HighSide Technical Paper

Authored & Maintained by Jonathan Warren, CTO @ HighSide, Inc.

HighSide is the name of our company and also the name of a stand-alone desktop and mobile client. The client encrypts messages so that only people participating in a group can read them. Here we discuss how that works.

This is a technical overview of how HighSide works for those who are interested in specifics. You do not need to read or understand any of this to use HighSide correctly unless you are the admin for your company in which case you might choose to just read the section on authentication.

Each message is encrypted then HMAC'd with a randomly generated unique 256 bit key using AES-CTR, hereafter the "AESEphemKey". Then, for every participant who must receive the message, the AESEphemKey is encrypted with the participant's 512 bit secp256k1 elliptic curve public key and that encrypted AESEphemKey is added to the top of the ciphertext as a header. The whole thing is signed using ECDSA and SHA256, and then the information is sent to the server and relayed to the receiving clients. Each receiving client checks the signature, finds their header, decrypts the AESEphemKey using their private key, and then decrypts the main ciphertext using this AESEphemKey. Sending files works similarly except that files are split, compressed, encrypted, and sent in pieces to speed things up. Key

authentication is taken care of by an admin at your company; if users trust the admin then they do not need to all verify each other's keys.

Handy list of things that HighSide (the company) can see:

- The IP of connected HighSide clients
- The time that you connect and disconnect from the server
- The sender and receiver of messages
- The approximate size of messages or files
- The names of groups

List of things that HighSide (the company) cannot see:

- The content of messages
- The content of files
- The names of files

And when we say that we "cannot see" these things, we do not mean that employees are forbidden from seeing these things; we mean that we have specifically engineered the system such that no honest HighSide employees, malicious HighSide employees, or hackers who hack our server will be able to see the content of your messages or files. At no point does unencrypted message data pass through or exist on our servers either on disk or in memory. This is fundamentally different from other services who protect messages with SSL: in that case the data is necessarily decrypted at the server. With HighSide clients, SSL isn't necessary at all; we debated whether to even bother with it.

Contents

Creating keys

To generate your encryption keys and your address, this process is used. The first half is almost exactly what is done with modern Bitcoin wallets; the second half is what is done in Bitmessage.

1. Using Python's os.urandom module, 132 bits of random data is generated
2. These 132 bits are used to select 14 words from the word list
3. These 14 words are saved as the "seed". This seed can be viewed in the HighSide UI.
4. PBKDF2 is used to "stretch" the seed. It uses a hard-coded salt, 4096 iterations, and SHA512.

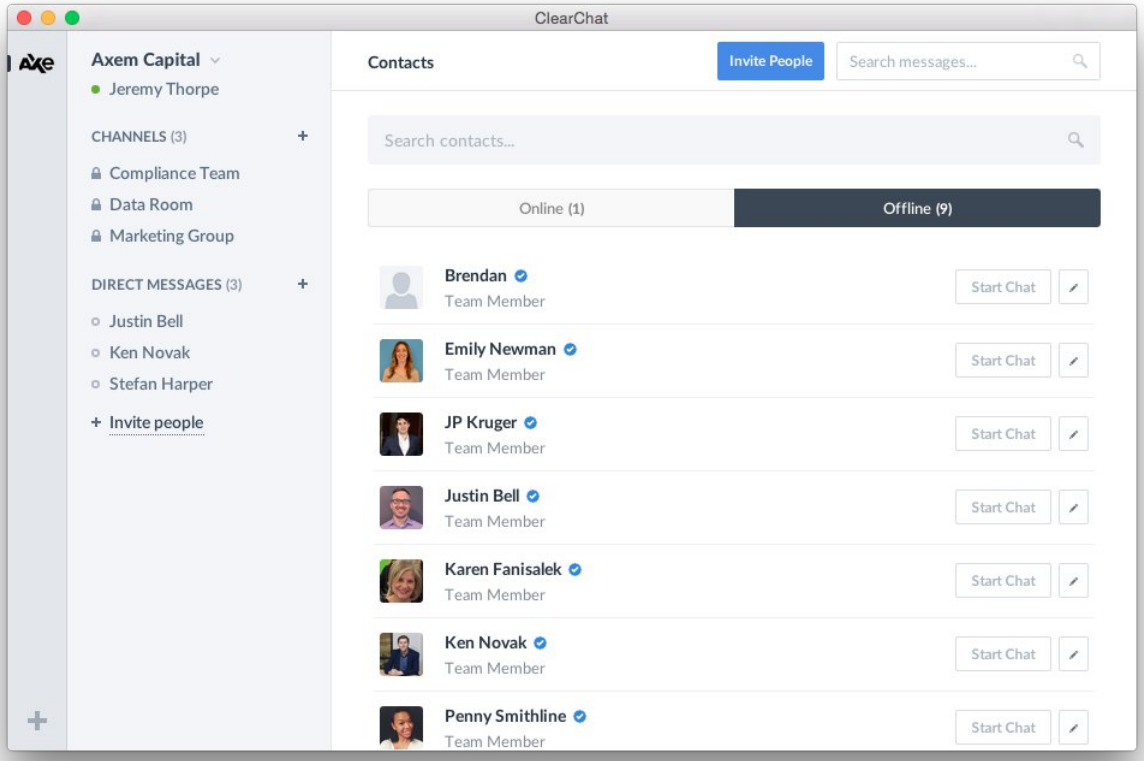
5. This seed is then appended with a nonce so that we can, in the future, generate more addresses from this same seed.
6. The seed and nonce are then hashed twice with SHA512.
7. The first 32 bytes of this hash constitutes our ECC private signing key.
8. The nonce is then incremented and we double-hash again. This hash constitutes our ECC private encryption key.
9. A pair of ECC point multiplications on the secp256k1 curve are done to turn the private keys into public keys.
10. The public signing key and the public encryption key are together hashed using SHA512 and then RIPEMD160.
11. A version number (currently 1) is prepended to the front of the RIPE data and a checksum is appended to the end. This is encoded using base58, and “CH-” is prepended onto the front to differentiate it from Bitmessage and Bitcoin addresses. This is your address.
12. Your keys are encoded in Wallet-Import-Format and saved in the keys.dat file.

The peanut gallery might wonder why we use secp256k1 instead of Curve25519:
1) Curve25519 is not supported by pyelliptic and we do not roll our own crypto
2) secp256k1’s faults on the SafeCurves website do not concern us nor does the origin of the base point G
3) the Bitcoin project is a multi-billion dollar incentive to research, discover, and exploit any vulnerabilities in ECC and secp256k1. Such exploitation would make the vulnerabilities public.

Creating the company

An admin, Alice, will create an account on the website for herself by inputting her name and email address. Alice is the only person at her company who ever needs to make an account on the website. HighSide's server will generate a 20 byte token ("signup token") and email her the token. Admin Alice downloads the HighSide client and copies the signup token into it. The client generates some new encryption keys and will try to connect to the server. When it does, it delivers its public keys and the signup token. The server looks up which company is associated with that signup token, sees that there are not currently any admins defined for that company, and then permanently records Alice's address as the admin for the company.

Adding users



To add a user, Admin Alice goes to her contacts list within the HighSide application and clicks *Invite People*. She inputs Bob's name and Bob's email address. Her client then generates a signup token for Bob, saves a hash of the signup token in her personal contacts list, and sends Bob's name, Bob's email address, and the signup token to the server. The server emails Bob telling him that Alice has added him to HighSide and includes the signup token. The server saves Bob's name and the hash of his signup token to the database. Alice's client, meanwhile, uploads a new signed complete copy of her contacts which includes Bob's name and the hash of his signup token. This is distributed down to other users who see Bob show up in their contacts list. They each individually request Bob's public keys based on the hash of the signup token- a request that the server cannot fulfill because it does not yet know Bob's public keys. All of this happens within a second or two of Alice adding Bob to her contacts list.

Some time later, Bob installs HighSide and enters the signup token that he received in his email. His client generates keys and connects the same way Alice's client did when she first signed up. The server sees that his signup token is associated with Alice's company and that an address is not yet assigned to the contact. It assigns Bob's address (which the server generates from the pubkeys). The server then notifies everyone else in the company that the hash of the signup token, of which they were previously made aware from Alice's contacts list, is associated with Bob's address and pubkeys. Each user finds the hash of the signup token in their contacts, observes that there is no address associated with the entry, and then saves Bob's address. Alice does this too. From this point forward, when Alice uploads a signed contacts list, it includes Bob's actual address instead of the hash of the signup token.

Any users who were offline when this exchange occurred will receive an updated contacts list when they connect. They will then request any missing pubkeys which they do not already have stored.

Authentication

Clearly everyone needs to be sure that they are talking to the correct person instead of an attacker. You might notice that the process above is *Trust On First Use* which means that the clients are all trusting the first address and public keys which come down from the server. Once people are connected, everyone's address is shown in everyone else's contacts list. If you want to *verify* that you are talking to Charlie, for example, you can go to him and make sure that Charlie's address is the same in your client as it is in his. Likewise, he should make sure that your address is the same in his client as it

is in yours. After that point, both you and Charlie can be sure that you are talking to the correct person and that no man-in-the-middle attack can take place. If the HighSide server starts acting maliciously and sends down different keys for either you or Charlie, neither client will accept them. This method of key authentication is great in the sense that no one has to trust a third party. This is how our competitors handle key authentication. The problem is that we know from experience that users generally do a terrible job of verifying each other's PGP key fingerprints, a server's SSH key fingerprints, or fingerprints within apps. They mostly just don't understand why it must be done. HighSide aims to help this situation.

Each company has one or more *admins* whose job it is to invite users to the company and verify their addresses. If Bob and Charlie both trust Admin Alice to check people's addresses then they need not each individually verify all of their co-worker's addresses. Under this system, messages and files you send will be secure even if HighSide's own servers are hacked. Verified users will have a special visual designation in the contacts list to motivate other users to become verified.

Sending and Receiving Messages

Suppose Alice wishes to send a message to a group in which Alice, Bob, and Charlie are participating.

1. Alice generates a random 16 byte msgID
2. Alice generates a random 32 byte AESSeed
3. Alice appends the following things together in a variable called "payload":

1. message version
2. +msgID
3. +number of part IDs (used only for attachments)
4. +her address
5. +the 16 byte group ID (this was chosen earlier by the server)
6. +the current time
7. +the number of recipients (3 in this case). For each recipient (including herself):
    1. +the address of the recipient
    2. +ciphertext of the AESSeed encrypted with this destination's public key

4. Alice appends the following things together in a variable called "plaintext":
    1. +Alice's address
    2. +the message purpose. This is a standard message so messagePurpose = 1.
    3. +the message data
5. Alice does a double-SHA512 hash of the AESSeed. The first 32 bytes are an HMAC key. The last 32 bytes are the AESEphemKey.
6. Using AES-CTR, Alice encrypts the plaintext with the AESEphemKey.
7. Alice HMACs the ciphertext with the HMAC key.
8. Alice appends the following things to the end of payload:
    1. +the HMAC digest
    2. +the AES ciphertext
    3. +a signature (ECDSA, SHA256) covering all of the payload data accumulated thus far
9. Alice then queues the payload to be sent to the server.

10. Server acknowledges the message based on msgID so that Alice knows that the server received it okay (Alice will resend automatically in 10 seconds if the ack doesn't arrive)
11. Server saves the payload in the database
12. Server prepends its *own* message version and time stamp to the front of the payload
13. Server relays the payload to the listed recipients who are in the group
14. Bob downloads the payload and sends an acknowledgement to the server. This acknowledgement is specific to Bob's client so that if Bob is has another computer with the same keys, the server will send this same payload to it also.
15. Bob verifies that the server's time stamp is not more than 65 minutes in the future
16. Bob verifies that he has not already received this msgID
17. Bob verifies that the sending address is present in his contacts list, sees that it is from Alice, and he verifies that he has her pubkeys.
18. Since this is a message to a group, Bob verifies that he and Alice are in the group
19. Bob verifies that Alice's time stamp is reasonable
20. Bob checks Alice's signature which covers everything in the payload except for the server's message version and the server's time stamp.
21. Bob finds his header in the payload and decrypts the ciphertext using his ECC key. He now has the 32-byte AESSeed.
22. Bob does a double-SHA512 hash of the AESSeed. The first 32 bytes are the HMAC key. The last 32 bytes are the AESEphemKey.
23. Bob checks the HMAC using the HMAC key.

24. Bob decrypts the main ciphertext using the AESEphemKey.
    If anything goes wrong with any of this decryption process in such a way that would allow others to read the message but not Bob, Bob sends a message to everyone else in the group notifying them that he was unable to read the message which the others will display immediately with special formatting.
25. Bob checks to make sure that Alice's address from step 17 is the same as the address listed in the plaintext from step 24.
26. Bob saves and displays the message. The server's time stamp is used as the authoritative time stamp for the UI so that users cannot cheat and post reordered messages.
27. Bob goes through the list of people who were *supposed* to receive the message and for each one to whom Alice did not actually send the message, another message is displayed in Bob's UI notifying him of the fact that they could not read Alice's message.

For attachments, HighSide reads 2MB of a file at a time, compresses it, encrypts it, hashes it, sets the first 16 bytes of the hash as a partID and uploads it to the server. A list of partIDs is transferred in a message as described above, along with one big 64 byte hash of the plaintext file data once it is all reassembled for added assurance that the data is unmodified. Received parts can be decrypted and decompressed out of order. Once all of the parts arrive, a typical computer can write the file out to disk at 118 MB/s assuming that the disk can work that quickly.

Turning off compression is not currently supported but likely will be some day.

Auto-updates

HighSide downloads updates and prompts you to install them. On connection, the server tells the client the minimum recommended and minimum required build number. If the client sees that it is obsolete, it connects through SSL to the HighSide website and downloads the latest binary file and also a signature file. Once downloaded, it does a signature verify operation using the binary file, the signature file, and two HighSide public keys which are hard-coded in the source code. If one of the two verify operations passes then the binary file is saved to disk and the user is prompted to upgrade. Our private keys are secured on dedicated permanently air-gapped password protected Linux machines.

Thank you. For more information, please reach out to us directly at contact@highside.io

Last updated: 6/12/2017*

*Edited formatting.