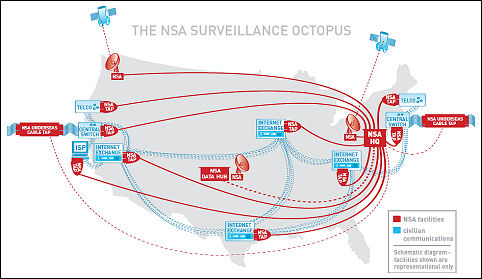
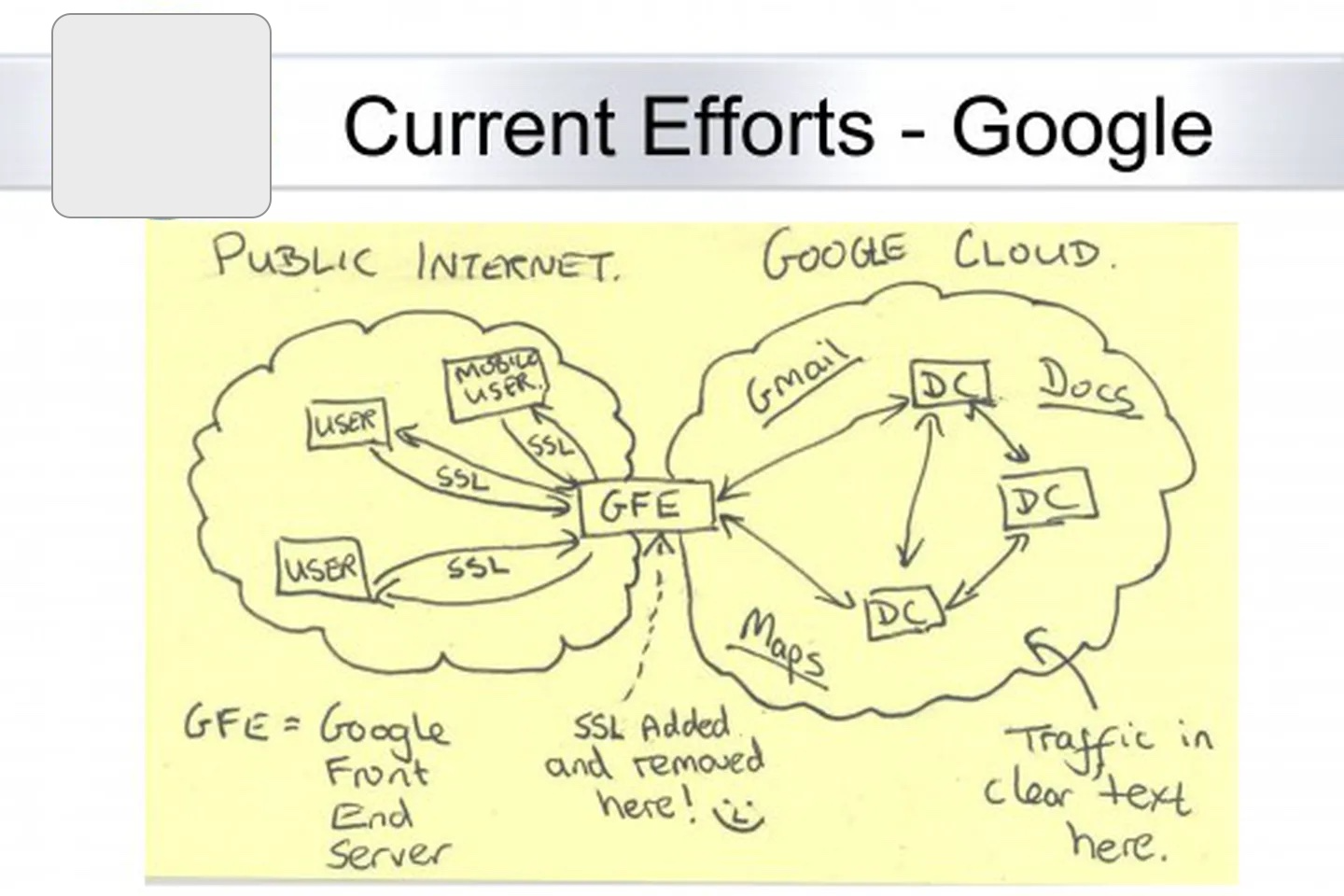
**EazyPGP ­­** Created: 31 October 2023

Author: Samuel Huang Last updated: 1 Nov 2023



NSA surveillance Octopus from Snowdon leaks



NSA muscular Google cloud from Snowdon leaks

**Disclaimer:** the author doesn’t have the slightest clue whether the conspiracies above from internet are real. It’s only here as special effect to grab your attention. The proposed **E2E**(end-to-end) encryption standard **EazyPGP** in this paper could help tackle these imaginary scenarios. Also this is a **work-in-progress PoC** so real Cryptography engineer has every right to say ‘*Haa Amateur’*. 😆

**Introduction**

Apologies if someone gets offended by the pictures in previous page but two pictures are worth a thousand words of why the proposed EazyPGP standard may have value.

*So what the heck is EazyPGP?*

No, it’s not an implementation of **OpenPGP** standard. It’s a proposed message level security(**MLS**), E2E encryption standard using just three hands picked top cryptographic algorithms: PKI asymmetric algorithms ***Curve25519***for encryption and ***Ed25519***fordigital signature plus symmetric algorithm ***ChaCha20-Poly1305***for encrypting payload as opposed to transport level security(**TLS**) using HTTPS/SSL.

*So what’s the point of using* ***MLS*** *when we have* ***TLS*** *already?*

We see and use TLS every day as the padlock icon of address bar in browser(i.e. HTTPS), as the de facto standard to secure internet, to secure communication between two computers, to secure APIs or websites. What’s the point? Good question!

Gee, how to tell it politely without ticking people off.. TLS is kind of rigged for surveillance by govts, big tech, dictator or whatever already. Don’t take my words for it, see it for yourself.

* [How Some Governments Eliminate HTTPS/TLS Encryption!](https://www.youtube.com/watch?v=37irG5pKur8)
* <https://security.stackexchange.com/questions/20803/how-does-ssl-tls-work>
* <https://security.stackexchange.com/questions/6290/how-is-it-possible-that-people-observing-an-https-connection-being-established-w>

*OK, so what does the name EazyPGP stand for?*

The name EazyPGP is inspired by the infamous [OpenPGP](https://www.openpgp.org/) which has a fascinating origin story where the US govt almost sent its inventor Professor Phil Zimmermann to prison as the cryptographic software he created was considered a munition back in the 90s. Later on, the use of OpenPGP never goes mainstream and is considered dying right now for being difficult to use, has lack of support from big tech, many negative feedback and issues. More about this later. So EazyPGP stands for ‘*Easy to use PGP(Pretty Good Privacy)*’ in an attempt to simulate original OpenPGP to create a much (dare I say it) superior and easier to use standard. Note OpenPGP doesn’t support Curve25519 at the time of writing this.

EazyPGP will be discussed under the scenario of securing **API** for the remainder of this paper.

**Chain of TRUST**

Before going down the rabbit hole, it’s necessary to set the stage by explaining clearly how a website (or any entity) can generate a SSL certificate from Root CA(Certificate Authority) for HTTPS, else all you hear may be blah blah then node to pretend you understand later on like in a technical meeting. After all, when it comes to security, the Devil is in the details. First, clarifications.

A PKI certificate is made up of a pair of keys, a public key and a private key. The term ‘**Root CA**’ refers to the top-level certificate(hence Root) that includes both the public key and the private key of the Certificate Authority, including information(i.e. metadata) about the CA, and the digital signature created by signing both the public key and metadata with the private key.

What power does a ‘**Root CA**’ have?

* **Highest Authority:** It’s the top-level authority in a PKI with a self-signed certificate, meaning that it has signed its own public key with its private key. And this ‘**public key**’ is normally referred to as the ‘*trusted Root CA certificate*’ or ‘*Root CA certificate*’ which gets distributed/shared to be pre-installed in devices. Just to clarify, a root CA certificate doesn’t include the private key of root CA. Yes, it’s confusing. A PKI certificate refers to both public key and private key but root CA certificate normally refers to just the public key.
* **Chain of Trust:** The Root CA's primary role is to establish trust. It signs the public keys of intermediate CAs or end entities, creating a chain of trust. Each subsequent certificate in the chain is signed by the private key of the entity or CA immediately above it in the hierarchy. (Alarm bell should start ringing..)
* **Pre-Installed Certificates:** The public key of the Root CA is included in the default trust stores of web browsers, operating systems, and other software. This inclusion is what allows users to trust certificates signed by the Root CA.
* **Critical to PKI Security:** The security and trustworthiness of the entire PKI system depend on the security of the private key of the Root CA. If the private key of the Root CA were compromised, it could potentially undermine the trust in all certificates issued within that PKI.

**How does a Root CA create a new certificate?**

Let’s go through the process of how a Root CA generates a new certificate, which might be a SSL/TLS certificate for securing a website or API:

1. **Generate a Key Pair:**
   * The first step is to generate a new key pair for the entity that will use the certificate. This key pair consists of a public key and a private key. The private key is kept securely by the entity, while the public key is included in the certificate.
2. **Create a Certificate Signing Request (CSR):**
   * The entity (e.g., a web server) creates a Certificate Signing Request (CSR). The CSR includes information about the entity (such as its domain name) and the entity's public key.
3. **Submit CSR to the Root CA:**
   * The entity submits the CSR to the Root CA for approval and signature. This typically involves sending the CSR and any necessary information to the Root CA for verification.
4. **Root CA Verification:**
   * The Root CA verifies the information provided in the CSR to ensure that the entity requesting the certificate is legitimate and has control over the domain for which the certificate is being requested.
5. **Signing the Certificate:**
   * If the verification is successful, the Root CA uses its private key to sign the CSR, creating a new certificate for the entity. This signed certificate includes the entity's public key, information about the entity, and a digital signature from the Root CA.
6. **Issuing the Certificate:**
   * The Root CA issues the signed certificate to the entity. This certificate is now a trusted certificate because it has been signed by the trusted Root CA.
7. **Installing the Certificate:**
   * The entity (e.g., web server) installs the issued certificate along with its private key. The public key from the certificate is made publicly available for users to establish secure connections.
8. **Chain of Trust:**
   * When users connect to the entity (e.g., visit a website over HTTPS), their browsers verify the entity's certificate by checking its digital signature against the public key of the Root CA, which is pre-installed in the browser's trust store.

This process establishes a chain of trust: the Root CA signs an intermediate CA or end-entity certificate, and this signed certificate is trusted because the public key of the Root CA is trusted. The hierarchy can involve multiple levels of CAs, with the Root CA at the top.

**Let’s talk about TRUST**

Now we are ready to highlight the problems of HTTPS/TLS to make it crystal clear else nobody in their right mind would have the motivation to implement, use and support *EazyPGP* standard.

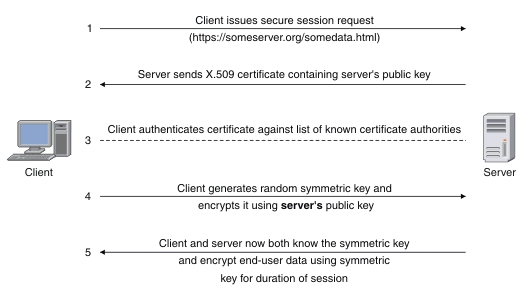
In case the reader never watched the YouTube video in page 2 or still doesn’t see the problem of TLS, I will just spell out the problem. Let’s take a closer look of how SSL handshake takes place where the generation and sharing of symmetric key from client to server takes place. Once the handshake is completed, future communication between client and server can use the same symmetric key for encryption and decryption.

By regurgitating the description from <https://www.ibm.com/docs/en/cics-tg-zos/9.3.0?topic=ssl-how-connection-is-established>, a simplified sequence of SSL handshake can be summarised by the diagram below.

**SSL handshake**

The following diagram shows what happens during an SSL handshake:

Figure 1. SSL handshake



1. The client sends a request to the server for a secure session. The server responds by sending its X.509 digital certificate to the client.
2. The client receives the server's X.509 digital certificate.
3. The client authenticates the server, using a list of known certificate authorities(CA)
4. The client generates a random symmetric key and encrypts it using server's public key.
5. The client and server now both know the symmetric key and can use the SSL encryption process to encrypt and decrypt the information contained in the client request and the server response.

Note this is actually a good example of how hybrid symmetric and asymmetric encryption works to achieve secure communication by leveraging the strengths of both symmetric and asymmetric encryption algorithms. Confused? PKI public key of server is used by the client to encrypt a randomly generated symmetric key (e.g. password to encrypt communication between client and server within a session by AES algorithm) before sending the encrypted symmetric key from client to server, then only server can decrypt the data to get the encryption key by using its PKI private key, hence the name asymmetric encryption.

Even if the encrypted symmetric key is intercepted/monitored by a third party during transit there would be no way to access the symmetric key to decrypt the encrypted communication later on.

So if asymmetric encryption is so cool why not just use it all the way through for communication between client and server? Because asymmetric encryption is generally 10-100 times slower than symmetric encryption(according to Google Bard)! This is because asymmetric encryption uses more complex mathematical operations and longer keys. So it makes sense to only use asymmetric encryption to transport symmetric key, which does the heavy lifting of E2E encryption between both parties.

Later on, we will see how EazyPGP tops this by simulating SSL handshake also with a ***twist***, *no actual symmetric encryption key will ever be sent across the web yet both parties will still be able to derive the same symmetric key*for E2E encryption of all communication within a session. Stay tuned..

Where were we? Yes, let’s take a closer look of the SSL handshake diagram above. What’s wrong with it? What’s wrong is in step 3, the client will **BLINDLY** **trust any server** that can present a certificate signed by a root [Certificate Authority](https://www.ssl.com/faqs/what-is-a-certificate-authority/) [(CA)](https://en.wikipedia.org/wiki/Certificate_authority) certificates. And a computer normally comes pre-installed with a set of such trusted root CA certificates.

Let's break it down. When you connect to a secure website (using HTTPS, for example), the website presents a digital certificate to your browser. This certificate is signed by a CA, and your computer checks whether the CA's signature is valid by verifying it against the root certificates stored on your system. If the CA's certificate is in the list of trusted root CAs, and the presented certificate is properly signed, your browser accepts the connection as secure.

This basically opens up Pandora’s box and virtually any application on your computer can sneak in a root CA certificate to your computer under some legal pretence so all HTTPS traffic from one’s computer could potentially be spied on. And trust me, there are many of these software, e.g. firewall, Anti-Virus, commercial VPNs, free utility software, ..

Here's how it works in more detail:

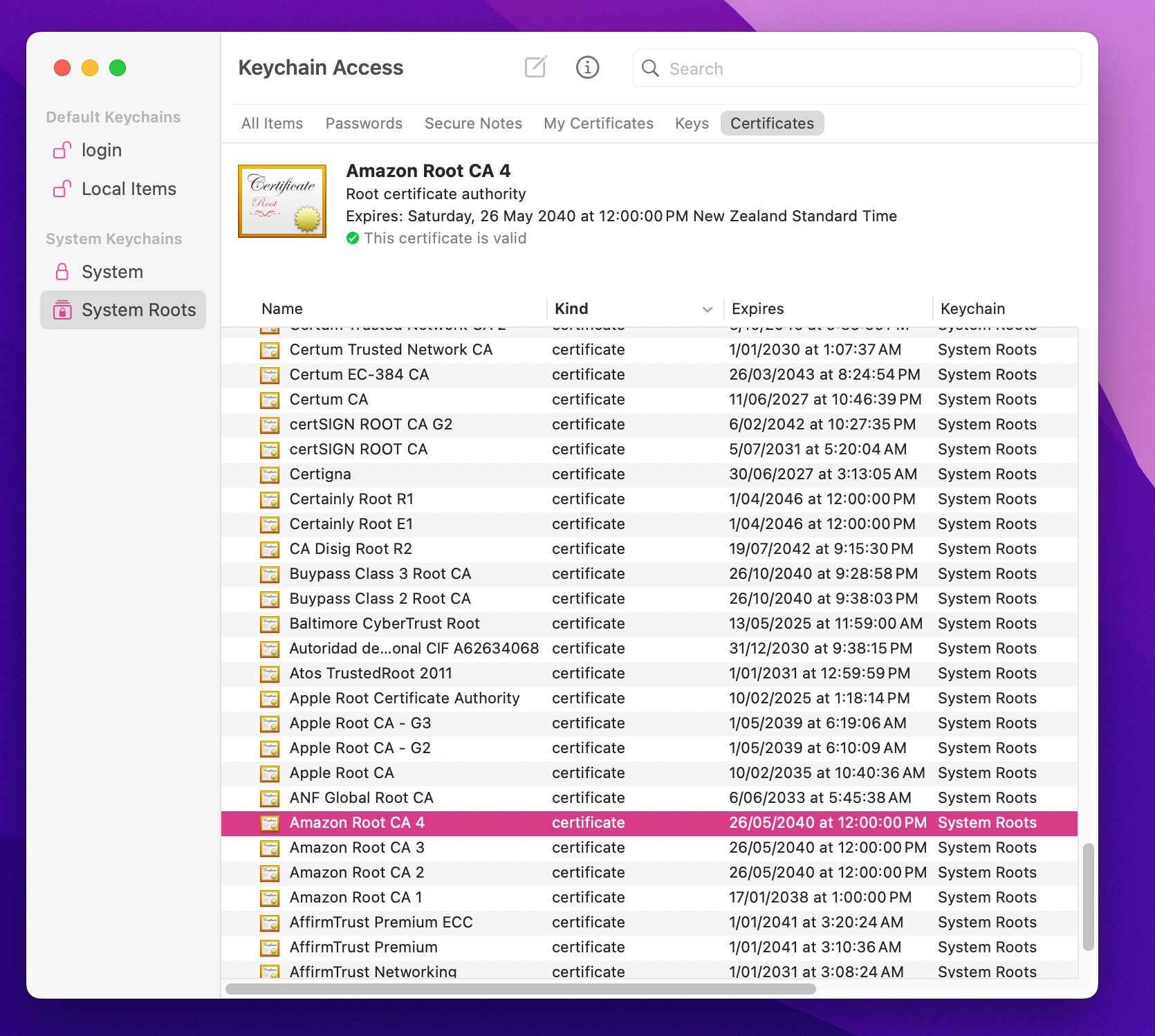
1. Software installs a root CA certificate into your device as part of its installation
2. Your device initiates a connection to a secure website, say, "https://www.example.com."
3. The ISP/VPN/Anti-Virus/firewall/online server or router/[entity] intercepts this connection then responds with a fake certificate
4. Your device receives the fake certificate, and since it's signed by the trusted root CA (which your device trusts), it appears valid to your browser
5. Your device establishes a secure connection with the [entity], thinking it's connecting to "www.example.com."
6. The [entity], forwards your connection request to the real "www.example.com" server.
7. The [entity] decrypts and inspects the traffic before re-encrypting it and forwarding it to the real server.

This process essentially allows the *entity* to perform a **MITM** attack on the HTTPS connection without alerting both parties. It relies on the fact that your device trusts the entity's root CA certificate, making the interception seamless from the user's perspective. To detect this, user needs to compare the public key of the connection with the actual SSL/TLS public key of the website. This is beyond what most user is capable of. Although ‘SSL certificate pinning’ of modern browsers may help mitigate this issue, it’s not fool proof and has its own disadvantages. What if the HTTPS connection is established without a browser? Heck, for all we know, Three Letter agency and/or big tech are capable of stripping away HTTPS to log plaintext communication without using certificates at all.

Strictly speaking, the entity doesn’t even need to install any malicious Root CA certificate in your device since each device is already pre-installed with a set of such certificates, see, if the entity is the government or the big **Three Letter agency**, then all it needs to do is persuades, compels or slaps a national security order (NSO) to those top CA companies that happen to operate within the country of such government to cut them a new certificate signed by the private key of Root CA. This new legit certificate would then be trusted by major web browsers and operating systems. Since money rules in this world, it’s in the best interests of these CA companies to comply and keep quiet if they want to keep making millions in revenues. If the entity is Big Three Letter agency, then it’s a safe bet the countries of [Five Eyes](https://en.wikipedia.org/wiki/Five_Eyes) are all in on this.

**Innocent trusted Root CA certificates**

Let’s do a little reality check to see how many trusted root CA certificates exit in my *MacOS Sonoma 14.0* right now. Simply open ‘Keychain Access’ app => System Roots => Certificates.



Holy moly, there is a total of 159 System Roots certificates in my Mac right now. Just to name a few: *Symantec, Cisco, Visa, Microsoft, Amazon, Apple, Go Daddy, DigiCert, SwissSign, GlobalSign, Certum, SSL.com*, ...

Looks like every big tech and shadowy organisations (i.e. front stores for other big tech or govts) come out of the woodwork when we take a closer look.

Does a computer really need that many pre-installed trusted Root CA certificates?

Do we fully understand what each of these root CA certificate is doing or used for? No.

Do we trust it not to do anything unethical? No.

I will leave the reader to look up pre-installed root CA certificates in Windows OS.

**The NEED for Message Level Security (MLS)**

The aforementioned *soap opera* could have been avoided if we simply used E2E encryption between client and server where actual encryption and decryption happens on your device in addition to HTTPS. Decryption of received data happens in your device. Encryption of data to send happens in your device before being sent out to server by HTTPS. Likewise on the server side. If a bad actor somehow strips away HTTPS to reveal unencrypted data, all they will ever see is our encrypted data. This is basically *encryption within encryption*, Inception in the IT world.

Is this overkill? Totally, if one doesn’t care about online privacy but NO for any financial transaction APIs and definitely NO for any self-respecting IT professional. 👌

As of now (2023), one of the most well-known, highly rated and widely respected open-source, end-to-end (E2E) encryption protocol by security experts seems to be [Signal Protocol](https://signal.org/). It’s used by Signal app, a popular message sharing app for privacy-conscious users, including journalists, activists. This is also the app publicly endorsed by Edward Snowden in the past.

Hold on a sec, I hear you say, if Signal protocol is open-source is it possible to use it in Java API?

Yes, it is possible and it has been done already, see <https://github.com/signalapp/libsignal-protocol-java>. So why don’t I use it? Well, I can’t find user friendly documentation that explains how the protocol actually works and I am not keen to dig into its huge codebase to learn the protocol. Examining code to learn how Protocol works without documentation is not good for mental health unless that’s your thing. So it’s more fun to do things the hard way by researching and coming up with one myself. This may be better off in the long run than just calling a black box API and have no idea of what’s going on behind the scene. But if one has a crazy deadline to deliver E2E encryption, then by all means, use it.



[**The A-Team**](https://www.youtube.com/watch?v=wyz_2DEah4o)

In this section we are going to introduce and examine the three chosen cryptographic algorithms, its history, creators, commercial use and popularity. Why? Because a security standard is only as strong as its weakest link. Plus the last thing we want is pick a weak, compromised, slow or insecure algorithm. Not to mention the Three Letter agency has some of the world’s best cryptographers helping to set the standards endorsed by the NIST for the whole world to follow.

So the job of picking the right algorithms really can’t be taken lightly else we might as well stick with HTTPS. Needless to say, there is no guarantee the algorithms I pick aren’t compromised. They were chosen to the best effort of my research by using ChatGPT, Google Bard, classic googling to check technical articles and forums.

Speaking of research, credits and thanks have to be given to ChatGPT and Google Bard for explaining, showing and even generated many of the Node.js code for feasibility study as we will see later. This is actually more productive than simply googling and reading articles all over the web and forums to try to piece things together. But be warned the answers returned by these AI bots are not always correct or even make sense, common sense and cross checking with google search required.

Let’s set some criteria for choosing the algorithms. It needs to be open-sourced, readily available in JavaScript or Node.js library for client side. Server side can use whatever language later on. It needs to be used in many popular commercial applications. It need to have good community support and easy to use. The last thing we want is to hand roll or implement these algorithms ourselves.

[**Curve25519**](https://en.wikipedia.org/wiki/Curve25519)

* Designed by [Daniel J. Bernstein](https://en.wikipedia.org/wiki/Daniel_J._Bernstein) and published in 2006/2005
* A PKI key exchange algorithm, an elliptic curve cryptography (ECC) curve that is used to **generate public and private keys and exchange public keys** for establishing secure connection. It is **one of the most secure and fastest curves in ECC**, and used in a variety of applications, including:
  + Virtual private networks (VPNs): Curve25519 is used to implement secure VPN connections in apps such as OpenVPN and WireGuard
  + Signal Messenger: The Signal Protocol, the cryptographic protocol behind the Signal messaging app, uses Curve25519 for key exchange and digital signatures. It plays a crucial role in establishing secure communication channels.
  + Tor Browser: Curve25519 is used in the Tor network for key exchange in the Tor Browser. It helps ensure secure and private communication over the Tor network.
  + SSH (Secure Shell): Some implementations of the SSH protocol use Curve25519 for key exchange. It provides a secure method for establishing encrypted connections for remote access.
  + LibreSSL and OpenBSD: Curve25519 is supported in LibreSSL, a fork of OpenSSL, and is the default key exchange algorithm in OpenBSD's SSL/TLS implementation.
  + Bitcoin and Cryptocurrency: Curve25519 is used in some cryptocurrency-related applications, including certain aspects of Bitcoin cryptography.
  + Secure File Transfer Applications: Some secure file transfer applications and protocols use Curve25519 for key exchange and authentication to ensure secure and private file transfers.
* This algorithm will be chosen to compute the actual symmetric encryption key used in ChaCha20 without the need to transport actual encryption key to the other side, i.e. both sides can compute the same encryption key by using its own Curve25519 private key and the Curve25519 public key of the other party.
* Risk level: little

[**Ed25519**](https://ed25519.cr.yp.to/)

* Designed by Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. Published in 2011.
* A PKI digital signature algorithm that uses elliptic curve cryptography. It is one of the most secure and efficient digital signature algorithms available
* Designed as a more secure and efficient alternative to ECDSA
* Applications using Ed25519
  + OpenSSH: Ed25519 is the default authentication algorithm for OpenSSH, a secure shell (SSH) client and server for remote login and file transfers.
  + Git: Ed25519 is the default signing algorithm for Git, a distributed version control system for tracking changes in source code during software development.
  + HTTPS certificates: Ed25519 is supported by many certificate authorities (CAs) for signing HTTPS certificates. This helps to ensure that users are connecting to the correct website and that their data is encrypted in transit.
  + Software distribution: Ed25519 is used by many software companies to sign their software distributions. This helps to ensure that the software has not been tampered with before it is installed.
  + Code signing: Ed25519 is used by many software developers to sign their code. This helps to verify the authenticity of the code and protect it from tampering.
  + Email signatures: Ed25519 is used by many people to sign their emails. This helps to verify the authenticity of the sender and protect the email from tampering.
  + Messaging apps: Ed25519 is used by many messaging apps, such as Signal and WireGuard to sign user’s identity/public key
* This algorithm will be chosen to create digital signature from payload before sending to the other side so the receiver can confirm identity of sender.
* Risk level: little

[**ChaCha20-Poly1305**](https://blog.cloudflare.com/it-takes-two-to-chacha-poly/)

* ChaCha20-Poly1305 is a cryptographic function that combines both ChaCha20 encryption algorithm (data confidentiality) and the Poly1305 authentication algorithm (data integrity)
* Creators:
  + ChaCha20 was designed by Daniel J. Bernstein, a well-known cryptographer and computer scientist. Released in 2008.
  + Poly1305 was designed by Daniel J. Bernstein along with Tanja Lange. Released in 2005.
* **ChaCha20** is a symmetric encryption algorithm. It is a stream cipher designed for symmetric key cryptography, meaning the same secret key is used for both encryption and decryption
* **Poly1305** is a hash function used as a one-time message authentication code (MAC). A MAC is a cryptographic primitive that can be used to verify the authenticity of a message. It is typically used to protect messages from tampering or forgery (i.e. provides data integrity)
* Use Cases:
  + TLS Cipher Suites: ChaCha20-Poly1305 is commonly used as a cipher suite in the Transport Layer Security (TLS) protocol. It provides a secure and efficient alternative to traditional block ciphers like AES.
  + VPNs (Virtual Private Networks): ChaCha20-Poly1305 is utilized in VPN protocols as a secure encryption and authentication mechanism. It is often favoured for its speed and resistance to certain types of attacks.
  + SSH (Secure Shell): Some implementations of the SSH protocol use ChaCha20-Poly1305 for encryption and authentication.
  + Secure Communication Protocols: It is used in various secure communication protocols and applications where a combination of encryption and authentication is required
* This algorithm will be chosen to do the actual E2E encryption of communication between the two parties.
* Risk level: little

**EazyPGP standard/protocol for API**

* ‘Shared secret key’ below will refer to the shared key computed from Curve25519 between Client and Server
  + Curve25519( {Public key of Server}, {Private key of Client} ) =

Curve25519( {Public key of Client}, {Private key of Server} ) = Shared secret key

1. Key Generation:

* Ed25519
* Each party, client and server, generates its long-term Ed25519 key pair
* These long-term key pairs (public key and private key) are securely stored by each party
* Public key of client will be sent to server in advance and vice versa but never private key. Note there is no security risk of public key being intercepted or made public but it’s still good practice to share public key on need to know basis to minimize risk.
* Curve25519
* Each party, client and server, generates its long-term Curve25519 key pair
* These long-term key pairs (public key and private key) are securely stored by each party
* Public key of client will be sent to server in advance and vice versa but never private key. Note there is no security risk of public key being intercepted or made public but it’s still good practice to share public key on need to know basis to minimize risk.

1. Client challenges Server by making a Client challenge request (simulating SSL handshake):

* Client initiates communication to indicate its intention to establish a secure channel by sending a request to a Server API say ***http://{{server URL}}/challenge***
  + HTTP request headers

Content-Type: application/json

sig: {{signature}}

cid: eazy-pgp

* cid stands for Client ID.
* {{signature}} is signature of plaintext payload before ChaCha20 encryption by using Ed25519 private key of client
  + HTTP request body (all {..} data below is Base64 encoded string, e.g. {mac})

*{*

*"data": {encrypted-data}${nonce}${mac}*

*}*

* + - {encrypted-data} – ChaCha20 encrypted data by using shared secret key
    - {nonce} – Randomly generated 96 bits
    - {mac} – message authentication code(MAC) of data before ChaCha20 encryption
    - Request body example

{

“data”: “AZT5Eb3z3soazi7s6Nd+V/jH1XDE84R+N0hxqKOUqKi3vdpv9ho8dPGb0Em12idVU2IYwDuCbAaxzPvkUCwuzvG1a77LEahV1RhYtiiAhPrIFGudEQ0=$WDRDHTvhfjhMnwkK**$**D1QNpjmvFYpfrloYyVAksw==”

}

* + Actual plaintext payload before ChaCha20 encryption has format
    - {

"client\_challenge": {Base64 encoded string of randomly generated 16 Bytes}

"timestamp": {timestamp}

}

* + - Example

{

"client\_challenge": "vqeZ679dI11Ns3xRD5Wp9A==",

"timestamp": "2023-10-29T02:20:35.637Z"

}

1. Server receives Client’s challenge request
   * If HTTP header sig doesn’t match an existing Client Id, return error response then exit
   * If challenge request data format doesn’t match

*{*

*"data": {encrypted-data}${nonce}${mac}*

*}*

return error response then exit

* + Validate identity of client by validating challenge request's signature from the value of HTTP header ‘sig’ by using Client's Ed25519 public. Return error response then exit if signature validation fails.
  + Decrypt *{encrypted-data}* above with ChaCha20 by using the shared secret key then verify the decrypted data hasn’t been tampered with by checking {mac} from request challenge with Poly1305. If anything fails (i.e. ChaCha20 or Poly1305), return error response then exit.
  + Optional: If timestamp of decrypted client challenge is more than 10 seconds, reject request, return error response then exit. This is defending against replay attack.
  + Send response back to client’s challenge request
    - Response body has encrypted data format below by ChaCha20-Poly1305 using the shared secret key

*{*

*"data": {encrypted-data}${nonce}${mac}*

*}*

* + - * Actual plaintext response body of {encrypted-data} has format

{

“timestamp”: “2023-10-29T02:20:35.865Z”,

“client\_challenge”: “vqeZ679dI11Ns3xRD5Wp9A==”,

“sessionId”: “j+CRZTCflYSMOtOJxLRO+Q==”

}

* + - * Value of client\_challenge is copied from decrypted challenge request. Client can check this to avoid potential attack if it doesn’t match
      * Value of **sessionId** is Base64 encoded string of randomly generated 16 bytes.
      * Server will save a record of this *key-value* pair in its cache with key={sessionId} and value={client\_challenge}. This record will expire and gets removed from the cache after 15 minutes. Why? To force the client to start a new session after 15 minutes
      * Response HTTP header

Content-Type: application/json

sig: {signature}

* {signature} is computed from plaintext response body above by Ed25519 private key

1. Client validating Server’s response to its Client request response
   * Validating server response data format, if it’s not the usual format below, display error message and stop

*{*

*"data": {encrypted-data}${nonce}${mac}*

*}*

* Validate server’s signature from HTTP response header ‘sig’ against its Ed25519 public key, if validation fails, displays error message and stop
* Decrypt *{encrypted-data}* above with ChaCha20 by using the shared secret key then verify the decrypted data hasn’t been tampered with by checking {mac} from response body with Poly1305. If anything fails (i.e. ChaCha20 or Poly1305), display error message then stop.
* Optional: If timestamp of decrypted response data is more than 10 seconds, reject response, displays error message then exit.
* Validate client\_challenge from decrypted response matches original client\_challenge sent to server in Client challenge request. If different, display errors then stop.

OK, by now if everything goes well, both client and server have confirmed each other’s identity, it’s time to exchange ephemeral (i.e. temporary) Curve25519 public keys to each other. You may ask ‘Why not keep using the long-term Curve25519 public key?”. Umm bad idea, we want to maintain perfect forward secrecy (PFS) so even if the private key of Curve25519 is compromised it cannot be used to generate key to decrypt past or future communications recorded outside that session.

So the long-term Curve25519 public keys will only make its appearance during mutual authentication and exchange of ephemeral Curve25519 public key between client and server. Thinking forward, any client device (i.e. browser extension, mobile or desktop app) wanting to use EazyPGP would need to have long-term Curve25519 and Ed25519 keys registered on the device. To top it off, 2FA, Micrsoft/Google authenticator could be introduced to enable the use of long-term Curve25519/Ed25519 keys on device so even if those keys are compromised, it’s still no showstopper.

1. Client can now make a request to server’s ‘Exchange Key’ API say ***http://{{server URL}}/exchange-key*** to exchange ephemeral Curve25519 public key with the server.
   * Generates a new ephemeral Curve25519 key pair
     + As usual, the client will have request body of format below using shared secret key and ChaCha20-Poly1305

*{*

*"data": {encrypted-data}${nonce}${mac}*

*}*

* + - Plaintext request body of *{encrypted-data}* has format

{

“timestamp”: “2023-10-29T02:20:35.911Z”,

“sessionId”: “j+CRZTCflYSMOtOJxLRO+Q==”,

“public\_key”: “XgAgI1lUEm+HYkzjhz8u7pgA7EDWv4ST0el6mEoEGV8=”

}

* + - * public\_key: the ephemeral Curve25519 public key. You may ask ‘Hold on, why encrypt public key, isn’t it safe to transport it as plaintext in HTTP header? Well, it’s more fun this way for whoever wants to take a shot at breaking it.
      * The sessionId will tie the request to server cache’s key-value record step 3 above. This request should fail if server doesn’t have this record in its cache.
    - Request HTTP header

Content-Type: application/json

sig: {signature}

* {signature} is computed from plaintext request body above by Ed25519 private key

1. Server receives client’s request to exchange temporary public Curve25519 key. As usual, validate request body data format, decrypt {encrypted-data}, validate timestamp then validate HTTP header signature {sig}. Return error message and stop if any fails.

Look up request’s sessionId from the server cache (step 3) to see if such record exist, if not, return error and stop. If yes, generate a new ephemeral Curve25519 key pair, save the new key pair and the new ephemeral public key from client to the cache. All these will be retrievable from server cache by sessionId. Now return server’s new Curve25519 public key to the client in exchange key response as

*{*

*"data": {encrypted-data}${nonce}${mac}*

*}*

* + - Plaintext response body of *{encrypted-data}* has format

{

“timestamp”: “2023-10-29T02:20:35.911Z”,

“sessionId”: “j+CRZTCflYSMOtOJxLRO+Q==”,

“public\_key”: “XgAgI1lUEm+HYkzjhz8u7pgA7EDWv4ST0el6mEoEGV8=”

}

* + - * public\_key: the ephemeral public key of Curve25519 from server.
    - As usual for HTTP response header, add signature of plaintext response by server’s Ed25519

Content-Type: application/json

sig: {signature}

1. Secure channel is now established between client and server as all communications to/from the server’s API at **http://{{server URL}}/{URI}** will be encrypted using the newly shared secret key, derived from the exchanged Curve25519 public key.

Both client’s request and server’s response will still use the same encrypted data format below. However {*encrypted-data}* can now be anything.

*{*

*"data": {encrypted-data}${nonce}${mac}*

*}*

HTTP request header now has a mandatory sessionId to tell server which shared secret key to use for encryption and decryption. HTTP header sig is now redundant/optional e.g.

Content-Type: application/json

sessionId: {sessionId}

Replay attack can be mitigated by

* Enforcing a timestamp field in decrypted data so server could reject the request if it’s more than. 5~10 seconds from now.
* Server keeping track of a list of unique {nonce} from requests having the same sessionId within a session. Before server responds to any request, check to see if the nonce from request exists already for the given sessionId, if yes, replay attack is detected. A cap limit can also be put on the number of nonce to track within a session for each sessionId entry. If limit is reached, remove sessionId entry from server cache then asks client to renew session (i.e. client challenge request and exchange public keys as before) again in error response, e.g. 300 requests within 15 minutes. This would also be a good way to enforce rate limiting of an API per Client Id basis.
* Remove any sessionId entry from server cache after 15 minutes

Congratulation if you made it this far. If you have really read through the dry Protocol section and is hungry for more, what is wrong with you?

**Thinking Exercises..**

Modify the algorithm so each request and response cycle will use a new symmetric key after key exchange while retaining the use of ephemeral Curve25519 public key within a session, as opposed to using the same symmetric key throughout the session.

Security concerns or issues with this protocol?

What’s the main challenge of using a Protocol like this between “browser, desktop or mobile app” and a server?

* Is it feasible for both client and server to pre-register public keys in decentralized Web3 and blockchain for each other to access at runtime to establish secure channel?
* A dedicated centralized online service registry of public keys for any two registered parties to access and to establish secure channel at runtime?
  + Off the shelf solutions for this?

Is it even feasible for browsers to implement a Protocol like these with a webserver?

Encoding video and audio files with ChaCha20?

* Browser extensions accessing Web3, DApp and Blockchain to load public keys of server at runtime to establish secure channel?
* What does Signal Protocol have to say about encoding video and audio files?

How about an open-source public key registry app across major OS that sync with Web3/Blockchain daily to load latest public keys of registered websites/servers so applications on your device can simply use it to establish secure channel with any registered websites/servers on earth? Same idea of pre-loaded root CA certificates on each device on page 8. Private keys need to be stored in your local devices with a secure open-source app in this case.

**Feasibility Study**

Time to see how easy or difficult it is to use Curve25519, Ed25519 and ChaCha20-Poly1305 in Node.js. Fortunately, it turns out to be not difficult at all. And NO, I won’t provide EazyPGP implementation in this project, maybe next time. My current prototype is half baked, full of bugs and lacks testcases. Feel free share it with me if you beat me to it.

**Curve25519**

// @authors: ChatGPT, Samuel Huang

// Created: June-2023

const fs = require('fs');

const nacl = require('tweetnacl');

// Generate Curve25519 key pair for the client

const clientKeyPair = nacl.box.keyPair();

const clientPublicKey = Buffer.from(clientKeyPair.publicKey).toString('base64');

const clientPrivateKey = Buffer.from(clientKeyPair.secretKey).toString('base64');

// Write client public key to PEM file

fs.writeFileSync('client\_public.pem', clientPublicKey);

// Write client private key to PEM file

fs.writeFileSync('client\_private.pem', clientPrivateKey);

console.log('Client Public Key:', clientPublicKey);

console.log('Client Private Key:', clientPrivateKey);

// Generate Curve25519 key pair for the server

const serverKeyPair = nacl.box.keyPair();

const serverPublicKey = Buffer.from(serverKeyPair.publicKey).toString('base64');

const serverPrivateKey = Buffer.from(serverKeyPair.secretKey).toString('base64');

// Write server public key to PEM file

fs.writeFileSync('server\_public.pem', serverPublicKey);

// Write server private key to PEM file

fs.writeFileSync('server\_private.pem', serverPrivateKey);

console.log('Server Public Key:', serverPublicKey);

console.log('Server Private Key:', serverPrivateKey);

// Compute the shared secret key between client and server

const clientSharedSecret = nacl.scalarMult(clientKeyPair.secretKey, serverKeyPair.publicKey);

const serverSharedSecret = nacl.scalarMult(serverKeyPair.secretKey, clientKeyPair.publicKey);

const clientSharedSecretHex = Buffer.from(clientSharedSecret).toString('base64');

const serverSharedSecretHex = Buffer.from(serverSharedSecret).toString('base64');

console.log('Shared Secret (Client):', clientSharedSecretHex);

console.log('Shared Secret (Server):', serverSharedSecretHex);

// Data to encrypt

const data = 'This is a secret message';

// Convert data to Uint8Array

const dataBytes = Buffer.from(data, 'utf8');

// Nonce generation (24 bytes)

const nonce = nacl.randomBytes(nacl.box.nonceLength);

// Encryption

const encryptedData = nacl.box(dataBytes, nonce, Buffer.from(serverPublicKey, 'base64'), Buffer.from(clientPrivateKey, 'base64'));

const encryptedMessage = encryptedData ? Buffer.from(encryptedData).toString('base64') : null;

// Simulate receiver receiving base64 encoded text, then convert it to binary form for decryption

const encryptedData2 = Buffer.from(encryptedMessage, 'base64');

// Decryption

const decryptedData = nacl.box.open(encryptedData2, nonce, Buffer.from(clientPublicKey, 'base64'), Buffer.from(serverPrivateKey, 'base64'));

const decryptedMessage = decryptedData ? Buffer.from(decryptedData).toString('utf8') : null;

console.log('Original message:', data);

console.log('Encrypted message:', encryptedMessage);

console.log('Decrypted message:', decryptedMessage);

// Console output of running code

Client Public Key: leCVh3L5d8531l433n+/rrk0cmXwfQS40FjZl2zwxmc=

Client Private Key: J5PDVyjcdX/LNy/MEQKKm8jmJlOe240v7wYGzuGSZtI=

Server Public Key: Yf6PDTJaB2D8yWOGPn48FQ7YIkUlaovKrvR0q2P1QWw=

Server Private Key: Bg/ZDOcPCalyKVZErWo4OBxOf9U2Eoi/UxgMQmxpne0=

Shared Secret (Client): NDxExYpXXLvmu2+lmbyyE6dHWzqf+vuvgoaOwtKLoX4=

Shared Secret (Server): NDxExYpXXLvmu2+lmbyyE6dHWzqf+vuvgoaOwtKLoX4=

Original message: This is a secret message

Encrypted message: 9/DjkLFaduYsEDlJqUTfhsQJjdOrEcqUWu2LOJ+HRvzco1twlC6f9w==

Decrypted message: This is a secret message

**Edh25519**

// @authors: ChatGPT, Google Bard, Samuel Huang

const fs = require('fs');

const nacl = require('tweetnacl');

const util = require("tweetnacl-util") // encoding & decoding

// Load the private key PEM file

const privateKey = fs.readFileSync('private\_key.pem', 'utf8').trim();;

console.log('privateKey: ' + privateKey)

// Convert the private key to a Uint8Array object

const privateKeyBuffer = Buffer.from(privateKey, 'base64');

console.log('privateKey length: ' + privateKeyBuffer.length )

const privateKeyUint8 = Uint8Array.from( privateKeyBuffer );

const signer = nacl.sign.keyPair.fromSecretKey( privateKeyUint8 );

const data = 'This is some data to sign.';

const messageBytes = util.decodeUTF8( data );

const signature = nacl.sign.detached(messageBytes, signer.secretKey);

console.log('signature: ' + signature);

// Load the public key PEM file

const publicKey = fs.readFileSync('public\_key.pem', 'utf8').trim();

console.log('publicKey:', publicKey);

// Convert the public key to a Uint8Array object

const publicKeyBuffer = Buffer.from(publicKey, 'base64');

console.log('publicKey length:', publicKeyBuffer.length);

const publicKeyUint8 = Uint8Array.from( publicKeyBuffer );

// Verify the signature using public key from a PEM file

const data2 = 'This is some data to sign.';

const messageBytes2 = util.decodeUTF8( data2 );

const isVerified = nacl.sign.detached.verify(messageBytes2, signature, publicKeyUint8);

// Verify the signature

// const isVerified = nacl.sign.detached.verify( messageBytes, signature, signer.publicKey );

// Check if the signature is verified

if ( isVerified ) {

console.log('Signature verified!');

} else {

console.log('Signature not verified!');

}

*// Console output of running code*

privateKey: 2DlCDUXzCHlhOywYHlcbgyGr4D7lTaiXGfTYVvnQ6GrIP/aUAbGLGm8O0aageOE3

hcOjgmUVupOiIuZzTzFpPA==

privateKey length: 64

signature: 103,155,122,34,172,210,208,132,220,121,88,107,207,196,214,139,235,246,53,176,174,199,99,228,15,45,233,28,61,32,35,255,105,71,140,155,48,203,199,67,231,97,72,213,151,207,220,254,176,111,147,181,107,181,103,116,50,148,157,193,238,181,196,8

publicKey: yD/2lAGxixpvDtGmoHjhN4XDo4JlFbqToiLmc08xaTw=

publicKey length: 32

Signature verified!

**ChaCha20-Poly1305**

// @authors: CodePal, ChatGPT, Samuel Huang

const crypto = require('crypto');

function encryptText(text, key) {

const nonce = crypto.randomBytes(12);

// Create a cipher object using the key, nonce, and authTagLength

const cipher = crypto.createCipheriv('chacha20-poly1305', key, nonce, {

authTagLength: 16

});

let encrypted = cipher.update(text, 'utf8', 'base64');

encrypted += cipher.final('base64');

const tag = cipher.getAuthTag();

return { encryptedText: encrypted, nonce: nonce, tag: tag };

}

function decryptText(encryptedText, key, nonce, tag) {

const decipher = crypto.createDecipheriv('chacha20-poly1305', key, nonce, {

authTagLength: 16

});

decipher.setAuthTag(tag);

let decrypted = decipher.update(encryptedText, 'base64', 'utf8');

decrypted += decipher.final('utf8');

return decrypted;

}

// Generate a secret key

const key = crypto.randomBytes(32);

// Encrypt some text

const text = 'Hello, world!';

const encryptedData = encryptText(text, key);

// Uncomment this line then run code to verify MAC authentication is working in decryption

// encryptedData.tag = crypto.randomBytes( 16 );

console.log('Encrypted text: ', encryptedData.encryptedText);

console.log('Nonce: ', encryptedData.nonce.toString('base64'));

console.log('MAC: ', encryptedData.tag.toString('base64'));

// Decrypt the text

const decryptedText = decryptText(

encryptedData.encryptedText,

key,

encryptedData.nonce,

encryptedData.tag

);

console.log('Decrypted text: ', decryptedText);

// Actual console output

Encrypted text: CANlQb+ZaB1ytImOgw==

Nonce: nNFB59n7MKpr4Pzn

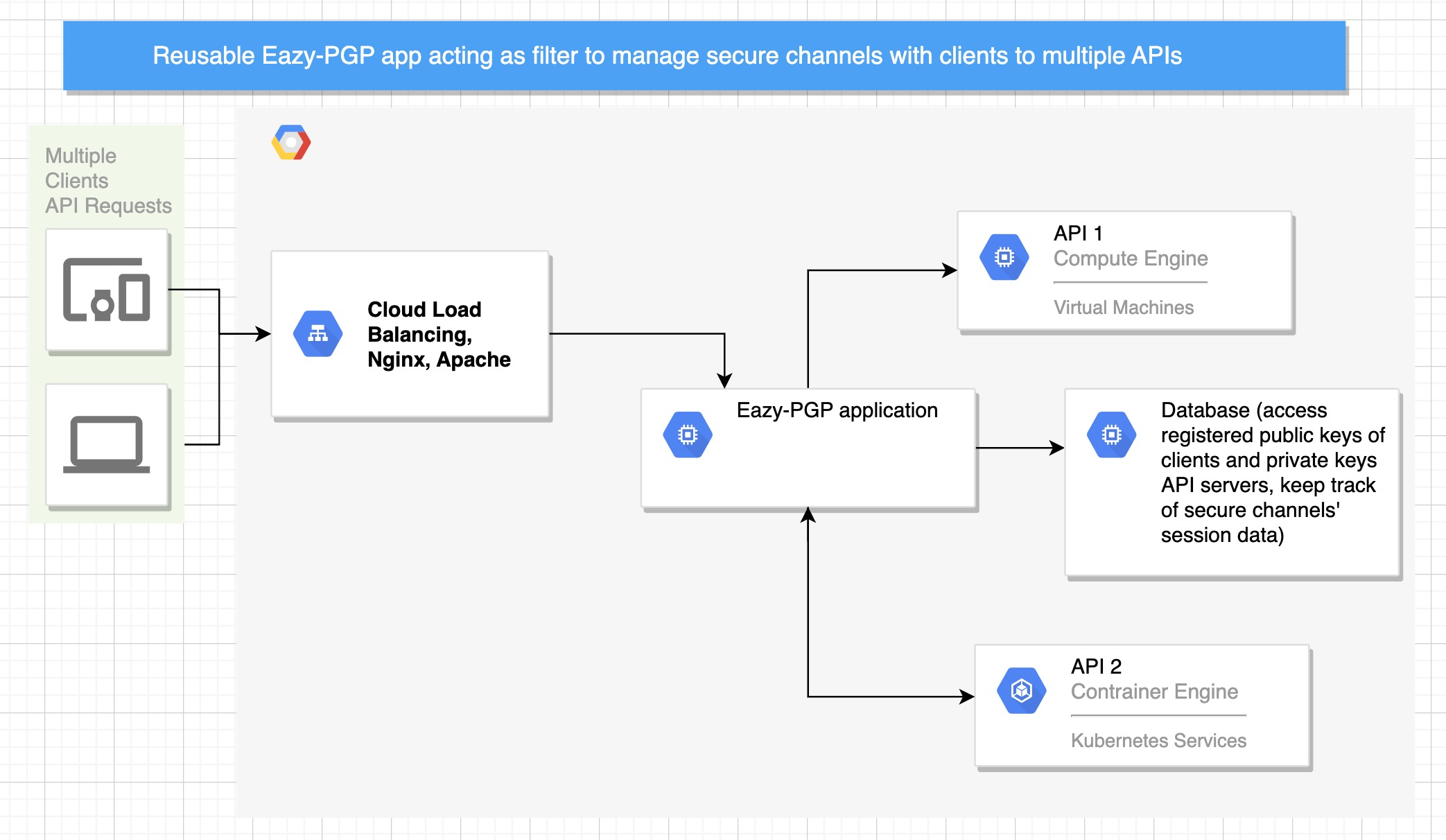
MAC: ia01UpCPdBMQcpTy7TtzDg==

Decrypted text: Hello, world!

That’s it. Easy Peasy. See my Github code for more details.

**What Now?**

Assuming this protocol is implemented within an API using Node.js or Spring Boot, what happens if a new API wants to use it? Redo it again by copy-and-paste? This is a big No-No and we need a reusable solution, a filter that manages and establishes secure channels with clients for multiple APIs. Communications between Eazy-PGP app and APIs will be in plaintext.



Hey maybe we could have an online, subscription based, off-the-shelf E2E encryption application/server that can be deployed to AWS, GDP, Azure, IBM, .. at the click of a few buttons. Client-Side SDK can then be provided for major languages (e.g. Java, C#, Node.js, Kotlin, React Native, Flutter, ...) in Github for developers to use and play around with Eazy-PGP application. Not to mention opportunities for consulting and commercial support. There you go,

one business idea, you are welcome.



Hopefully you have learned something interesting and useful. Till next time.. 👋