable for Banking / Healthcare and other sensitive/compliance industries and to think about it majority of their applications would be dynamic/real time. Would like to know a good test case from CloudFlare as to what kinda SSL customers they intend to sign in, given that they have gone this extent to come with such a game-changer.

Reply • Share ›

collinmanderson • 4 years ago

It would be neat to have some sort of pgp-style signature on http responses to guarantee that cloudflare doesn't modify them. Though that still doesn't solve the whole problem.

Reply • Share ›

cdnguy • 4 years ago

Can someone please point out how is this feature different than the split proxy ssl method used for many years by all WOC vendors? for example http://www.theregister.co.u...

I understand it's being used as a cloud service and not using appliances, but that's hardly inventing a new method. What am I missing?

Also, a ton of patents cover split proxy ssl, hope you made sure you are not violating any IP here...

Reply • Share ›

Fernando Ike • 4 years ago

The explain is clear and congrats to solution. I'm happy that you send patches to open source projects.

So, I have question about cache and Keyless SSL.

How works cache page/objects in Edge servers? I guess that have two SSL/TLS termination to delivery page/objects directly to Edge servers cache, right?

Reply • Share ›

ronf • 4 years ago

Minor nit: The example AES128-SHA cipher suite should say 160-bit SHA (or SHA-1) for integrity, not 128-bit.

Reply • Share ›

Nick Sullivan ➚ ronf • 4 years ago

Thanks! Fixed

Thanks! Fixed.

1 · Reply · Share ›

Daniel • 4 years ago

Hi !

There is one feature that was announced in your previous post (https://blog.cloudflare.com..., which is not described here :

"When the Raspberry Pi was plugged in, the connections went through from a
browser as they would normally. The lock appeared and the connection
was secured, end-to-end. The minute the Raspberry Pi's power was
disconnected, HTTPS access terminated."

Could you please tell us more about that ? How is this achieved ? What are the technical challenges ?

Thanks !

· Reply · Share ›

Guest • 4 years ago

Hi !

A quick question about one of the features described in your previous post (here : https://blog.cloudflare.com... :

"When the Raspberry Pi was plugged in, the connections went through from a
browser as they would normally. The lock appeared and the connection
was secured, end-to-end. The minute the Raspberry Pi's power was
disconnected, HTTPS access terminated."

How is this achieved ? Is it by continuously polling the "oracles", and invalidating all related session keys if the distant server fails to respond in time ? What are the specific technical challenges of this feature ?

· Reply · Share ›

Cesar S Falcao • 4 years ago

Nice, nice tech.
Some questions :
1- What will be the price?
2- 20 countries around the world and are located within less than 20ms of 95% of the Internet’s active population" I think it won't apply for South America, since there's no CF DC here.
3- Is there a way to DoS the link from CF to the Private key server? And how to prevent it?

Reply • Share ›

Pete S. ➚ Cesar S Falcao • 4 years ago

I can't speak for #1 as I don't work at Cloudflare, but in regards to your other questions:

2. In fact, there are three Cloudflare datacenters in South America: one in Colombia, one in Brazil, and one in Chile. Other than Antarctica, the only continent where they don't have facilities is Africa (I assume that would change at some point). See https://www.cloudflare.com/...
3. Probably, though doing so would require the bad guy to know the IP address of the customer's key server(s). This information is not public, and if the customer sets up their firewalls to only respond to queries from Cloudflare (and not the whole internet) port-scans and the like won't be able to locate it through automated means.

2 Reply • Share ›

Cesar S Falcao ➚ Pete S. • 4 years ago

Thanks. It has been months since I checked the map, very glad for the S. America DCs (maybe I just got used to the lack of support from the tech industry here - I know, taxes, high prices and others complications apply).

Reply • Share ›

eastdakota Mod ➚ Cesar S Falcao • 4 years ago

And Lima and Buenos Aires are coming,