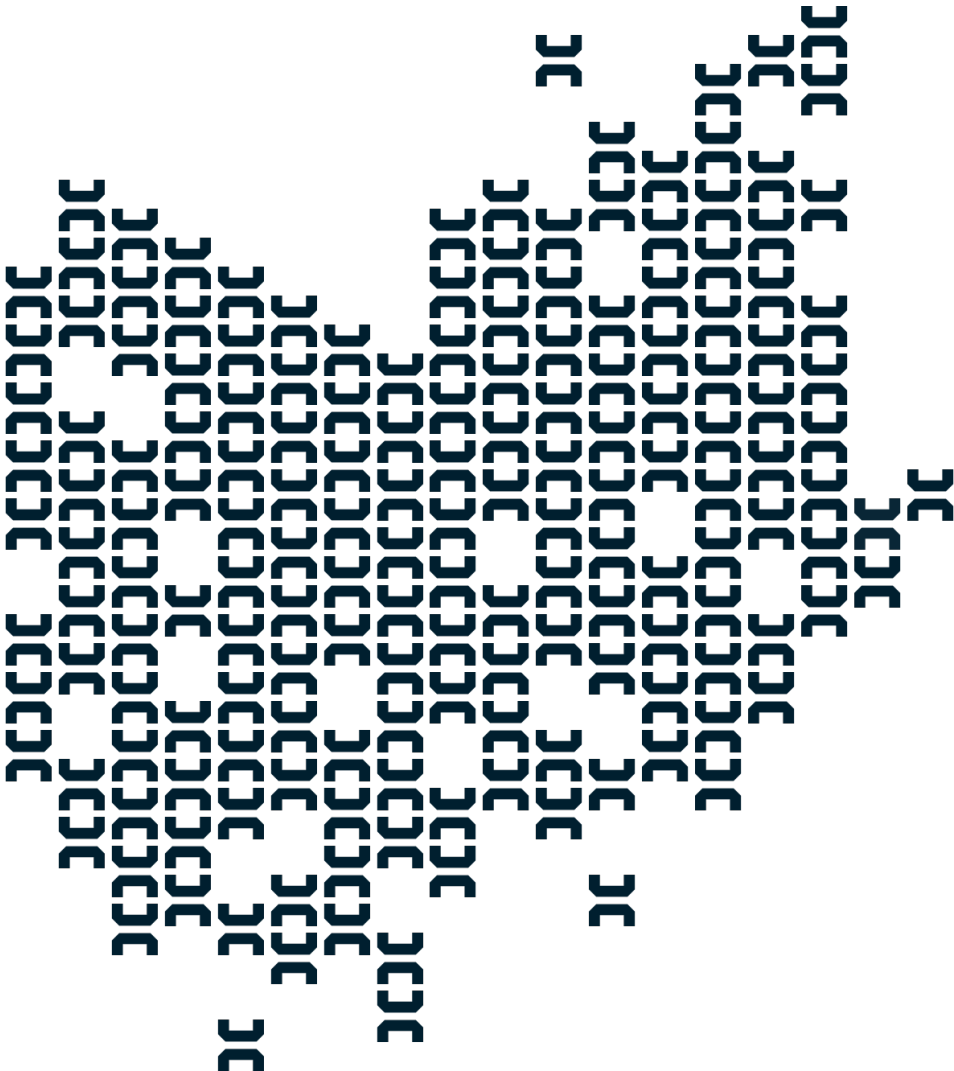
HIGHSIDE

How **HighSide's** distributed Encryption & Authentication Protocol Works

HighSide's desktop & mobile application (not SecureTeams)

TechBrief

Q1 2021



HIGHSIDE

HighSide's Encryption & Authentication Protocl

HighSide applications leverage the HighSide distributed Encryption & Authentication Protocol (HEAP) to encrypt transmissions and broker access to the encrypted content. The following content is specific to the application of HEAP within the HighSide stand-alone desktop and mobile applications. While other integrated products, like SecureTeams, leverage HEAP, the application and operation may vary.

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

TL;DR 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-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 in your team; 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 channels

HIGHSIDE

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.

Table of Contents

- Creating keys 3
- Creating the team 4
- Adding users 4
- Authentication 5
- Sending and receiving messages 8
- Auto-updates 11

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, 146 bits of random data is generated
2. These 146 bits, plus 8 bits to use as flags, are used to select 14 words from the word list.
3. These 14 words are saved as the "secret key". This secret key can be viewed in the HighSide UI.
4. PBKDF2 is used to "stretch" the secret key. It uses a hard-coded salt, 4096 iterations, and SHA512.
5. This secret key is then appended with a nonce
6. The secret key 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, and 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 team

An admin, Alice, will create an account in the client for herself by inputting the team name, her name and her email address; the email address is only used to notify her that something is amiss (like someone used her private secret key in a new client). The client generates some new encryption keys, connects to the server, and tells the server that it wishes to make a new team. The server then permanently records Alice's address as the admin for the team.

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 email address. Alice's client then generates a cryptographically signed 'contact' message that includes Bob's name and assigned user group. Other clients will later use this to make sure that Bob was added to the team by a team admin and not the server acting maliciously.

Upon receiving this cryptographically signed contact message and seeing that it is new and that Bob's email address is appended, the server generates a signup token for Bob and emails him saying that Alice has added him to HighSide. The welcome email includes the signup token. At this point, Bob shows up in other users' contact lists.

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 team and that an address is not yet assigned with the contact. It assigns Bob's address to the contact which the server generates from Bob's supplied public keys. The next time Alice connects to the server, she will upload a new contact for Bob that includes his address.

At this point, if an attacker or the server tries to replace Bob's keys with his own in order to impersonate Bob, he would not be able to because Bob's contact, which contains Bob's address, is cryptographically signed by Alice. If the attacker modifies the existing contact, no other clients would accept it. If he tries to create a new contact with a maliciously-spelled name like B0B instead of Bob in an attempt to trick busy people, the attack would fail because the new contact is not signed by Alice.

Authentication

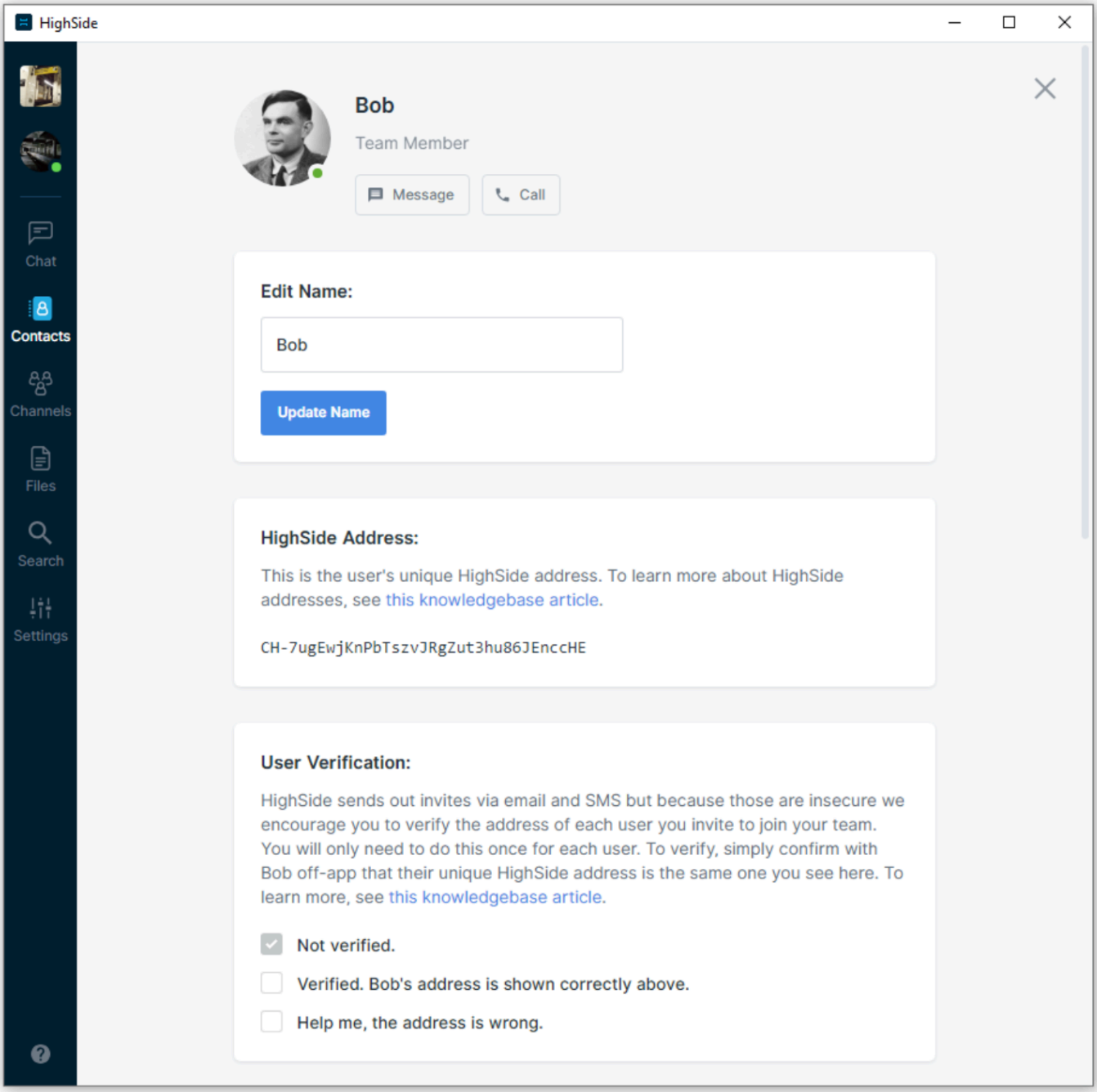
Clearly everyone needs to be sure that they are talking to the correct person instead of an attacker. You might notice that the process above is Trust On First Use which means that the clients are all implicitly trusting the first address and public keys which come down from the server. If the attacker wasn't present and executing a man-in-the-middle attack when you joined your team then Trust On First Use is good enough. But how can you be sure?

Verification

Once users connect to the server once, their address is shown on their Team Member page. 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 team has one or more admins whose job it is to invite users to the team 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.


HighSide
Chat
Contacts
Channels
Files
Search
Settings
Bob
Team Member
Message
Call
Edit Name:
Bob
Update Name
HighSide Address:
This is the user's unique HighSide address. To learn more about HighSide addresses, see this knowledgebase article.
CH-7ugEwjKnPbTszvJRgZut3hu86JEnccHE
User Verification:
HighSide sends out invites via email and SMS but because those are insecure we encourage you to verify the address of each user you invite to join your team. You will only need to do this once for each user. To verify, simply confirm with Bob off-app that their unique HighSide address is the same one you see here. To learn more, see this knowledgebase article.
Not verified.
Verified. Bob's address is shown correctly above.
Help me, the address is wrong.

Adding Admins

Alice may promote a second user, Bob, to be an admin. To do this, Alice uploads a cryptographically signed and timestamped message to the server saying basically, “Bob, using this address, is now an admin too”. The server distributes this cryptographically signed message to users on connection who check the signature and then add Bob to their list of admins for this team. There is no practical or theoretical limit to the number of admins in a team.

Sending and Receiving Messages

Suppose Alice wishes to send a message to a channel 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 channel 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
3. +1 or 0 – indicating weather this recipient was @mentioned. The server uses this when deciding whether to send a push notification.

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 channel

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 checks whether he already has Alice's address and contact stored locally. If not, he requests it from the server and saves this full message structure to be processed later. After it arrives, Bob starts over processing this message.
18. Since this is a message to a channel, Bob verifies that he and Alice are in the channel
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 channel notifying them that he was unable to read the message which the others will display immediately with scary red 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.

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.

Local Storage Encryption

HighSide clients receive missed messages on connection and new messages while connected. This message data is written to disk so that the local message search tool can work efficiently and so that messages display quickly in the UI when clicking around the client.

On the desktop application and an upcoming version of the mobile apps, locally stored sensitive data like message content is encrypted with a key stored in the server. After the client connects successfully, the server sends down this 'local storage key' for clients to store in memory to use to read and write their locally stored message data to disk. AES-256-CTR+HMAC is used for this operation. This way if a laptop is stolen that is powered off or where the HighSide client isn't running, the thief would not be able to recover any message content.

This is also useful if an administrator wants to use HighSide's 'Remote Wipe' feature. With this feature, the server sends down a message telling the HighSide client to erase all messages, keys, and all other ancillary information used for this team and then to exit. The 'wipe' command is cryptographically signed by the admin and verified by the recipient. But the wipe command only works if the client comes online to receive it. A recently fired disgruntled employee might not connect his work laptop to the Internet and may instead hand it to a competitor or foreign government and so his client might never receive the wipe command. Fortunately, local storage encryption prevents an attacker who images the user's hard drive from getting the message content even with the voluntary assistance of the user.

Location Restrictions

In the settings area, admins may apply a time or location restriction to groups of users and this setting is relayed down to users in the cryptographically signed 'company settings' message.

The time and location restriction is enforced client-side. On mobile, clients get their location through their normal location services provider. On desktop, clients make a list of nearby Wi-Fi access points and relay that information to the HighSide server which uses the data to look up the device's location. This is only done if the location restriction is turned on. The threat model used for securing messages, files, company settings, and team membership is much stronger than that used for the location restriction.

Scalability

There is no theoretical maximum number of users that can be added to a team. There are no operations users perform that involve 'everyone' in the team. For example, when Bob comes online, the server does not notify everyone in the team. Instead, only people who have direct message conversations open with Bob are notified so that they can display a green dot next to his name.

Channels are currently limited in size to 100 people but this will be raised over time as cell phones get more powerful.