Privacy on Solana with Elusiv and Light: All You Need to Know About Privacy on SOL

Prerequisites

- Node.js installed (to install the required dependencies, run the scripts, and send $SOL privately)
- Basic understanding of TypeScript
- Basic understanding of @solana/web3.js

What's this Article About?

Privacy is at the core of modern blockchain technology. Public ledger systems, transactions, account balances, and, oftentimes, a program's source code is made open for anyone to see. While this unprecedented level of transparency is championed as a hallmark of blockchain technology, privacy can be a vital requirement for many applications or everyday use.

Both Elusiv and the Light Protocol offer cutting-edge solutions for privacy on Solana by leveraging zk-SNARKS for private transactions, private programs, and decentralized compliance solutions. In this article, we delve into the importance of privacy, explore why blockchains such as Solana need privacy features, and provide a step-by-step guide on how to send SOL privately using Elusiv's TypeScript SDK and Light's zk.js module.

Privacy is Important

Privacy is a fundamental human right essential to autonomy and the preservation of human dignity. Privacy is necessary for a free and open society as it underpins the exercise of other rights, such as freedom of speech and freedom of association. This allows for the creation of a space separate from political life, ensuring that anyone can think, speak, express ideas, or behave without being constantly monitored by a third-party. Recent public controversies such as Edward Snowden's public disclosures, the information exposed by WikiLeaks, and the Facebook-Cambridge Analytica data scandal have ignited intense debates over the importance and boundaries of privacy.

Everyone Should Care About Privacy

At first glance, it may seem counterintuitive to advocate that public ledgers, built on top of decentralized technologies that champion transparency as one of its fundamental tenets, should have privacy-enabling features. Whether you're a staunch cypherpunk or an average user, everyone

should care about privacy. You might think that since you aren't doing anything harmful or illegal online, then who cares about seeing what I do on the blockchain. Consider this thought experiment:

We have two characters: Alice, an average user who values her privacy and wants to make purchases without scrutiny; and a malicious actor, which I will abbreviate as MA. A malicious actor could be an oppressive government, a controlling employer, or a cybercriminal, for example. Say Alice lives in a country where the government surveils each citizen constantly, or Alice works for a company with strict policies against certain legal purchases (e.g., political donations, buying books on certain subjects). One day, Alice makes a purchase using a traditional blockchain, such as Solana, believing it to be a secure and private method of payment. The sequence of events is as follows:

- Alice's transaction is recorded on the public

ledger. Her transaction includes her public key, the receiver's public key, and the amount sent

- MA, wanting to monitor or control Alice's behavior, starts tracing her transactions using her public key
- MA discovers Alice's purchase, which they deem "objectionable" under their rules
- MA takes action against Alice, which, depending on the circumstances, could range from legal action, imprisonment, workplace sanctions, social shaming, or blackmail

This thought experiment illustrates the stark reality of how a lack of privacy on the blockchain could affect the average user. With the rise of on-chain sleuths and malicious data-harvesting technologies, it is becoming quite trivial to trace the average user's transactions, which can lead to substantial real-world consequences. In an increasingly digital world, without privacy we risk the erosion of individual sovereignty, having all our decisions be monitored, and have every aspect of our existence commodified. Blockchains will only accelerate this if we do not implement privacy-enabling technologies. This is where Elusiv and the Light Protocol come in.

What is Elusiv?

Elusiv describes itself as "a blockchain-based, zero-knowledge privacy protocol" that "enables users and applications to access universal encryption". Elusiv seeks to achieve privacy without sacrificing security, safety, and decentralization by applying zero-knowledge cryptography and multi-party computation to the user and network. This goal of universal encryption allows Elusiv to

provide tools that enable users to control their data. Now, users can choose what gets shared on the blockchain and what doesn't.

How Does Elusiv Work?

Elusiv works by having users send funds to a shared pool governed by the Elusiv program, which tops up their private balances. From the pool, users can send and receive tokens such as SOL, as well as withdraw funds. It is important to note that with this shared pool model, people can still observe deposits to and payments from Elusiv. With more users and transactions, however, it becomes increasingly difficult to link deposits and payments to specific accounts over time - it's easier to link a lone user that puts in 10 SOL and takes out 10 SOL than it is to link a user among millions that makes and takes incremental deposits and payments. Anonymity loves company.

Funds in the pool do not contain any information about their depositors, but,using zero knowledge proofs, secret values are associated with a user's funds so proofs of authority can be made over unspent funds. The zero knowledge proofs used in Elusiv are known as zk-SNARKS (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge). Without delving into the complexities of zk-SNARK proofs, think of them as a mathematically proven way to show your friend that you know a secret password without ever telling them what it is. Here, your friend is able to quickly check that you're telling the truth without having to ask you anything else. Thus, Elusiv can keep track of your unspent funds without telling all of Solana how much you have in the pool. If you're keen on learning more about Zero-Knowledge Proofs, I urge you to check out Porter's class on Zero-Knowledge Proofs.

What is the Light Protocol?

Light Protocol describes itself as a "[n]ext-gen zkLayer on Solana" that enables private program execution directly on Solana. There are three concepts fundamental to Light that enables privacy on Solana:

- Private state
- UTXO model
- Verifying private state transitions on-chain

How Does the Light Protocol Work?

Light Protocol allows on-chain state to be encrypted. Here, users "own" this state and have exclusive decryption rights. Unlike Elusiv's pooled fund approach, Light Protocol introduces a UTXO (Unspent Transaction Outputs) model to Solana. A UTXO is tied to a shielded keypair and can hold two balances, one in SOL (since it is the fee token in Light Protocol) and one in another SPL token. Under this approach, each transaction generates a new UTXO that is appended to a linear list thereby avoiding in-place updates; state is concealed as opposed to leaking information about which state is being updated and by whom.

Program logic on Solana is almost always an escrow with some unique constraints. Light Protocol innovates upon this design by encoding program logic into zero-knowledge circuits (zk-SNARKS) on the client side where a computation and proof is generated. This proof is shared on-chain where a dedicated verifier program, crafted specifically for the circuit, checks its validity. Upon successful verification, the custom verifier Program calls another native verifier that completes various security checks and adds the new commitments to Light's Merkle tree program.

Albeit out of the scope of this article, Light allows for the creation of your own custom Private Solana Program (PSP) from scratch. It is recommended that you read the rest of this article and check out their documentation before you attempt to create your own. Their documentation includes a tutorial on how to create your own custom PSP, which can be found here.

This is cool and all, but how can I apply this to programming on Solana directly?

Sending SOL Privately using Elusiv

The Full Code

Breaking Down the Elusiv Code

First, we import the noble-ed25519 library as we do not have any wallet connectivity set up. In a production setting, you would use the Solana Wallet Adapter instead to sign transactions. Here, we only need the sign

functionality, however, using import { sign } from "@noble/ed25519";

may sometimes return an error that etc.sha512Sync

is not set. In order to avoid this error, we set it ourselves - don't worry too much about this.

We then import Solana's web3.js and Elusiv's TypeScript SDK with the necessary modules.

Here, we define CLUSTER

as "devnet"

and set our RPC to the Helius devnet RPC. If you wanted to use this script on mainnet, you would assign CLUSTER

as "mainnet"

and get a Helius API key from one of the following plans 😎

We mark our main

function as asynchronous since we await

several functions inside its block. We then establish our connection to devnet using RPC

, and create a Keypair

object from the utility function generateKeypair

. We also create a recipientPublicKey

with the aforementioned generateKeypair

function so we can receive the shielded SOL at a random address. For reference, the generateKeypair

function is defined as follows:

This function creates a test Solana keypair that we will use for this script. If you wanted to use this on mainnet, you would replace the keypair with your own wallet keypair. Remember, to never

store your private key in plain text and to import it properly using the correct file path.

We need some devnet SOL since we are using newly generated test keypairs. We use the aidropSol

utility function to airdrop SOL to our new keypair, which is defined as follows:

Here, we airdrop 1 SOL (1 * LAMPORTS_PER_SOL

) to ourselves on devnet. We retrieve the latestBlockHash

so we can pass the blockhash and the last valid block height into connection.confirmTransaction()

thereby confirming that we've airdropped 1 SOL to the correct address.

Elusiv requires an input seed to ensure certain operations are deterministic, which is why we import and use SEED_MESSAGE

. SEED_MESSAGE is a required message to sign as it is used to generate the Elusiv seed; it allows the application to decrypt your private assets with their associated secret values so you can keep track your funds. We use Buffer.from()

to create a new buffer (a raw binary data structure) from the SEED_MESSAGE

with utf-8 encoding. We need to use slice

since Solana's keypair type has it so the first 32 bytes is the private key and the last 32 bytes is the public key. Thus, we sign the required seed message using our private key.

Here, we define elusiv

, which is an instance of Elusiv, using our seed

, keyPair.publicKey

, connection

, and CLUSTER

. Thus, we're creating an Elusiv instance on devnet using the devnet RPC, our public key, and the long message we had just signed.

We then fetch our private balance - the amount of Lamports we have in the Elusiv pool - which is returned as a BigInt. The rest of the main

function is encapsulated in a try-catch-finally block

, which is defined as follows:

If we have a private balance, balance

will be greater than 0 and the first block of the if-else

block will execute. The reason we cast 0 as BigInt

in the if

condition is because getLatestPrivateBalance

's return type is BigInt

. After, we create signature

, a variable set to the return value of send()

. The send()

function is defined as follows:

This function sends our transaction via Elusiv and returns a Promise, which is of type ConfirmedSignatureInfo

(a confirmed signature with its status). As our arguments we take an Elusiv instance, a recipient's public key, an amount to be sent, and the token type (which does not have to be Lamports and could be a number of other tokens such as USDC, USDT, mSOL, BONK, and SAMO). We then set txt

using elusiv.buildSendTx()

. buildSendTx()

is a method on the Elusiv instance that builds a transaction to send tokens to a specified recipient using the private balance. We then send off and return the transaction using our return elusiv.sendElusivTx(txt)

statement, passing in our built transaction txt

.

If we do not have a private balance, the else

block executes:

We log to the user that they do not have a private balance and that we are going to top them up. We then build the top up transaction using the .buildTopUpTx()

method found on the Elusiv instance. We pass in a value of 1 SOL and a token type of Lamports since we are sending SOL. Then we sign the top up with a .partialSign()

on the transaction's keypair since we are topping up from our public key. We then send off the built transaction using elusiv.sendElusivTx(topUpTxData)

and log the transaction signature.

The remainder of the try-catch-finally

block is as follows:

If we have any errors sending the SOL via Elusiv, we log them to the console using console.error()

. Then, we exit the program using process.exit(0)

.

To initiate this entire process, we call the main()

function at the bottom of the file.

Sending SOL Privately using Light Protocol

The Full Code

Breaking Down the Code

First, we import the necessary modules from Anchor, Solana/web3.js, and Light to interact with Solana using the Light Protocol. We then proceed to define a number of utility functions before getting to our main

function - Light has a more complex initialization process compared to our Elusiv implementation.

Here, we create the function initializeSolanaWallet

, which generates a new Solana wallet using the Keypair.generate()

method.

Unlike Elusiv, Light provides an airdropSol

function in their library to airdrop Lamports to a specified public key. We create the utility function requestAirdrop

to send SOL to the publicKey

passed.

In order to send shielded SOL via Light, we need to set up a test relayer using the TestRelayer

module. Relayers play a significant role in the Light ecosystem as they are web servers that forward the ZK proofs, which prove state transitions, from the client to the respective on-chain program. They also pay for the transaction fees in return for a fee at successful proof verification. The fees are 100k Lamports, which we set with new BN(100_000).

We then initialize the Light Provider with our aptly named initializeLightProvider()

function. This function sets up a provider and links it with our Solana wallet and the test relayer using the configuration provided by Light, confirmConfig

- it is of type ConfirmOptions

and deals with the transaction verification step, commitment levels, etc.

The initializeLightUser

function is pretty self-explanatory - it initializes a user for Light Protocol using the lightProvider

passed in.

The performShieldOperation

function allows the user to shield SOL, making it private. Here, we are shielding more than 1 SOL, although we are only sending 1 shielded SOL, in order to account for fees.

This is arguably the most important utility function as it actually executes a private transfer. Here, we privately transfer 1 SOL from user

to the specified recipient testRecipientPublicKey

.

The main

function serves as the entry point for our script. We mark it as asynchronous as there are a number of async functions called within its scope. It runs all the steps in sequence, starting off with the wallet initialization and ending with executing a private transfer. First we set up our Solana wallet and connection. Then we airdrop SOL to our wallet and initialize the relayer and Light Provider. We then initialize our wallet as a Light User and shield our SOL. We then create a test recipient, airdrop them some SOL, initialize them as a Light User, and send them 1 SOL privately. Like our Elusiv code, we put the logic into a try-catch-finally

block in order to catch any errors and automatically exit the program once completed.

You may notice that, unlike Elusiv where we were using devnet, here we are setting the connection to "http://127.0.0.1:8899". At the time of writing this article, it is important to note that Light Protocol is not deployed on devnet/mainnet so Helius keys wont work (yet). For localhost, you'll want to run a test-validator with the necessary preloaded accounts and programs in order to run your code. You can install the Light Protocol CLI for this and run the following command:

This will start a validator in the background. Currently, you will also need to add chai (an assertion library used to make testing easier) manually since there are some unit tests inside the UTXO class. This, however, should be resolved in the next release. Your package.json

for this script should look something like this:

Conclusion

Congratulations! In this tutorial, we delved into the intricacies of privacy, its importance, and how to transact SOL privately using Elusiv and the Light Protocol. With Elusiv, we saw how to retrieve private balances, send SOL privately, and even facilitated automated top-ups in the event that the private balance is depleted or does not exist. With the Light Protocol, we saw how to set up its complex, highly modular initialization process, shield our SOL, and execute a private transfer.

So, you may be wondering which one you should use? Well, it depends! With Elusiv, our code focused on topping up a private balance from a shared pool and sending transactions privately. The script interacts with a predefined smart contract that governs the Elusiv pool to achieve this. With Light Protocol, we established a more flexible, albeit complex, initialization process to create a workflow that manages multiple users interacting with privacy-preserving Solana programs. With Light we can also create our own custom Private Solana Programs (PSPs), if needed. Both have their tradeoffs with respect to areas such as documentation and specific implementation, however both protocols aim for privacy and secure transactions. If your application requires a more granular control over privacy features and you want to encrypt specific parts of the application state, then consider using the Light Protocol. Or, if you're building something simpler and want to add a layer of privacy for token transfers on devnet/mainnet, Elusiv could be the better option.

The techniques, code, and insights found in this article offer a robust starting point for building privacy-focused applications on

Solana. The significance of privacy in blockchain technology cannot be overstated. As the demand for decentralized systems grows, so does the need for privacy-enabling features that empower users, protect their rights, and bring to life the true aims of early blockchain technology. Privacy matters, and now with both Elusiv and the Light Protocol you can make Solana a safer blockchain!

If you've read this far anon, thank you!

Additional Resources / Further Reading

- A Cypherpunk's Manifesto
- Elusiv
- Elusiv Docs
- Elusiv SDK
- Elusiv Samples
- Light Protocol
- zk.js
- Private Solana Program Example Implementations