Signata

Release 0.1

Congruent Labs

Jun 25, 2022

CONTENTS

Signata is a Decentralized Identity project for smart contract-based blockchains.

- See *Chains* for implementation-specific differences between blockchains that Signata is deployed on.
- See *Contracts* for smart contract addresses.
- See *DAO* for details on governance for the project.
- See *Design* for architecture and design documentation.
- See *Examples* for examples on how to use Signata identities.
- See *Identities* for details on how identity lifecycle is managed on the chain.
- See *Integrations* for implementation details on integrations with the Signata platform.
- See *Links* for links to development communities and other connected services.
- See *Risk* for information on the Risk Oracle solution.
- See *Veriswap* for information about the Veriswap service.

Note: This project is under active development and smart contracts may be changed at any time.

CHAPTER

ONE

CONTENTS

1.1 Chains

1.2 Contracts

This document details all contract addresses for deployed contracts on each network in the sections below.

Use the relevant block explorer for the network to examine the contract deployment details.

Testnet contracts are not yet listed here. Most test contracts will only be deployed on the Ethereum testnets.

Note: These contracts are still in development and may change at a later date. Use this site to check the latest contract addresses for any integrations you are developing.

1.2.1 Ethereum Mainnet

	Address
SATA Token	`0x3ebb4A4e91Ad83BE51F8d596533818b246F4bEe1`
dSATA Token	`0x49428f057dd9d20a8e4c6873e98afd8cd7146e3b`
Governor	`0x3D3255D21654B9a8325DfE6353ac6B37352Eb80B`
Timelock	`0x30b0106d9140902d7d495a7f21d282852e9f59d8`
SignataIdentity	`0x6B47e26A52a9B5B467b98142E382c081eA97B0fc`
SignataRight	`0x7c8890a02abd24ff00c4eb1425258ea4b611d300`
Veriswap	`TBC`
Deployer	`0x042fc4ea3f836e1ea5dc4fb70ec90ded51c09eca`
Token Vault	`0xc77aab3c6d7dab46248f3cc3033c856171878bd5`

1.2.2 Ethereum Rinkeby

These addresses may change at any time.

	Address
SignataIdentity	0xB24e28A4B7fED6d59D3BD06af586f02fDdfa6385
AtomicSwap	0x2dc3cd99b4e31805156886834e233903ff394032

1.2.3 Binance Smart Chain Mainnet

	Address
SATA Token	0x6b1C8765C7EFf0b60706b0ae489EB9bb9667465A
SignataIdentity	0x2EE533c160eB0488f2Db54DA4523b110E66a96Ff
SignataRight	0x55E9B159FF6E6cD9188Cc4394db3EC944d467B0d
Veriswap	TBC
Deployer	0xEAD57fE351280E7D905a5E21e9fd7d5bb0Ec97e7
Liquidity Vault	0x50e63fe25e8cf75c948a1ba26021ea0883cdc5b3

1.2.4 Avalanche C-Chain

	Address
SATA Token	0xbec0A9aEa58b6a0c0f05a03078f7E7Dcecc13A95
SignataIdentity	0x3ebb4A4e91Ad83BE51F8d596533818b246F4bEe1
SignataRight	0x0BaFDe3aDAd83b679FAE5E9793Cd44ab247c6096
Veriswap	TBC
Deployer	0x042fc4ea3f836e1ea5dc4fb70ec90ded51c09eca
Liquidity Vault	TBC

1.2.5 LGCY Network

	Address
SATA Token	TBC
SignataIdentity	TBC
SignataRight	TBC
Veriswap	TBC
Deployer	TBC
Liquidity Vault	TBC

1.2.6 Metis Stardust Network (Testnet)

	Address
SATA Token	`0x3ebb4A4e91Ad83BE51F8d596533818b246F4bEe1`
SignataIdentity	`0x545f8952A5cADF63DeE9658C189B309FAd5d789f`
SignataRight	`0xe1A0f23b29fba2c82d8af15D354aaE048AE1Cd13`
Veriswap	TBC
Deployer	`0x042fc4ea3f836e1ea5dc4fb70ec90ded51c09eca`
Liquidity Vault	TBC

1.2.7 Metis Andromeda Network (Mainnet)

	Address
SATA Token	`0x3ebb4A4e91Ad83BE51F8d596533818b246F4bEe1`
SignataIdentity	`0x545f8952A5cADF63DeE9658C189B309FAd5d789f`
SignataRight	`0xe1A0f23b29fba2c82d8af15D354aaE048AE1Cd13`
Veriswap	TBC
Deployer	`0x042fc4ea3f836e1ea5dc4fb70ec90ded51c09eca`
Liquidity Vault	TBC

1.2.8 Fantom Mainnet

	Address
SATA Token	`0x3ebb4A4e91Ad83BE51F8d596533818b246F4bEe1`
SignataIdentity	`0x545f8952A5cADF63DeE9658C189B309FAd5d789f`
SignataRight	`0xe1A0f23b29fba2c82d8af15D354aaE048AE1Cd13`
Veriswap	TBC
Deployer	`0x042fc4ea3f836e1ea5dc4fb70ec90ded51c09eca`
Liquidity Vault	TBC

1.3 DAO

Signata Governance is currently managed on GitHub using a dedicated repository using Issues to track each proposal.

1.3.1 Management

All Signata DAO proposals are managed on GitHub.

GitHub

1.3.2 Interface

All Signata DAO contracts are implementations of OpenZeppelin standards, and so Tally has been used for DAO management.

Tally

1.3.3 Timelock

Derived from OpenZeppelin Contracts (last updated v4.6.0) (governance/TimelockController.sol)

Verified Etherscan Contract

1.3.4 Governor

Derived from OpenZeppelin Contracts (last updated v4.6.0) (governance/Governor.sol)

Implements Governor, GovernorSettings, GovernorCompatibilityBravo, GovernorVotes, GovernorVotesQuorumFraction, and GovernorTimelockControl.

Verified Etherscan Contract

1.4 Design

Warning: This is a living document, and can change at any time.

1.4.1 Abstract

Developers of online services spend countless hours of effort in the construction of systems to collect private information to identify users, and even more effort in maintenance of these systems to remain compliant with local and international laws and regulations. Taking payment for services only adds to this burden, as organizations must pay and rescind control to 3rd party service providers to deliver secure and fraud-resistant services for the collection and management of payment information.

In this document we discuss the Signata service as a means to bind the next generation of identities: identities that are kept anonymous from service providers, and we present IdGAF - the Identity Guard & Anonymity Framework as a decentralized on-and-off-chain solution for the identification, authorization, and lifecycle management for modern identity. We will present the ability for users to self-assert identities onto chains via smart contracts, as well as the ability for service providers to validate and maintain known anonymous identities via off-chain solutions.

We will discuss the capabilities already in place with Signata for the management of hardware wallets and interactions with blockchains, the next phases of the product to incorporate the IdGAF as a full identity and payment platform, to introduce the new SATA token to back these systems, and how these services will provide the first proof of concept for the independent integration of other services using this framework.

1.4.2 Introduction

The online identity management world is in a constant state of flux. Centralized identity providers (such as Google, Facebook, and Okta) are attempting to lead the charge with centralized authentication services for simplified management, and the era of the password is looking to quickly become obsolete. However, centralized identity management requires users to rescind all control of their identity and access to the service providers that manage their identities, instead of retaining the control over their individual authentication capabilities and identity assertions. These centralized providers typically fund themselves by building unprecedented tracking data on individuals, observing the use of identities within their services and outside through an ever-growing network of online tracking systems.

Signata is a platform built by Congruent Labs to reveal the true smartcard capabilities of Yubico YubiKeys, bridge individuals' identities to their digital content, and to interact with blockchains. The core capability of Signata is currently to deliver a hardware-based wallet for cryptocurrency storage, but the technologies that underpin YubiKeys also provide the ability to authenticate, digitally sign content, and bind identities to factors of authentication.

Signata's use of well-established smartcard capabilities with YubiKeys drives a natural path to expansion of the Signata service to integrate more functionalities such as authentication and digital signatures, but also for the expansion into more integration of users identities and authentication systems onto blockchains instead of just interacting with them.

This document proposes to introduce a new ERC-20 token for Signata called SATA. This token will serve a number of purposes. In future releases of the platform the SATA tokens will be used to interact with a platform of smart contract-based decentralized identity services that Signata is currently developing - both as core internal capabilities for the product, but additionally as on-and-off-chain anonymity preserving systems that external applications can integrate and consume to build an identity ecosystem unbound by central authorities. This new platform will be known as the Identity Guard & Anonymity Framework (IdGAF).

We believe existing capabilities of identity management on blockchains is trying to realise protocols and systems that were designed against non-blockchain systems - Signata will instead deliver a platform that maintains the core tenants of blockchains, being:

- Anonymous but cryptographically trusted identification of individuals,
- Decentralized assertion of content, and
- Secured payment for services or interactions on the chain.

Service Providers using IDGAF, including Signata as the first proof of concept, will be able to securely authenticate and authorize users with anonymous credentials by combining on-chain verification of data produced by the credential holder (assuring ownership of the credential), on-chain verification of authorizations, and off-chain verification of information held within the service itself (ensuring the credential receives appropriate authorization).

This capability will allow for service providers to authenticate users, collect payments, and provide access control to systems without knowing any identifiable information about the user - unless they want to collect that information themselves and the user consents to the collection of the information.

1.4.3 Product Description

IdGAF (Identity Guard & Anonymity Framework)

The Identity Guard & Anonymity Framework will be delivered as a set of on-chain contracts and off-chain systems to deliver a fully-capable authentication service for applications. This framework will deliver a number of key subsystems, each bound or related to the cryptographic capabilities of blockchain addresses, records, and interactions.

As each of these systems is built and released, they will be delivered within an open identity marketplace, although the open source nature of these components will not be constrained to exclusive access via this marketplace.

Self-Asserted Identity Authorities

Each individual will establish an anchor credential within their chosen device. This anchor credential will be retained to approve the binding of addresses added or imported from other systems, providing users the capability to self-assert approval of cryptographic material for use with authentication and authorization.

Users are not restricted in the issuance of their own authority credentials, nor are they limited in the number of credentials issued by their authorities, so that users can adapt their identities to the specific contexts that they are asserting them within.

Users may, in normal operations, lose access to or have their identity authorities compromised. In the event of an identified compromise, the individual can either self-assert the cancellation of their own identity authority (assuming they still retain control over it), or replace their identity authority with a new authority (and undertake re-assertion of the new authority to connected providers).

Identity authorities ultimately introduce the largest vector of attack from external parties - compromise the authority and one can deny service or assume the identity of the stolen authority. One of the mitigations for this attack vector will be the enforcement of hardware-based key storage will be essential to the manner in which users interact with the IdGAF, much akin to how the Universal 2nd Factor (U2F) provides hardware-based protection to the use of authentication credentials. Not all authentication systems can interact with hardware devices (including many mobile devices limited by physical interfaces and operating system policies), and so a credential delegation capability will also be introduced to facilitate the creation of credentials issued with constrained capabilities to ensure that users can still access systems they need without exposing credentials to undue risk within lower-assurance devices.

Anonymized Identity Providers (DeREx)

In the current Identity Provider market, most providers offer the combination of some form of persistent identity collection solution, single sign-on capabilities, session management across services, and (for more advanced integrations) adaptive risk solutions for observing unexpected user behaviour.

Connected IdGAF service providers will instead deliver the core capability of persisting identities but retaining anonymity, as well as offering the ability for the capture and management of payment for services directly linked to the identity provider. With this integrated approach system developers no longer need to integrate two disparate systems to achieve the same overall outcome for their products - users can authenticate securely and pay for services within the same set of transactions, and without needing to surrender personally identifiable information to the service provider.

Service providers can additionally relieve themselves of the responsibility to capture and store identity and payment information, removing the potential exposure of identifiable information once a system has experienced a data breach or leak.

Connected providers will be presented as the Decentralized Rights Exchange (DeREx), providing a unified platform for 3rd parties to integrate and consume these services.

Decentralized X.509 (Dex509)

Public Key Infrastructure (PKI) systems have been built and naturally evolved to suit operation on blockchains. Considering the core capabilities of certificate authorities, the security controls imposed to protect them are designed to effectively replicate the features that blockchains now inherently offer - they store an immutable sequence of events much like the individual blocks and transactions managed on chains.

IdGAF-enabled services that interact with authentication, signing, and encryption certificates will be able to additionally push and pull certificate records into the chains. Assertions of authority/signing status of public keys will permit service providers to inherently trust assertions made by specified authorities as a transitive, but still anonymous, trust model similar to trust models within the PKI ecosystem.

1.4.4 Disclaimer

The plans, strategies, and implementation details described in this whitepaper will likely evolve and, accordingly, may never be adopted. Congruent Labs Pty Ltd reserves the right to develop or pursue additional or alternative plans, strategies, or implementation details associated with the Signata platform.

SATA tokens are being distributed by Congruent Labs Pty Ltd pursuant to the Terms and Conditions (the "terms") of the token available at https://sata.technology/. For complete details, review the terms. SATA tokens are not securities, investments, or currency, and are not sold or marketed as such. Participation in the collection of SATA tokens involves significant technological and systemic risks. The distribution of SATA tokens is not open to individuals who reside in or are citizens of the United States or Canada. The distribution period, duration, pricing, and other provisions may change as stated in the terms. SATA tokens do not in any way represent any shareholding, participation, right, title, or interest in Congruent Labs Pty Ltd, their respective affiliates, or any other company, enterprise, or undertaking, nor will SATA entitle token holders to any promise of fees, dividends, revenue, profits, or investment returns, and are not intended to constitute securities in Australia or any relevant jurisdiction.

The SATA token distribution involves known and unknown risks, uncertainties, and other factors that may cause the actual functionality, utility, or levels of use of SATA tokens to be materially different from any projected future results, use, functionality, or utility expressed or implied by Congruent Labs Pty Ltd in the terms.

1.5 Examples

Warning: Any source code provided on this site is unaudited and provides no guarantee that it will be functional or safe to use. Ensure you test before deploying to mainnet or production use.

1.5.1 Applying Signata Identities to Authentication Systems

Claims-based Enforcement

TODO

1.5.2 Applying Signata Identities to Smart Contracts

An identity by itself is of no value. Its value is derived from those that recognise it, and what systems it can be interconnected to.

This document details the application of Signata identities to blockchain-based systems. It does not prescribe a recommended or best-practice approach for integration, only a starting point that can be developed upon for real-world systems.

Integrating identity validation into smart contracts is extremely simple - connecting to an instance of the SignataIdentity and SignataRights contracts within a network will provide the basis of enforcing Signata-based identity enforcement into any contract function that you wish to use it in.

Simple Access Control

At a basic level, smart contracts can verify that an address has been registered with an identity contract.

For open identity contracts this provides next to no assurance as any address may register themselves as an identity and pass this check. But for an access-controlled identity contract, the provisioning of the identity can be restricted to specific accounts with a process of issuance having been performed by the contract owner.

```
SignataIdentity private signataIdentity;

/* ... */

// get delegate - will throw if the identity does not exist
signataIdentity.getDelegate(msg.sender);

// or check if locked - also will throw if the identity doesn't exist
// and will also reject locked identities
require(
    !signataIdentity.isLocked(msg.sender),
    "The sender's identity is locked."
);
```

Rights Based Access Control

To enforce stronger access control in contracts, validating that an address holds a specific ERC721 right is possible by checking against those that have been

At a basic level, smart contracts can verify that an address has been registered with an identity contract.

For open identity contracts this provides next to no assurance as any address may register themselves as an identity and pass this check. But for an access-controlled identity contract, the provisioning of the identity can be restricted to specific accounts with a process of issuance having been performed by the contract owner.

```
SignataRight private signataRight;
SignataIdentity private signataIdentity;
address private trustedMinter;

/* ... */

// check the minter of an asserted right
require (signataRight.minterOf(nftRightIndex) == trustedMinter, "Right is not from␣
↪trusted minter");

// get the delegate address of the sender
address delegateAddress = signataIdentity.getDelegate(msg.sender);

// check the owner of the asserted right
require (signataRight.ownerOf(nftRightIndex) == delegateAddress, "Right is not owned by␣
↪sender!");
```

With these three checks above, we've validated that the asserted right of the sender is actually owned by them via their identity delegate key, we've validated that the identity is actually currently active (getDelegate will fail if the identity is currently locked or destroyed), and we've validated that the minting of the right came from a trusted source.

Business Register

Most jurisdictions will manage a public registry of all operating businesses, tracking each with a unique identifier so they're identifiable separately to the individuals that own, operate, or work in those businesses.

TODO

Staking Pool

For staking pools, you can simply deny entry to the pool if the user does not hold a specific identity or right.

And for unstaking, adding the same checks can ensure that a mutated identity cannot claim the rewards if that is a desired outcome.

```
SignataIdentity private signataIdentity;

/* ... */

function stakeTokens(uint256 _amount, uint256[] memory _tokenIds) public {
  require(
    getLastStakableBlock() > block.number,
    'this farm is expired and no more stakers can be added'
  );

  // try to get the delegate of the account. this will fail if the account isn't␣
↪registered, or is in an unusable state
  signataIdentity.getDelegate(msg.sender);

  _updatePool();

  if (balanceOf(msg.sender) > 0) {
    _harvestTokens(msg.sender);
  }

  uint256 _finalAmountTransferred;
  if (pool.isStakedNft) {
    require(
      _tokenIds.length > 0,
      "you need to provide NFT token IDs you're staking"
    );
    for (uint256 _i = 0; _i < _tokenIds.length; _i++) {
      _stakedERC721.transferFrom(msg.sender, address(this), _tokenIds[_i]);
    }

    _finalAmountTransferred = _tokenIds.length;
  } else {
    uint256 _contractBalanceBefore = _stakedERC20.balanceOf(address(this));
    _stakedERC20.transferFrom(msg.sender, address(this), _amount);

    // in the event a token contract on transfer taxes, burns, etc. tokens
    // the contract might not get the entire amount that the user originally
    // transferred. Need to calculate from the previous contract balance
    // so we know how many were actually transferred.
    _finalAmountTransferred = _stakedERC20.balanceOf(address(this)).sub(
```

(continues on next page)

(continued from previous page)

```
      _contractBalanceBefore
    );
  }

  if (totalSupply() == 0) {
    pool.creationBlock = block.number;
    pool.lastRewardBlock = block.number;
  }
  _mint(msg.sender, _finalAmountTransferred);
  StakerInfo storage _staker = stakers[msg.sender];
  _staker.amountStaked = _staker.amountStaked.add(_finalAmountTransferred);
  _staker.blockOriginallyStaked = block.number;
  _staker.timeOriginallyStaked = block.timestamp;
  _staker.blockLastHarvested = block.number;
  _staker.rewardDebt = _staker.amountStaked.mul(pool.accERC20PerShare).div(
    1e36
  );
  for (uint256 _i = 0; _i < _tokenIds.length; _i++) {
    _staker.nftTokenIds.push(_tokenIds[_i]);
  }
  _updNumStaked(_finalAmountTransferred, 'add');
  emit Deposit(msg.sender, _finalAmountTransferred);
}
/* ... */
```

Token Lock

Token locks can be restricted to only accept locks from validated identities, before the lock is created.

```
SignataIdentity private signataIdentity;

/* ... */

function createLocker(
  address _tokenAddress,
  uint256 _amountOrTokenId,
  uint48 _end,
  uint8 _numberVests,
  address[] memory _withdrawableAddresses,
  bool _isNft
) external payable {
  require(
    _end > block.timestamp,
    'Locker end date must be after current time.'
  );

  // try to get the delegate of the account. this will fail if the account isn't␣
↪registered, or is in an unusable state
  signataIdentity.getDelegate(msg.sender);

  _payForService(0);
```

(continues on next page)

(continued from previous page)

```
  if (_isNft) {
    IERC721 _token = IERC721(_tokenAddress);
    _token.transferFrom(msg.sender, address(this), _amountOrTokenId);
  } else {
    IERC20 _token = IERC20(_tokenAddress);
    _token.transferFrom(msg.sender, address(this), _amountOrTokenId);
  }

  lockers.push(
    Locker({
      owner: msg.sender,
      isNft: _isNft,
      token: _tokenAddress,
      amountSupply: _isNft ? 1 : _amountOrTokenId,
      tokenId: _isNft ? _amountOrTokenId : 0,
      start: uint48(block.timestamp),
      end: _end,
      withdrawable: _withdrawableAddresses,
      amountWithdrawn: 0,
      numberVests: _isNft ? 1 : (_numberVests == 0 ? 1 : _numberVests)
    })
  );
  uint16 _newIdx = uint16(lockers.length - 1);
  lockersByOwner[msg.sender].push(_newIdx);
  lockersByToken[_tokenAddress].push(_newIdx);
  if (_withdrawableAddresses.length > 0) {
    for (uint16 _i = 0; _i < _withdrawableAddresses.length; _i++) {
      lockersByWithdrawable[_withdrawableAddresses[_i]].push(_newIdx);
    }
  }
  emit CreateLocker(msg.sender, _newIdx);
}
/* ... */
```

DAO Governor with Identity Validation

The simplest way to enforce identity usage within a DAO is to just validate the voter when they attempt to cast votes or delegate their voting rights.

```
/* override votes on the governor */
castVote(uint256 proposalId, uint8 support)
castVoteWithReason(uint256 proposalId, uint8 support, string reason)
castVoteWithReasonAndParams(uint256 proposalId, uint8 support, string reason, bytes␣
↪params)
castVoteBySig(uint256 proposalId, uint8 support, uint8 v, bytes32 r, bytes32 s)
castVoteWithReasonAndParamsBySig(uint256 proposalId, uint8 support, string reason, bytes␣
↪params, uint8 v, bytes32 r, bytes32 s)

/* override delegation on the token contract */
delegate(address delegatee)
```

(continues on next page)

(continued from previous page)

```
delegateBySig(address delegatee, uint256 nonce, uint256 expiry, uint8 v, bytes32 r,␣
↪bytes32 s)
```

Batch Airdrop

Note: this is not a common way to do airdrops, as batch ERC20 token transactions are expensive for the sender. If you do wish to distribute tokens in this manner, then you can skip any recipients that aren't registered identities.

```
SignataIdentity private signataIdentity;

/* ... */
function bulkSendErc20Tokens(
  address _tokenAddress,
  Receiver[] memory _addressesAndAmounts
) external payable returns (bool) {
  _payForService(0);

  IERC20 _token = IERC20(_tokenAddress);
  for (uint256 _i = 0; _i < _addressesAndAmounts.length; _i++) {

    Receiver memory _user = _addressesAndAmounts[_i];

    try signataIdentity.getDelegate(_user.userAddress) {
      _token.transferFrom(msg.sender, _user.userAddress, _user.amountOrTokenId);
    } catch { } // ignoring the index if the identity was not found
  }
  return true;
}
/* ... */
```

Merkle Airdrop

Merkle-based drops are far more cost effective for distributing tokens to a large number of holders. If you wish to only distribute tokens to created identities, or identities that hold a specific NFT right (such as a KYC claim), then you can simply inject a check into the claim function to prevent invalid identities from claiming the airdrop.

For a merkle-based drop it would be advisable to only create the proofs for addresses that are actually valid in the first place, performing that computation off-chain, but as identities may mutate after the claim contract is deployed it can provide an extra level of on-chain assurance.

```
/* ... */
function claim(uint256 index, address account, uint256 amount, bytes32[] calldata␣
↪merkleProof) external override {
  require(claimsEnabled, "Airdrop::claim: Claims not enabled.");
  require(!isClaimed(index), "Airdrop::claim: Drop already claimed.");

  // try to get the delegate of the account. this will fail if the account isn't␣
↪registered, or is in an unusable state
  signataIdentity.getDelegate(account);

```

(continues on next page)

(continued from previous page)

```
  bytes32 node = keccak256(abi.encodePacked(index, account, amount));
  require(MerkleProof.verify(merkleProof, merkleRoot, node), "Airdrop::claim: Invalid␣
↪proof.");

  _setClaimed(index);

  require(token.transfer(msg.sender, amount), "Airdrop::claim: Transfer failed.");
  emit Claimed(index, msg.sender, amount);
}
/* ... */
```

Dividend Rewards Distribution

Some DeFi projects use “dividend” based distribution of assets. Enforcing of holding an identity, or a specific NFT right, can be simply included in-line with functions that set the balance of the address. If it holds a Signata Identity, then it can receive a balance fo the dividend contract. Otherwise it is treated like an excluded contract

```
/* ... */
function setBalance(address payable account, uint256 newBalance)
    external
    onlyOwner
{
    if (excludedFromRewards[account]) {
        return;
    }

    // if it's not a registered identity, then it will be treated like it's in the␣
↪exclusion list
    try signataIdentity.getDelegate(account) {
      if (newBalance >= minTokenBalanceForRewards) {
          _setBalance(account, newBalance);
      } else {
          _setBalance(account, 0);
      }
    } catch { return; }
}
/* ... */
```

1.6 Identities

1.6.1 About Signata Identities

An identity is created with 3 keys, an **Identity** key, a **Delegate** key, and a **Security** key.

Note that the examples in this documentation are provided in JavaScript and are derived from the sata-contracts-v1 repository unit tests. If you wish to see more comprehensive details on how to interact with Signata identities, then for now the unit tests are the best option.

Elliptic curve signatures in the examples below are created using `ethereumjs-util`. Any time you see `Util.ecsign(...`, it will have come from the following import:

```
const Util = require('ethereumjs-util');
```

Cryptographic signing in other languages can vary considerably.

About Key Lifecycles

Each time a private key is used for any publicly-available information (such as digitally signed records), the risk of key material compromise will increase through known-plaintext attacks. Most Public Key Infrastructure systems enforce key life limits (typically 1 to 2 years) on an active key, rolling over to new keys before the key could theorically become compromised.

Signata will not enforce key lifespan lengths for identities, however the use of the 3-key identity lifecycle model will help to mitigate any risk of exposure through prolonged use of an identity, and also allow for simple rollover to a new set of keys if a user suspects any compromise.

The **recommended approach** to handling keys with Signata identities is to use 3 separate seeds for deriving the 3 keys needed to create each identity. Simply derive the 0th key from each seed for the first identity, then the 1st key from each seed for the second identity, and so on. In the unit test suite you will see we have derived all keys from the same seed - this was done purely to keep the unit tests simple but is not the recommended approach for production use.

Identity Key

The identity key is the core identity identifier. This key is intentionally “useless” in the context of lifecycle management with Signata. That is, it cannot be used to perform lifecycle event management of itself, just to ensure the key material is not used extraneously. It will bear all rights assigned to it, and be used for proving ownership, but all lifecycle events are instead handled by the Delegate Key.

Delegate Key

The delegate key is the lifecycle management key for a Signata identity. It is used to lock, unlock, destroy, and rollover identities after they are created.

It is not recommended that these keys be derived from the same seed as the identity key, otherwise a seed compromise will immediately compromise this key as well.

Security Key

The security key is a fallback key, specifically designed to provide recovery mechanisms in the event of a compromise. In ideal circumstances this key should never be needed to be used, it will only be used for important lifecycle events.

It is not recommended that these keys be derived from the same seed as the identity key, otherwise a seed compromise will immediately compromise this key as well.

Domain Separator

TODO: Define this

```
const CHAINID = 1; // CHAINID is determined from the network the contract is deployed on,
↪ it's set as the ETH mainnet ID here for the example

const IDENTITY_CONTRACT_ADDRESS = '0x6B47e26A52a9B5B467b98142E382c081eA97B0fc'; // again,
↪ this is an example. This is just the ETH mainnet ID contract address.

// these consts are fixed for all networks:
const EIP712DOMAINTYPE_DIGEST =
↪'0xd87cd6ef79d4e2b95e15ce8abf732db51ec771f1ca2edccf22a46c729ac56472';
const NAME_DIGEST = '0xfc8e166e81add347414f67a8064c94523802ae76625708af4cddc107b656844f';
const VERSION_DIGEST =
↪'0xc89efdaa54c0f20c7adf612882df0950f5a951637e0307cdcb4c672f298b8bc6';
const SALT = '0x233cdb81615d25013bb0519fbe69c16ddc77f9fa6a9395bd2aecfdfc1c0896e3';

// deriving the DOMAIN_SEPARATOR for the contract. You can derive this at any time after␣
↪the contract has been deployed.
const DOMAIN_SEPARATOR = web3.utils.sha3(
  EIP712DOMAINTYPE_DIGEST
    + NAME_DIGEST.slice(2)
    + VERSION_DIGEST.slice(2)
    + CHAINID.toString('16').padStart(64, '0')
    + IDENTITY_CONTRACT_ADDRESS.slice(2).padStart(64, '0')
    + SALT.slice(2),
  {encoding: 'hex'}
);
```

1.6.2 Creating an Identity

An identity is registered within the `SignataIdentity` contract using the following process:

- A `SHA3 Hash` is generated of the truncation of the `TXTYPE_CREATE_DIGEST`, the `Delegate Address`, and the `Security Address`.
- The `SHA3 Hash` is digitally signed by the `Identity Private Key`.
- The `SHA3 Hash`, the `Delegate Address`, and the `Security Address` are sent to `create()` with the sender being the `Identity Key` to register the identity.
- A `Create` event is emitted from the contract upon successful creation.

All three keys *must* be distinct, and they cannot already be in use with another identity (that is, you cannot re-use any of the keys more than once).

TODO: Define the purpose of 0x1901 before the DOMAIN_SEPARATOR

```
const TXTYPE_CREATE_DIGEST =
↪'0x469a26f6afcc5806677c064ceb4b952f409123d7e70ab1fd0a51e86205b9937b';

const inputHash = web3.utils.sha3(
  TXTYPE_CREATE_DIGEST
    + d1.slice(2).padStart(64, '0')
    + s1.slice(2).padStart(64, '0'),
  {encoding: 'hex'}
);

const hashToSign = web3.utils.sha3(
  '0x19' + '01' + DOMAIN_SEPARATOR.slice(2) + inputHash.slice(2),
  {encoding: 'hex'}
);

const { r, s, v } = Util.ecsign(
  Buffer.from(hashToSign.slice(2), 'hex'),
  Buffer.from(i1Private.slice(2), 'hex')
);

const createReceipt = await idContract.create(v, r, s, d1, s1, { from: i1 });

await expectEvent(createReceipt, 'Create', {
  identity: i1,
  delegateKey: d1,
  securityKey: s1
});
```

1.6.3 Locking an Identity

To prevent an identity being modified or used, such as in the event you believe it has been compromised, you can flag the identity as "locked". You will not be able to use the identity whilst it is locked, but the delegate can still perform a rollover to ensure any compromise can be mitigated.

Locking is just a call to the `lock()` function by the delegate, no additional signatures are needed.

```
await expectEvent(await idContract.lock(i1, { from: d1 }), 'Lock', { identity: i1 });

expect(await idContract.isLocked(i1, { from: i1 })).to.equal(true);
```

Unlocking an identity is more complicated and will be covered in its own section.

1.6.4 Unlocking an Identity

To unlock an identity, the `delegate` and `security` keys are needed to digitally sign the authorization of unlocking.

This is more complicated than locking, as locking is designed to be able to able to be performed quickly to respond to a threat but unlocking to require a user to assess whether the original threat still remains.

```
const TXTYPE_UNLOCK_DIGEST =
↪'0xd814812ff462bae7ba452aadd08061fe1b4bda9916c0c4a84c25a78985670a7b';

```

(continues on next page)

(continued from previous page)

```
const inputHash = web3.utils.sha3(TXTYPE_UNLOCK_DIGEST + "0x01".slice(2).padStart(64, '0
↪'), {encoding: 'hex'});

const hashToSign = web3.utils.sha3(
  '0x19' + '01' + DOMAIN_SEPARATOR.slice(2) + inputHash.slice(2),
  {encoding: 'hex'}
);

const sig1 = Util.ecsign(
  Buffer.from(hashToSign.slice(2), 'hex'),
  Buffer.from(d2Private.slice(2), 'hex')
);

const delegateV = sig1.v;
const delegateR = sig1.r;
const delegateS = sig1.s;

const sig2 = Util.ecsign(
  Buffer.from(hashToSign.slice(2), 'hex'),
  Buffer.from(s2Private.slice(2), 'hex')
);

const securityV = sig2.v;
const securityR = sig2.r;
const securityS = sig2.s;

const unlockReceipt = await idContract.unlock(
  i2,
  delegateV,
  delegateR,
  delegateS,
  securityV,
  securityR,
  securityS,
  { from: d2 }
);

await expectEvent(unlockReceipt, 'Unlock', {
  identity: i2
});
```

1.6.5 Destroying an Identity

Once an identity is no longer required, it can be destroyed to prevent further use. Identity destruction cannot be undone, it is permanent. If you wish to disable an identity temporarily, just lock it instead.

Identity destruction requires both the `delegate` and the `security` keys to digitally sign the authorization of destruction.

Also bear in mind that this is a blockchain and all the history of the identity will still remain on the chain after destruction. Destruction in this case simply means to prevent the key from ever being used again in the Signata identity ecosystem, and identity providers must honour the mutation that has been made.

```
const TXTYPE_DESTROY_DIGEST =
↪'0x21459c8977584463672e32d031e5caf426140890a0f0d2172da41491b70ef9f5';

const inputHash = web3.utils.sha3(TXTYPE_DESTROY_DIGEST, {encoding: 'hex'});

const hashToSign = web3.utils.sha3(
  '0x19' + '01' + DOMAIN_SEPARATOR.slice(2) + inputHash.slice(2),
  {encoding: 'hex'}
);

const sig1 = Util.ecsign(
  Buffer.from(hashToSign.slice(2), 'hex'),
  Buffer.from(d1Private.slice(2), 'hex')
);

const delegateV = sig1.v;
const delegateR = sig1.r;
const delegateS = sig1.s;

const sig2 = Util.ecsign(
  Buffer.from(hashToSign.slice(2), 'hex'),
  Buffer.from(s1Private.slice(2), 'hex')
);

const securityV = sig2.v;
const securityR = sig2.r;
const securityS = sig2.s;

const destroyReceipt = await idContract.destroy(
  i1,
  delegateV,
  delegateR,
  delegateS,
  securityV,
  securityR,
  securityS,
  { from: i1 }
);

await expectEvent(destroyReceipt, 'Destroy', {
  identity: i1
});
```

1.6.6 Rolling Over an Identity

Much like the destruction event, rollovers also need signatures from both the `delegate` and `security` keys. Rollover will transfer control of an identity address to new delegate and security keys, the identity itself will remain the same.

After this mutation has been made the old delegate and security keys can no longer be used.

```
const TXTYPE_ROLLOVER_DIGEST =
↪'0x3925a5eeb744076e798ef9df4a1d3e1d70bcca2f478f6df9e6f0496d7de53e1e';
const inputHash = web3.utils.sha3(
  TXTYPE_ROLLOVER_DIGEST
    + d3.slice(2).padStart(64, '0')
    + s3.slice(2).padStart(64, '0')
    + "0x00".slice(2).padStart(64, '0'), {encoding: 'hex'});

const hashToSign = web3.utils.sha3(
  '0x19' + '01' + DOMAIN_SEPARATOR.slice(2) + inputHash.slice(2),
  {encoding: 'hex'}
);

const sig1 = Util.ecsign(
  Buffer.from(hashToSign.slice(2), 'hex'),
  Buffer.from(d2Private.slice(2), 'hex')
);

const delegateV = sig1.v;
const delegateR = sig1.r;
const delegateS = sig1.s;

const sig2 = Util.ecsign(
  Buffer.from(hashToSign.slice(2), 'hex'),
  Buffer.from(s2Private.slice(2), 'hex')
);

const securityV = sig2.v;
const securityR = sig2.r;
const securityS = sig2.s;

const rolloverReceipt = await idContract.rollover(
  i2,
  delegateV,
  delegateR,
  delegateS,
  securityV,
  securityR,
  securityS,
  d3,
  s3,
  { from: d2 }
);

await expectEvent(rolloverReceipt, 'Rollover', {
  identity: i2,
  delegateKey: d3,
```

(continues on next page)

(continued from previous page)

```
  securityKey: s3,
});
```

1.7 Integrations

1.8 Links

This document details all links to external services related to the Signata token.

1.8.1 News

- Blog
- Twitter
- Telegram

1.8.2 Development Community

- Discord
- Github

1.8.3 Exchanges

SATA

- Ethereum Uniswap V2 SATA-WETH
- Ethereum Uniswap V3 SATA-WETH
- Ethereum Bancor SATA-BNT
- Ethereum Sushiswap SATA-WETH
- P2PB2B SATA-USDT
- Coinsbit SATA-USDT
- TimeX SATA-USDT
- Binance Smart Chain ApeSwap SATA-WBNB

dSATA

- Ethereum Uniswap V2 dSATA-WETH

1.9 Risk

Warning: Any source code provided on this site is unaudited and provides no guarantee that it will be functional or safe to use. Ensure you test before deploying to mainnet or production use.

1.9.1 signata-risk-api

Deployment

The Risk API is a python flask service using Supabase for record retrieval and storage. It can be easily deployed on services like DigitalOcean with the following environment variables set:

`SUPABASE_URL`

`SUPABASE_KEY`

The Congruent Labs Risk API service is hosted at `https://risk.signata.net/`

Get Risk Level

Request:

`GET /api/v1/riskLevel/{addr}`

Response:

Just responding with single values in the body for consumption by oracles.

`0 - unknown risk level`

`1...5 - risk levels`

Errors:

`400 - Invalid Address`

Add Risk Event

Request:

`POST /api/v1/riskEvent`

Response:

`200 OK`

Errors:

`400 - Invalid Address`

1.10 Veriswap