Avoiding Leaps of Faith: Trustless Identity Verification with Ethereum

George Kudrayvtsev

December 26, 2018

Abstract

Applications that provide encrypted communication, whether messaging platforms, remote access tools, or secure websites, rely on an implicit chain of trust that is easily broken. A malicious entity existing on the path between two users – which can include the platform provider itself – can imperceptibly read the contents of what is perceived to be an "encrypted" communication channel. This paper presents a countermeasure: Ethereum-backed identity verification. By using the Ethereum blockchain to securely store authentication details, users can verify the authenticity of any incoming data independently, without trusting its source. We integrate this countermeasure into an encrypted messaging platform called Beacon that provides unparalleled and verifiable security against such interception attacks.

I. Introduction

Users value the security and privacy of their digital communications now more than ever. The latter half of the decade saw breaches compromising and exposing over two billion records containing personal and financial information [4]. Such breaches are unavoidable in the game of cat-and-mouse between hackers and security engineers, but the scale is unlike anything we have seen before. For many corporations, such breaches are just a cost of doing business. For others, security becomes a selling point. Messaging clients like Facebook and Skype introduced "secret conversations," claiming to offer communication privacy that is resilient to breaches and data collection [9, 15, 19]. Such claims increase confidence but lull users into a false sense of security: conversation "secrecy" is entirely contingent on trusting the medium through which the conversation occurs. The exchange of encryption keys occurs through an untrusted 3rd party: the platform itself. An adversarial platform can easily perform a *man-in-the-middle attack* which would compromise the privacy of conversations that pass through it. This is a well-known weakness of asymmetric cryptography among security researchers, but is misrepresented by products claiming complete "end-to-end encryption."

Are such violations of user trust a realistic concern? Though state-level actors have been known to attempt to infiltrate and coerce secure messaging platforms to control key exchanges [29, 30] or perform mass-scale man-in-the-middle attacks on Web traffic [7, 8, 13, 23], there is no existing evidence of *platforms* compromising their users' key exchanges. There is significant incentive, though; at the very least, data mining of behaviors and conversation patterns are a digital gold mine ripe for exploitation. Given corporate track records for ethical behavior, the potential of such silent manipulation should not be taken lightly. Whether by weakening the strength of encryption keys or compromising the exchange entirely, "clearly there are many nation-states and non-nation-state organizations with incentives and expertise suitable to perform this type of sabotage" [24].

Our Contribution

In this paper, we present and then solve what we call the "authentic key distribution problem." We begin by outlining the existence of this problem in a number of modern applications which motivate our subsequent solution. Section 2 provides a simple survey of common key distribution techniques.[1] Section 3 examines the potential of Ethereum-backed identities and section 4 integrates them into a key-exchange architecture. Then, section 5 combines all of these principles into an authentication flow that can operate through an adversarial platform and still achieve authentic key distribution. Section 6 discusses the potential shortcomings of this protocol and section 7 wraps things up by analyzing the untapped potential of blockchain-backed identity au-

[1] If desired, a thorough, rigorous analysis of the weaknesses specific to *messaging platforms* is available in [27].

thentication.

II. Shortcomings of Modern Encryption

Modern encryption relies on asymmetric cryptography; it enables two parties to establish a *shared secret* that's unknown to the outside world. They can then use this shared secret to encrypt future communications. The problem with asymmetric cryptography lies in *authentic key distribution*: given a public key and an associated identity, there's no way to validate the pair's authenticity. Thus, the means through which keys are acquired is vulnerable to interception and modification – a *man-in-the-middle attack*.

For a real world analogy, suppose you want to get in touch with Bob, a stranger. You have a mutual connection, Joe, who is your acquaintance. You ask Joe for Bob's contact information, and Joe gives you Bob's business card, which says Bob's email is `boblegit@email.com`. You have no choice but to trust Joe, so you email Bob and introduce yourself. Joe, though, is actually a malicious bad actor who gave you a fake business card. You proceed to converse with "Bob," but in reality are communicating with a person of Joe's choosing, who we'll call Mallory. Even more sneakily, perhaps Mallory silently forwards all of your messages to Bob: thus, you and Bob think you're talking to each other confidentially, but in reality, Mallory sees everything you say. Here, Mallory is the man-in-the-middle attacker.

How could you have exposed Joe's lie? The only way to be *completely* sure of Bob's contact information is to hear it from the source; in other words, you *must* ask Bob himself. Unfortunately, if Bob lives far away, this becomes a Catch-22: to ask Bob, you need his address; to get his address, you need to ask for it from someone you know.

Unfortunately, this problem directly translates to communication over the Internet. To securely connect to `https://www.bank.com` (or chat with Bob over a messaging platform, or log in to a work VPN, etc.), you need to ask someone you know for the relevant *public key* that would allow you to encrypt communication to them.

Countermeasures

There are a number of countermeasures to the key distribution problem that try to make the information you receive more trustworthy.

Leap-of-Faith Authentication

The simplest way to establish trust is to assume it. This authentication method relies on a "leap of faith" that trusts identity information the first time it is provided. Once the stream is established, all subsequent communications can be secured by comparing the presented identity against the result of the initial leap, but the first leap relies on completely blind trust. This method is most-famously used by the OpenSSH protocol:

```
$ ssh git@github.com
The authenticity of host 'github.com' can't be
established. RSA key fingerprint is ____.
Are you sure you want to continue connecting?
```

The fingerprint is then saved and validated on all subsequent communications, but fundamentally, the *entirety* of this method's security hinges on a critical assumption: the attacker is unlikely to be present during the first communication with a server [21].

Chain of Trust

The most widespread countermeasure to the authentic key distribution problem is establishing a "chain of trust." To continue with our analogy, instead of asking Joe—an untrusted acquaintance—you ask Alice, your best friend. She may know Bob directly, but that is an unlikely scenario for every possible "Bob" you'd ever want to talk to securely. What is more likely is that Alice knows a guy who she trusts, who knows a gal who he trusts, . . . , who knows Bob. You must then trust this chain to deliver you Bob's contact info. Naturally, if *any* person on this chain decides to act maliciously, all bets are off.

This technique is common on the Web: users trust a Certificate Authority (or CA) to authenticate the public key of the websites they visit. Websites provide a "certificate"[2] when you visit them; if it is valid, you can trust it. "Valid" means the certificate is eventually signed by a "Root CA." The Root signs certificates of a smaller CA, which signs the certificate of a smaller CA, . . . , which signs the certificate of the website you are connecting to [22]. Much like how Alice, as your best friend, is the implicitly-trusted beginning of the chain to Bob, the Root CA is hard-coded and implicitly trusted by your browser when confirming a certificate chain.

Now, you are unlikely to have more than a handful of best friends you would entirely trust with a chain of such critical identity information. Not so with

[2] A *certificate* is essentially a collection of identity information for a website (its name, its public key, etc.) and third-party confirmations of that information [20, 25]

certificate authoritities. To put the scope of this chain into perspective, Firefox trusts 164 Root CAs; the other browsers rely on the operating system's trust store. By default, Windows has over 400 and macOS has over 200 trusted root CAs [10]. That's a *massive* set of authorities to implicitly trust to behave correctly; after all, the U.S. trusts just *three* national credit bureaus with financial information, yet (at least) one of those has been breached [11].

What happens when mistakes or breaches occur? For example, what if a Root CA accidentally signs certificate used for a malicious website, or its signing algorithm is exploited to forge certificates? If the issuer has the infrastructure to modify the trusted certificate store of all of its users, like Microsoft did, it can salvage the situation by issuing a patch to ignore these erroneously-validated certificates [1, 2]. This type of solution is not always possible and definitely does not scale; instead, the main countermeasure is giving certificates expiration times and simply waiting it out...

Certificate Pinning When a system has more knowledge about its usage, it can perform a technique known as certificate pinning to preemptively tie specific certificates to specific authorities. For example, the Chrome browser comes bundled with a set of Google certificates which it stores locally. Whenever the browser accesses a Google website, it checks to ensure the website's certificate uses a public key in that set [20]. In other words, it cross-references the certificate chain against its known, locally-stored whitelist of allowed values.

Out-of-Band Verification

As we noted in our analogy, the way to get Bob's contact information without trusting anyone else is by asking him ourselves. We've established that this is largely impossible over the Internet. Suppose then, that in addition to our acquaintance Joe, we also have another acquaintance, Sally, who knows Bob but does *not* know Joe (so they can't conspire against us). Then, we ask them both for Bob's contact information and compare the results. If they match, we can have a higher level of confidence in the result compared to just asking one of them.

On the Internet, this essentially involves confirming the authenticity of a public key received through one channel through another, out-of-band, channel. Two popular secure messengers, WhatsApp and Signal, provide a means to verify the public keys being used in a conversation. WhatsApp—a Facebook product—is a closed source application, so again there is a necessity to trust that the public key it displays is genuine *and* the one actually being used for the communication stream. The source code to the Signal messenger is freely available, so its out-of-band verification method is more trustworthy.

This method *does* ensure the authenticity of the identity of a recipient *if* you trust that the message is delivered without modification,[3] but it is not a seamless process. The problems with out-of-band verification lie in its manual nature. Both Signal and WhatsApp require users to either scan each other's QR codes in person, share them with each other via social media or another platform, or match their keys up over a phone call [16, 28]. This process is painful, not automatic, and compromises the isolation of the conversation. **Sharing another point of contact with recipients** (and trusting that those details are delivered authentically) **is neither feasible for everyone nor desirable for every type of conversation**.

Of course, if a messaging platform makes claims of end-to-end encryption and does *not* provide a means of out-of-band verification, it demands a much greater level of trust from its users. The platform itself must be trusted entirely to deliver identity and encryption information unmodified between users; in other words, you trust the messaging platform to provide your recipient's public key. The platform could easily and imperceptibly replace the public key—much like Joe in our analogy—allowing it to read all of the messages that pass through. As we discussed in the introduction, such an imperceptible violation could tap into a digital gold mine of private user data.

Trusted Nodes

This method is often used in peer-to-peer-based systems[4] to establish trusted links between untrusted parties. It is a blend of the two previous countermeasures: when two peers wish to communicate, they exchange identity and key information through trusted peers. These trusted peers are often hardcoded or acquired through a chain of trust rooted in a trusted peer (much like the Certificate Authorities in Chain

[3] Again, barring large-scale collaboration among all of the platforms you use conspiring to *all* lie to you and perform identical man-in-the-middle attacks, but we will (perhaps naïvely) dismiss these as infeasible.

[4] We say "based" here because a true peer-to-peer system would treat *all* nodes equally. In such a system, every peer would be (un)trusted equally and so key distribution is as (un)reliable as that level of shared global trust.

of Trust). If necessary, they can compare the exchange across two or more trusted nodes (like the separate channels in Out-of-Band Verification) that could have become trusted through different means.

Naturally, the pitfalls of this system are nearly identical to the other sections. It requires "middleman" peers, which are effectively a centralized, implicitly-trusted authority. The potential is somewhat greater, though: a theoretical system that used a variety of trusted nodes—all acquired through independent means—would result in more trustworthy information; cross-referencing exchanges through multiple nodes would result in more reliable consensus. It would be akin to having certificates be signed by multiple independent Root CAs, or Alice asking Joe, Sally, and four other friends for Bob's address.

The reason this countermeasure is worth mentioning is to demonstrate that these weak points of trust are not limited to centralized communication systems. Furthermore, it demonstrates that there is both value in relying on consensus between multiple independent authorities as well as risk in that there is a larger distribution of trust across those same authorities.

In summary, the shortcomings of modern encryption lie largely in their reliance on trusting a centralized authority that's undeserved, unprovable, or—in the case of Signal's open source policies and non-profit structure—cumbersome to verify.

III. Ethereum-Backed Identities

We present a solution to the authentic key distribution problem that uses the Ethereum blockchain as a trusted 3rd party. This naturally begs the question: *Why trust Ethereum?*

The Ethereum blockchain is a highly-distributed, public, decentralized, append-only, constantly-verified database that stores transactions that manage, transfer, and track billions of dollars in wealth. Consensus on its state is mathematically verified quadrillions of times per *second* by millions of independent parties (miners). The aspect of widespread *independent* consensus on the state of the blockchain encourages a level of trust impossible to achieve among centralized systems. Furthermore, the fact that such a system has been entrusted with billions of dollars of wealth encourages its security, as the incentive to break the fundamentals of the blockchain is incomparable.[5] Thus, the architecture in this paper relies on trusting the validity of the Ethereum blockchain.

The Ethereum blockchain also features a virtual machine, allowing for code in "contracts" to execute across the network in a similar, publicly-verifiable way. In other words, an action or piece of code executing on the network means that all of those who have executed it have agreed upon its results.

We present a contract (whose code is presented in Appendix A) that provides association between usernames and their respective cryptographic identity. The contract acts as a simple key-value mapping, and the Ethereum blockchain ensures its validity. An individual[6] registers a username, giving them permanent ownership of it, then associate it with a Ed25519 public key that can be used to prove authenticity of *any* future communication. In the following section we will demonstrate how this contract can be used to create authentic key exchanges over an adversarial, centralized platform.

Prior Work

Using the blockchain as a source of identity information is not entirely a novel idea. There is a well-known problem in distributed naming systems known as Zooko's Triangle. It states that names can have up to *two* of the following properties: distributed, secure, or human-readable [31]. Security legend Dan Kaminsky and the late Internet hacktivist Aaron Swartz discussed the potential of Bitcoin to power a system providing these three properties [14, 26]. These theoretical efforts were realized by Namecoin, a Bitcoin fork aiming to provide a DNS alternative that enforced all three properties of the Triangle: names that are meaningful to humans, secured with cryptography, and decentralized by the blockchain's very nature.

Namecoin is marketed as a generic key-value store, but is primarily used for a decentralized DNS. It tightly couples the principles of the blockchain with the stored mapping, and incurs the limitations of Bitcoin as a network. There are significant limitations of Namecoin as a platform for exchanging identity information for secure messaging, and

[5] There have obviously been instances of money being stolen from cryptocurrency wallets, but these exploits were based on faulty contract code, weak secret keys, and other methods that do not violate the *mathematical principles* behind the blockchain's validity.

[6] well, specifically, a particular Ethereum wallet

no effort towards such a platform has been made [27]:

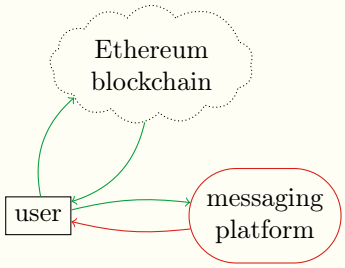
> *The system requires users to pay to reserve and maintain names, sacrificing* low key maintenance *and* automatic key initialization. *Users also cannot instantly issue new keys for their identifiers (i.e., there is no* immediate enrollment*) but are required to wait for a new block to be published and confirmed. In practice, this can take 10-60 minutes depending on the desired security level.*
>
> — *Unger* et al.

The platform presented in this paper eliminates the maintenance cost for names and initializes keys automatically. It also vastly improves the presented issue of "immediate enrollment": the Ethereum blockchain publishes blocks on the order of *seconds* rather than minutes. Furthermore, the Ethereum contract presented in this paper decouples key-value storage from its underlying blockchain; this is a significant architectural improvement and allows the two to develop independently.

Even more significantly, such systems have not been integrated into an end-to-end solution; the Beacon platform is a realization of these ideas and provides the first end-to-end encrypted messenger whose key exchange is not vulnerable to an adversarial platform provider.

Security Assumptions

From a high level, a blockchain-backed identity architecture looks something like this:



Green lines are trustworthy, while any data arriving along the red line must be externally verified. The implicit trustworthiness relies on the following security assumptions:

- **The blockchain can be trusted.** This is obvious and has already been discussed at length. There is an important caveat when introducing lightweights clients (like mobile users): storing a self-verified copy of the blockchain locally does not scale, nor is it always technically feasible. As such, users must trust a blockchain *provider* such as Etherscan or Infura to enable genuine interactions with Ethereum.
- **The contract hash is explicitly known by every user**. If the hash is gleaned from an external source, it is susceptible to an impersonation attack (as described in **Attack Vector**: Contract Impersonation). This is enforced by hard-coding the contract address into clients; because clients can be made freely by just following the protocol or compiling the open source code, there is a high risk and low incentive to the attack.
- **Usernames are unique**. This is enforced trivially by looking at the contract code in the Appendix A: line 19 enforces that a username must never have been previously registered. The contract also enforces a strict subset of available characters:

  ```
  0123456789[\]^_`
  abcdefghijklmnopqrstuvwxyz
  ```

  in order to ensure that there is no case-sensitivity impersonation risk (such as starting a conversation with `Alice` instead of `alice`).[7]

Additionally, users must know the precise username they wish to contact; server-side search results cannot be trusted. This is not a significant or unique problem, though: the same issue can be seen in other platforms such as text messaging, in which you must know (or recognize) your recipient's phone number.

IV. Trustless Architecture

If all of the principles outlined in Security Assumptions hold, the design guarantees an authentic key exchange and thus confidentiality in the subsequent conversation. This section covers the two key operations involved in a establishing such a communication stream: registering a username and starting a conversation.

Registration

As we've discussed, users must claim unique usernames and associate them with a cryptographic identity. Registration is split into two parts:

[7] Refer to the Case-Insensitivity section in the Appendix A if interested in a breakdown on why only *this* particular subset of characters is allowed.

blockchain registration and platform registration. In order to interact with the blockchain, the user needs an Ethereum wallet with a non-zero balance to pay for transactions. Wallet management and transaction handling is generally a tedious process, but we will establish an automated method of performing blockchain interactions later (see Seamless Mobile Wallets).

For the aforementioned "cryptographic identity," we use a 64-byte public key on the Ed25519 elliptic curve [3]. Keys on this curve allow us to achieve high security with shorter key lengths than traditional methods such as RSA [5, 12]. This key will henceforth be referred to as the *identity key*, and it will be used exclusively for *signing* future messages to cryptographically associate them with their user. When a user calls `register` on the contract, they permanently link their Ethereum address, their chosen username, their identity key, and the contract together in a public Ethereum transaction, which can be referred to by its unique transaction hash.

The user can now register with the platform, providing the aforementioned hash. Here, though, they can provide more superficial registration details such as a "real" name, an email address, etc. This data is signed by the identity key and can hence always be externally verified and associated with their permanent username.

Conversations

In order to establish a secure conversation, two users need to exchange their identity keys without fear of impersonation or modification by the platform. Once they have done so a single time, securing the rest of their communications is trivial.

The first user, Alice, asks the platform to provide the data needed to communicate with another user, Bob. Ordinarily, this is where the platform could violate user trust, providing alternative public keys and sneakily decrypting all subsequent messages. Here, though, Alice confirms the server-provided data by independently validating it against the Ethereum blockchain.

This could be done via executing the `lookupUser` method on the contract and validating the signature on server-provided registration blob, but is not preferred because it requires additional funds (from both the initiator *and* the recipient!) to execute the contract function on the blockchain.

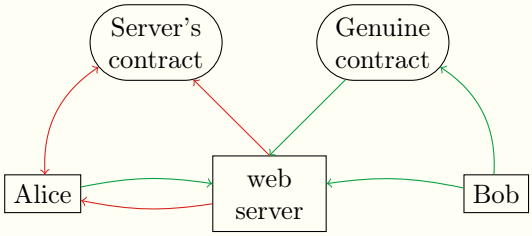
A better approach avoids direct execution of contract code. The server provides a single "unconfirmed" point of data that it claims is signed by Bob's identity key: the transaction hash from Bob's registration. Alice can independently look up and verify what occurred at that transaction. If legitimate, the transaction would contain the raw parameters that Bob passed to `register`. Alice needs to verify that:

- the transaction's smart contract matches the universally agreed-upon hash,
- the username matches exactly what Alice expects Bob's to be,
- the identity key stored by Bob on the contract matches the one in the server-provided information, and
- that same identity key validates the signature on the blob of "Bob"s data provided by the server, establishing that the rest of the data is also valid.

If these properties hold, Alice knows the server-provided details about Bob are legitimate and can use them to establish a secure communication stream.

***Attack Vector**: Contract Impersonation*

Recall the requirement in Security Assumptions that specified that all users know the contract address. If this were not the case, the server could create a separate contract, registering the same users with *server-controlled* identity keys, and use that as a relay between the "real" data and the impersonated data in order to eavesdrop on messages exactly as in the traditional model.



A hard-coded contract address in each client prevents this sort of redirection, and is an essential point of verification on Alice's part when starting a conversation with Bob.

Updating Keys

A handy method available in the Ethereum contract is `updatePublicKey`. A user's stored public key is independent of the Ethereum address key pair, so it is

possible to update the key after the initial registration by calling this method. Just like registration, this incurs a cost as a transaction on the Ethereum network (see the Key Update Cost section in Appendix A for a breakdown).

This operation is valuable following a key compromise. Though the user would have to be faster than an attacker, they could update their identity key on the blockchain (and subsequently on the platform) and maintain control of their account. Key compromises, though serious, does not risk the decryption of any previous communications; the Beacon protocol maintains perfect forward secrecy [6, 17].

The transaction hash originally submitted to the server during registration will continue to only point to the old identity key. Until the server is notified of the new one by the user, conversations using the previous key will continue. Key updates are guaranteed to be initiated by the username owner: not even the contract creator (i.e. the messaging platform) can change them (see lines 38–39 of the contract in Appendix A). This means that once the handshake key bundle is updated to include a pointer to the `updatePublicKey` transaction, recipients can be sure of its validity if they can correlate it with the initial `register` result. Thus, recipients are responsible for tracking rolling updates to the identity key and the multitude of transactions pointing to those updates.

V. Seamless Mobile Wallets

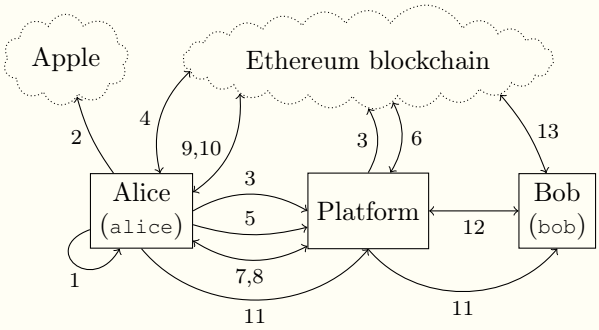
Users of a secure messaging platform are not interested in the blockchain, cryptocurrency, wallet balances, etc. They simply want a seamless way to ensure that their communications are private and safe. The platform we have outlined thus far requires a fee to pay for blockchain registration; this cost is unavoidable, so it must be hidden from the user to ensure a streamlined experience. There is a technology that users *are* familiar with, though, and that is in-app purchases. By presenting costs to the user in the local fiat currency, we simplify the user experience and reduce the necessary understanding to use the platform.

On their first launch, users generate a secp256k1 key pair locally. Their corresponding Ethereum wallet address can be derived from the public key [32]. To register, the user makes an in-app purchase to pay for the initial registration fee. This provides the funds (in fiat) necessary for the blockchain registration. The purchase receipt is validated server-side to ensure its authenticity, and the platform deposits the necessary funds directly into the user's mobile wallet from its own wallet.

Though this fund transfer incurs the delay blockchain processing delays as well as an extra transaction fee, this is simply factored into the overall one-time cost of registering a username. The cost derivation is outlined in the Registration Cost section of Appendix A.

VI. End-to-End Encryption, the Right Way

This diagram combines all of the pieces into the specific flow for starting a conversation between Alice, who is registering as a new user, and Bob, an existing user, whose username is simply `bob`.



1. Alice generates a secp256k1 key pair and derives her Ethereum address:

$$\begin{aligned} V_A, &\quad \text{Alice's Ethereum private key} \\ P_A, &\quad \text{Alice's Ethereum public key} \\ E_A = \textsc{Keccak}\left(P_A\right)\big|_{96..255}, &\\ &\quad \text{Alice's Ethereum address} \end{aligned}$$

   She also generates her identity key pair, (I_{Av}, I_A), from the Ed25519 elliptic curve. This curve is used for signatures because of its congruence with Curve25519 which is used later for the secret key exchange [3, 18]. Additionally, disassociating from the Ethereum key pair allows it to be updated independently in case of a potential compromise, discussed later in Updating Keys.

2. Alice makes a purchase to fund the registration fee, resulting in a purchase receipt signed by Apple (or any other payment platform).

3. Alice submits the purchase receipt for "reimbursement" along with her Ethereum wallet ad-

dress to the server, who then deposits the necessary funds into her wallet.

4. Alice uses these funds to register her public key on the blockchain, resulting in a transaction hash, Tx_A.

5. Alice builds her registration blob and signs it with her identity key. It might look something like this:

$$R_A = I_A \parallel \text{Tx}_A \parallel \texttt{alice}$$
$$S_A = \text{Sign}_{R_A}(n)$$

She submits (R_A, S_A) to the server.

6. The server validates that the registration really exists on the blockchain and stores Alice's information locally for performance.

7. Alice, now registered, submits a request to the platform to converse with Bob (well, `bob`).

8. The server responds with Bob's (unconfirmed) registration details: (R_B, S_B). Alice will extract Tx_B from R_B (the inverse of step 5).

9. Alice refers to the blockchain at Tx_B, which should contain Bob's username (`bob`) and I_B.

10. She validates I_B was registered by `bob` in Tx_B, and that I_B verifies (R_B, S_B). If everything checks out, Alice can be sure R_B is authentic.

11. Alice can now begin her secret key exchange algorithm of choice, knowing that she can send Bob an authentic and/or encrypted message. The server stores unread messages until Bob queries for them.

12. When Bob receives a message as part of a *new* conversation, he asks the platform for the same details that Alice did to send the message in the first place.

13. Like Alice, he validates the server's claims. If they are valid, he can behind his end of a key exchange.

The Beacon platform uses a key exchange algorithm based on Signal's well-vetted X3DH. Alice and Bob would have used the algorithm to establish a set of shared encryption keys; these would be used to encrypt their subsequent conversation via Signal's double-ratchet algorithm [6, 17, 18]. All messages pass through the platform and its servers but are indecipherable to anybody but Alice and Bob. There is no chance of a man-in-the-middle attack: they verified each others identities via an out-of-band, automated method and derived authentic, mutually-shared secrets.

VII. Shortcomings

Though this design solves an important trust barrier in communication over the Internet, there are still vulnerable gaps. Though flaws like the compromise of a private key are minimized through ephemeral, per-message key pairs, compromising the Ethereum wallet private key is still potentially devastating. Naturally, obvious weaknesses like operating system or application exploits, physical access by adversaries to devices, etc. are still unmitigated. There are also flaws in the architecture itself that are inherent to having a secure system.

With Great Power...

Private keys are just that, *private*. Losing them, whether through a broken phone, a new phone, or an uninstalled app, means that that account can no longer be recovered. Permanently.

By providing a backup method, such as saving a QR code, adding recovery devices, or a dead-man's switch, the risk can be mitigated. This backup *must* be stored offline, though, otherwise the entire architecture is moot; storing private keys in the cloud is somewhat self-defeating. Most users have multiple devices through which they access their messages; this is a further mitigation because they will have the private key "naturally" backed up in multiple places.

Client Trust

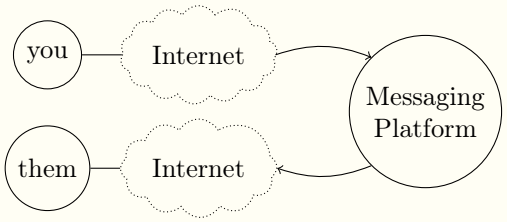
Users no longer need to trust the servers, but they still need to trust the clients. Despite being open-source, clients can still be modified with malicious intent. Without compiling (and sideloading) the app themselves, users cannot necessarily be sure that the source code matches reality.

Despite this trust barrier, the open-source aspect and transparency of the communication protocol itself mean that any number of clients could exist for the platform. Not being tied to a specific messaging app may be a sufficient deterrent to compromising apps in general; compromises can be easily discovered and migrating to another app would be as simple as transferring (and ideally updating) the private

key.

High-Profile Attacks

Despite the fact that we've closed the security gap between the user and the messaging platform, a high-profile actor with limitless resources like nation states could still theoretically intercept and modify the key exchange. The communication between two users across a centralized platform flows like so:



We have secured the communication between the user and the messaging platform (in both directions), but the communication over the Internet is stil vulnerable. A state actor, existing somewhere along the path between a user and its destination, could still intercept values from a key exchange and likewise modify them to suit their needs. This would require highly-specialized targeting, because the attacker would need to know the exact platform, its protocols, and precise user interactions at an exact point in time; this is (fortunately) not generalizable to more naïve attackers and isn't part of our threat model.

VIII. Conclusions and Future Work

This protocol is heavily extensible and can be integrated into *any* platform that requires authentic key distribution. It's neither tied to a specific architecture nor encryption scheme. As we discussed in Shortcomings of Modern Encryption, this includes messaging platforms, encryption on the Web, payment platforms, and plenty of other applications.

The chain of trust behind SSL encryption on the Web can be eliminated with public blockchain identities. These identities would replace certificates signed by "trusted" Certificate Authorities; with appropriate ownership validation methods it may be possible to entirely eliminate the current system of blind trust in CAs.

Because the core communication medium continues to be through a centralized platform, blockchain-authenticated identities can exist on top of *any* existing platforms. For example, an (unencrypted) Facebook Messenger conversation could become secure and private via browser addons. Of course, this requires that both users have the addon installed and are registered with the platform. Naturally, mobile platforms would need true integration to support the protocol; otherwise, the viewed messages would be gibberish.

Email is another area in which the protocol can ensure secure and private communications between users. Much akin to end-to-end encrypted messaging platforms, there are email providers such as ProtonMail that promise email privacy via automatic and seamless PGP encryption. Unfortunately, they suffer from the same trust drawbacks as the messaging platforms: the platform itself can transparently perform a man-in-the-middle attack. Integrating blockchain-backed identities into an email platform would ensure that such attacks are impossible.

We have established an architecture for a platform through which users can communicate without fear of impersonation or eavesdropping. The architecture has been implemented in the first blockchain-authenticated messaging platform we call Beacon. Its non-zero cost may be slightly prohibitive, but it should be alluring to the truly privacy-conscious. Beacon is an experiment in the accessibility, validity, and security of this novel progression in authentic key distribution.

References

[1] Microsoft security advisory 2718704, 2012.

[2] Microsoft security advisory 3046310, 2015.

[3] Bernstein, D. J., Duif, N., et al. High-speed high-security signatures. *Journal of Cryptographic Engineering* (September 2011).

[4] Bisson, D. The 10 biggest data breaches of 2018... so far. *Barkly*.

[5] Certicom Research. Recommended elliptic curve domain parameters. SEC 2, Standards for Efficient Cryptography, January 2010. secg.org.

[6] Cohn-Gordon, K., et al. A formal security analysis of the Signal messaging protocol. *IEEE Symposium on Security and Privacy* (2017).

[7] *but-of-course* department. How the NSA pulls off man-in-the-middle attacks: With help from the telcos. *Techdirt* (October 2013).

[8] *doubtful-that-google-is-happy-about-that* department. FLYING PIG: The NSA is running man in the middle attacks imitating Google's servers. *Techdirt* (September 2013).

[9] Facebook. Messenger starts testing end-to-end encryption with secret conversations. *Facebook Newsroom*.

[10] Fadai, T., Schrittwieser, S., et al. Trust me, I'm a Root CA! Analyzing SSL Root CAs in modern browsers and operating systems. *SBA Research*.

[11] FTC. The Equifax data breach.

[12] Giry, D. *Cryptographic Key Length Recommendation*. NIST and BlueKrypt, February 2017.

[13] ICTR, and GCHQ. Profiling SSL and attributing private networks. WikiLeaks.org, May 2017.

[14] Kaminsky, D. Spelunking the triangle: Exploring Aaron Swartz's take on Zooko's Triangle. January 2011.

[15] Lund, J. Signal partners with Microsoft to bring end-to-end encryption to Skype. *Signal Blog*.

[16] Marlinspike, M. Safety number updates.

[17] Marlinspike, M., and Perrin, T. *Double Ratchet Algorithm*. Open Whisper Systems, November 2016. Signal.org.

[18] Marlinspike, M., and Perrin, T. *X3DH Key Agreement Protocol*. Open Whisper Systems, November 2016. Signal.org.

[19] Microsoft. Skype private conversation. Tech. rep., Microsoft, January 2018.

[20] OWASP Contributors. Certificate and public key pinning. *Open Web Application Security Project*.

[21] Pham, V., and Aura, T. Security analysis of leap-of-faith protocols. *SecureComm* (2011), 337–355.

[22] Rescorla, E., and Mozilla. The Transport Layer Security (TLS) protocol version 1.3. RFC 8446, RFC Editor, August 2018. IETF.org.

[23] Schneier, B. Attacking Tor: how the NSA targets users' online anonymity.

[24] Schneier, B., Fredrikson, M., et al. Surreptitiously weakening cryptographic systems. *International Association for Cryptologic Research* (February 2015).

[25] Stallings, W., and Brown, L. *Computer Security: Principles and Practice*, 3rd ed. Pearson Education, Inc., 2015.

[26] Swartz, A. Squaring the triangle: Secure, decentralized, human-readable names. January 2011.

[27] Unger, N., Dechand, S., et al. SoK: Secure messaging. *IEEE Symposium on Security and Privacy* (2015).

[28] WhatsApp. End-to-end encryption. *WhatsApp Security and Privacy*.

[29] Whittaker, Z. GCHQ's not-so-smart idea to spy on encrypted messaging apps is branded 'absolute madness'. *TechCrunch*.

[30] Whittaker, Z. Telegram told to give encryption keys to Russian authorities. *ZDNet*.

[31] Wilcox-O'Hearn, Z. Names: Distributed, secure, human-readable: Choose two. Zooko.com, October 2001.

[32] Wood, Dr. G. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Ethereum, February 2018.

A The Beacon Ethereum Contract

This section provides the source code of the public Ethereum contract for the Beacon secure messaging platform. The contract allows users to register their 64-byte public key associated with a unique username. By allowing a username to be a 16-length string, we enable a massive amount of username permutations; of course, early adopters get the benefit of choosing simple, unique usernames, but the necessary complexity still grows relatively slowly. Twitter, a service with almost 350 million *active* users, still hasn't run out of usernames with a 15-character limit.

```
pragma solidity ^0.4.24;

contract BeaconProtocol {
  struct User {
    address owner;
    bytes32[2] publicKey;
    bool exists;
  }

  mapping(bytes16 => User) private registeredUsers;
  address private creator;

  constructor() public {
    creator = msg.sender;
  }

  function register(bytes16 username, bytes32[2] publicKey) public {
    // Ensure that nobody has previously registered this username.
    require(!registeredUsers[username].exists);

    // Filter the username to be non-empty and have all characters between
    // either 0-9 or `[` and `z`.
    for (uint8 i = 0; i < 16; ++i) {
      bytes1 c = username[i];
      if (c == 0x00) {
        require(i > 0); break;
      } else if (c > 0x5a) {
        require(c < 0x7b);
      } else {
        require(c > 0x2f && c < 0x3b);
      }
    }

    registeredUsers[username] = User(msg.sender, publicKey, true);
  }

  function updatePublicKey(bytes16 username, bytes32[2] publicKey) public {
    require(registeredUsers[username].exists &&
            registeredUsers[username].owner == msg.sender);

    registeredUsers[username].publicKey = publicKey;
  }

  function deleteUser(bytes16 username) public {
    // Deletion occurs primarily by the admin in order to limit the wallet
    // complexity required for users, but is also possible by the username
    // owner in more advanced use-cases.
    require(msg.sender == creator ||
            registeredUsers[username].owner == msg.sender);

    registeredUsers[username].exists = false;
    delete registeredUsers[username];
  }

  function lookupUser(bytes16 username) public constant returns (bytes32[2]) {
    return registeredUsers[username].publicKey;
  }

```

```
59    function kill() public {
60      if (msg.sender == creator) {
61        selfdestruct(creator);
62      }
63    }
64  }
```

The contract is verifiable by using the same compiler as the original code. In this case, the contract was compiled using Arch Linux's `solidity` package with the `--optimize` flag:

```
$ solc --version
solc, the solidity compiler commandline interface
Version: 0.4.25+commit.59dbf8f1.Linux.g++
```

This contract is on the Ethereum blockchain in this precise transaction, costing 397,126 gas to create. The raw, optimized bytecode of the contract is

```
608060405234801561001057600080fd5b506001805460160a060020a0319163317905561040e180610032600039600f
3006080604052600436106100c5763ffffffff7c0100000000000000000000000000000000000000000000000000000000
0060003504166333c787118114610071578063341c0e1b514610098578063d3d3eaa146100ad578063bf0b287a146100e
1578063f4bed68314610106575b600080fd5b34801561007d57600080fd5b506100966001608060020a03196004351660
2435610128565b005b3480156100a457600080fd5b506100966103305650b3480156100b957600080fd5b506100cf60016
08060020a03196004351661036d565b604080519182525190819003602001905f35b3480156100ed57600080fd5b506100
966001608060020a03196004351660243561038c565b34801561011257600080fd5b506100966001608060020a0319600
43516610411565b6001608060020a03198216600090815260208190526040812060020154819060ff1615610154576000
80fd5b600091505b60108260ff1610156102a4578360ff8316601081106101745fe5b1a7f0100000000000000000000000
0000000000000000000000000000000000000000000002905060016f860020a0319811615156101bf57600360ff83161161
01ba57600080fd5b6102a4565b7f5a000000000000000000000000000000000000000000000000000000000000000060016
0f860020a03198216111561022c577f7b000000000000000000000000000000000000000000000000000000000000000060
0160f860020a0319821610610227576000080fd5b610299565b7f2f000000000000000000000000000000000000000000000000000000
0000000000000000600160f860020a03198216118015610028e57507f3b000000000000000000000000000000000000000
0000000000000000000000000600160f860020a03198216105b1515610299576000080fd5b816001019150610159565b5
050604080516060810182523381526020808201938452600182840181815260016080600020a0319969096166000908152
9182905292902090518154737fffffffffffffffffffffffffffffffffffffff191673ffffffffffffffffffffffffffffffff
ffffffffffff9091161781559151908201559051600290910180546ff191691151591909117905556560015473ffff
ffffffffffffffffffffffffffffffffffff1633141561036b5760015473ffffffffffffffffffffffffffffffffffffffff
fff16ff5b565b6001608060020a0319166000908152602081905260409020600101549056560015b6001608060020a03198216
6000908152602081905260409020600201546ff1680156103e657506001608060020a0319821660009081526020819052
6040902054737fffffffffffffffffffffffffffffffffffffffffffffff1633145b15156103f157600080fd5b6001608060020a
0319909116600090815260208190526040902060010155565b60015473ffffffffffffffffffffffffffffffffffffffff
f1633148061046557506001608060020a0319811660009081526020819052604090205473ffffffffffffffffffffffff
ffffffffffffffff1633145b1515610470576000080fd5b6001608060020a0319166000908152602081905260408120600
28101805460ff1916905580547377ffffffffffffffffffffffffffffffffffffffffffffffff191681556001015556600a165627a7a
7230582004426eff9cbfe93c54a1e4755970e8526fa183580c73201308eb68ac2513f3d60029
```

Thus, the global, universal, hard-coded contract address that contains the public repository of users and their associated public key is at: **`0x325CE54bBcDcD46BaeA885725E8030e4026d23C1`**. This value should be hard-coded into any client using the Beacon messaging platform.

Case-Insensitivity

Case-sensitive usernames introduce a significant impersonation risk (i.e. receiving a message from `aIice` versus `alice`). Filtering must be done on the contract to ensure that only lowercase characters are allowed. This filtering slightly raises the registration cost (by approximately 1,000 gas) but is much safer for users. We only allow this subset of characters:

```
0123456789:[\]^_`abcdefghijklmnopqrstuvwxyz
```

This subset was chosen for its computational efficiency. The goal is to allow as many non-conflicting characters as possible. We choose lowercase letters for readability, corresponding to the range `0x61-0x7a` on the ASCII table. Additionally, we wanted to enable numbers, so `0x30-0x39`. To minimize registration gas costs, only symbols that extend from either one of these ranges are allowed. To avoid bracket confusion between `{}` and `[]`, we choose `[]` because that also lets us include the critical _ character. This extends the letter range to `0x5b-0x39` to include: `[\]^_`. Finally, we can also extend the number range to include

the colon (`0x3a`), but no further because that may enable confusion with semicolons. Thus the final ranges in the Ethereum contract are established.

Registration Cost

The main (meaningful) function in the contract is `register`. Its execution cost depends on the length of the username; the maximum length is 16 characters, which corresponds to the following approximate calculation, given a Usd-Eth exchange rate of $400:

$$\begin{aligned}\text{cost} &\approx 113000 \text{ gas} \cdot \left(\frac{23 \text{ gwei}}{\text{gas}}\right) \cdot \left(\frac{1 \text{ ether}}{1 \times 10^9 \text{ gwei}}\right) \cdot \left(\frac{\$150}{1 \text{ ether}}\right) \\ &= 0.002599 \text{ ether} \cdot \left(\frac{\$150}{1 \text{ ether}}\right) \\ &= \mathbf{\$1.04}\end{aligned}$$

This is the raw cost to register a new user within a minute at the current Ethereum exchange rate. To offset for transaction fees between wallets, payment platform service fees, exchange rate fluctuations, and server costs to run the messaging platform, Beacon has an official cost of $3 to register most usernames.

Key Update Cost

Following a key compromise, or a regular key rotation strategy, users can update their identity key by calling the `updatePublicKey` method. Its execution cost, like the Registration Cost, varies based on the size of the username, but it is generally cheaper because it replaces the existing stored key, rather than resulting in a new storage operation. Updating the public key of a max-length username only costs approximately 38,500 gas, or **$0.35** by the above calculation.