// WebApp Pentest

01.02.2025 - 01.14.2025

Account Abstraction Schnorr Signatures SDK

InFlux Technologies

HALBORN

Account Abstraction Schnorr Signatures SDK - InFlux Technologies

Prepared by: HALBORN

Last Updated 02/21/2025

Date of Engagement by: January 2nd, 2025 - January 14th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	1	6	2	1

TABLE OF CONTENTS

1. Introduction

`InFlux Technologies` engaged Halborn to conduct a security assessment on their web application beginning on 01/02/2025 and ending on 01/15/2025. The security assessment was scoped to the source code files provided to the `Halborn team`.

2. Assessment Summary

The team at **Halborn** was provided one week and a half for the engagement and assigned a full-time security engineer to verify the security of the scoped source code application files. The security engineer is a blockchain and smart contract security expert with advanced penetration testing, smart contract hacking, and deep knowledge of multiple blockchain protocols.

The security assessment identified multiple critical areas requiring attention in the analyzed codebase, involving several issues and misconfigurations that the `InFlux Technologies` should address to enhance the application's security.

The lack of proper validation for external calls raised concerns about unchecked interactions with third-party contracts, potentially leading to unintended execution of malicious code.

Several medium-severity issues were also observed. The use of predictable salts during contract deployments could expose the system to collision attacks, jeopardizing address uniqueness. Additionally, the default hash function was not clearly specified, which could create inconsistencies or weaken the cryptographic integrity of the system. Sensitive information, including private keys, was potentially being stored within environment variables without sufficient protection, amplifying the risk of unauthorized access. The codebase also relied on third-party dependencies with known vulnerabilities, potentially exposing the entire project to inherited security flaws. Moreover, hardcoded transaction cost parameters may limit flexibility and could be exploited if not carefully controlled.

Furthermore, the absence of public key validation could allow unauthorized entities to submit invalid keys, increasing the likelihood of malicious transactions being accepted.

Lower-risk findings included the presence of insecure methods, which, although not actively used, could become a risk if reintroduced or overlooked in future development cycles.

Some other low severity issues involved cryptographic key handling and transaction integrity. Specifically, the potential reuse of ***nonces*** in the ***Schnorr signature*** scheme presented a substantial risk of private key leakage, compromising the overall integrity of the signing process.

Finally, an informational observation was also noted regarding the library usage, emphasizing the importance of clearly documenting and maintaining external code integrations.

Overall, while the project demonstrates solid foundations in many areas, these identified issues highlight the need for a more comprehensive approach to input validation, cryptographic hygiene, and dependency management to ensure long-term security and resilience.

It is recommended to resolve all the security issues listed in the document to improve the security health of the application and its underlying infrastructure.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the penetration test... While manual testing is recommended to uncover flaws in logic, process and implementation; automated testing techniques assist enhance coverage of the solution and can quickly identify flaws in it.

The following phases and associated tools were used throughout the term of the assessment:

- Research about the scoped source code
- Technology stack-specific vulnerabilities and public source code assessment
- Vulnerable or outdated software
- Exposure of any critical information
- Application logic flaws
- Access Handling
- Authentication / Authorization flaws
- Lack of validation on inputs and input handling
- Brute force protections
- Sensitive information disclosure
- Source code review

4. RISK METHODOLOGY

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the LIKELIHOOD of a security incident and the IMPACT should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

- **10** - CRITICAL
- **9 - 8** - HIGH
- **7 - 6** - MEDIUM
- **5 - 4** - LOW
- **3 - 1** - VERY LOW AND INFORMATIONAL

5. SCOPE

FILES AND REPOSITORY

(a) Repository: account-abstraction

(b) Assessed Commit ID: 588c582

(c) Items in scope:

Out-of-Scope: aa-schnorr-multisig-sdk/src/generated/typechain/*, OpenZeppelin files, Third party libraries

REMEDIATION COMMIT ID:

- https://github.com/RunOnFlux/account-abstraction/pull/17

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	6	2	1

IMPACT X LIKELIHOOD

	HAL-04	HAL-03		
	HAL-08	HAL-06 HAL-07		
	HAL-09		HAL-05 HAL-02	
	HAL-01			

HAL-10				

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
NOT-USED INSECURE METHOD	LOW	RISK ACCEPTED - 02/05/2025
PREDICTABLE SALT (COLISSION ATTACK RISK)	MEDIUM	NOT APPLICABLE
LACK OF EXTERNAL CALLS VALIDATION	HIGH	RISK ACCEPTED - 02/05/2025
SENSITIVE INFORMATION IN ENV VARS	MEDIUM	RISK ACCEPTED - 02/05/2025
VULNERABLE THIRD-PARTY DEPENDENCIES	MEDIUM	RISK ACCEPTED - 02/05/2025
HARDCODED TRANSACTION COST	MEDIUM	NOT APPLICABLE
UNSPECIFIED DEFAULT HASH FUNCTION	MEDIUM	SOLVED - 02/05/2025
LACK OF KEY VALIDATION	MEDIUM	SOLVED - 02/05/2025
POTENTIAL NONCE REUSAGE (KEY LEAKAGE RISK)	LOW	SOLVED - 02/05/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LIBRARY USAGE RECOMMENDATION	INFORMATIONAL	ACKNOWLEDGED - 02/05/2025

7. FINDINGS & TECH DETAILS

7.1 NOT-USED INSECURE METHOD

// LOW

Description

The `KeyPair` class exposed the `privateKey` in the `toJson()` method, which could potentially serialize and expose sensitive private key information to any consumer of this function.
Although it was not invoked in the assessed version, its presence in the code represented a potential risk, particularly if the code is modified in future versions to call it or if the method is inadvertently exposed to insecure channels or logging systems.

Proof of Concept

- `src/types/key-pair.ts`

```
24  firstnumber=24
25  toJson(): string {
26    return JSON.stringify({
27      publicKey: this.publicKey.toHex(),
28      privateKey: this.privateKey.toHex(),
29    })
30  }
```

Score

Impact: 3

Likelihood: 2

Recommendation

- **Remove all the potential insecure methods** that are not being used.
- **Avoid Serializing Sensitive Data:** Do not expose private keys or sensitive information in the `toJson()` method or any other serialization mechanism.
- **Limit Access to Private Keys:** Private keys should only be used within secure contexts and should never be serialized or exposed through external interfaces.

Remediation

RISK ACCEPTED: According to the **InFlux Technologies team**, they wanted to accept the risk of this issue.

Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.2 PREDICTABLE SALT (COLISSION ATTACK RISK)

// MEDIUM

Description

The codebase used in different code points a predictable salt value (`const salt = "this is salt"`), which could lead to a collision attack risk.
A predictable salt could allow an attacker to precompute address collisions if they can anticipate the public keys used during contract deployments. This could result in unauthorized address generation, undermining the integrity of the smart account creation process.

Impact

Using a predictable salt in the smart account creation process increases the risk of:

- **Address Collision Attacks:** An attacker could precompute the same address, potentially overriding a legitimate deployment.
- **Unauthorized Asset Access:** If an address collision occurs, funds could be diverted or compromised.
- **Smart Contract Integrity Compromise:** The deterministic nature of the salt could allow unauthorized access to smart accounts with predictable addresses.

Proof of Concept

Listed below are some examples where the “salt” implementation was detected to be potentially insecure.

- `examples/account-address/account_address.ts`

```
29 function getAddressOffChain(combinedAddresses: string[], salt: string)
30   const factorySalt = "aafactorysalt"
31   const factoryAddress = deployments[polygon.id]?.MultiSigSmartAccountF
```

- `examples/account-address/account_address.ts`

```
78 const combinedAddresses = getAllCombinedAddrFromKeys(publicKeys, 3)
79 const salt = "random salt for randomly generated priv keys"
```

- `examples/account-deployment/factory-create-account-method-call-deployment.ts`

```
20 let privKey2
21 do privKey2 = randomBytes(32)
22 while (!secp256k1.privateKeyVerify(privKey2))
23 const schnorrSigner2 = createSchnorrSigner(privKey2)
24 const publicKey2 = schnorrSigner2.getPubKey()
25
26 const salt = "this is salt"
```

- examples/sign_3_of_3/sign-3_of_3.ts

```
/**
 * Multi Sig participant #3
 */
let privKey3
do privKey3 = randomBytes(32)
while (!secp256k1.privateKeyVerify(privKey3))
const schnorrSigner3 = createSchnorrSigner(privKey3)

const publicKey3 = schnorrSigner3.getPubKey()

/**
 * Participants select SALT for the Smart Account
 * (Same 3 Participants can have multiple smart account's where id of t
 */

const salt = "this is salt shared by participants 3"
```

- src/accountAbstraction/multiSigSmartAccount.ts

```
export async function createMultiSigSmartAccount({
  transport,
  chain,
  entryPoint = getEntryPoint(chain, { version: "0.6.0" }),
  accountAddress,
  combinedAddress = [],
  salt: _salt,
}: CreateMultiSigSmartAccountParams): Promise<MultiSigSmartAccount> {
  const client = createBundlerClient({
    transport,
    chain,
  })
  const salt = _salt ?? ethers.encodeBytes32String("salt")
```

Score

Impact: 5

Likelihood: 2

Recommendation

To mitigate the predictable salt issue:

- **Avoid Hardcoded Values:** Ensure salts are dynamically generated and not hardcoded in the source code.
- **Use Cryptographically Secure Random Salts:** Generate salts using secure randomness (`randomBytes()` or `ethers.utils.randomBytes`).

```
const salt = randomBytes(32).toString("hex")
```

- **Incorporate User Input:** Optionally, derive the salt from both user-specific data and secure random values.

Remediation

NOT APPLICABLE: Finally agreed with the **InFlux Technologies team** that this issue was not applicable.

Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.3 LACK OF EXTERNAL CALLS VALIDATION

// HIGH

Description

Non-validated external calls occur when a function invokes an external contract without verifying the return value or handling potential errors.
Several external calls were detected without proper validation.

Impact

This can lead to reentrancy attacks or unexpected side effects if the external call fails or returns an unexpected result, directly causing a potential impact in the availability or integrity of the environment.

Proof of Concept

Listed below, there are some examples of unvalidated calls that may fail or cause an unconsistent or unexpected behavior of the application execution flow.

- examples/account-address/account_address.ts

```
37 async function getAddressAlchemyAASDK(combinedAddresses: Address[], sal
38   const rpcUrl = process.env.ALCHEMY_RPC_URL
39   const transport = http(rpcUrl)
40   const multiSigSmartAccount = await createMultiSigSmartAccount({
41     transport,
42     chain: CHAIN,
43     combinedAddress: combinedAddresses,
44     salt: saltToHex(salt),
45     entryPoint: getEntryPoint(CHAIN),
46   })
47
48   return multiSigSmartAccount.address
49 }
50
```

- src/helpers/create2.ts

```
118 export async function getAccountImplementationAddress(factoryAddress: s
119   const smartAccountFactory = new ethers.Contract(factoryAddress, Multi
120   const accountImplementation = await smartAccountFactory.accountImplem
121   return accountImplementation
122 }
```

- src/helpers/factory-helpers.ts

```
export async function predictAccountAddress(
  factoryAddress: Hex,
  signer: Signer,
  combinedPubKeys: string[],
  salt: string
): Promise<`0x${string}`> {
  const smartAccountFactory = new ethers.Contract(factoryAddress, Multi
  const saltHash = ethers.keccak256(ethers.toUtf8Bytes(salt))
  const predictedAccount = await smartAccountFactory.getAccountAddress(
  return predictedAccount as Hex
}
```

- examples/account-deployment/user-operation-init-code-deployment.ts

```
const factoryAddress = deployments[CHAIN.id]?.MultiSigSmartAccountFacto

  const smartAccountAdddress = await predictAccountAddrOnchain(factoryA
  console.log("Smart Account Address:", smartAccountAdddress)

  const initTransactionCost = parseUnits("0.05", 18)
  const addBalanceToSmartAccountTransaction = await wallet.sendTransact
  await addBalanceToSmartAccountTransaction.wait()

  const transport = http(process.env.ALCHEMY_RPC_URL)
  const multiSigSmartAccount = await createMultiSigSmartAccount({
    transport,
    chain: CHAIN,
    combinedAddress: combinedAddresses,
    salt: saltToHex(salt),
    entryPoint: getEntryPoint(CHAIN),
  })
```

Score

Impact: 5

Likelihood: 3

Recommendation

- Implement proper error handling and validation for external calls.
- Use `try/catch` blocks to handle exceptions and log errors appropriately.
- Ensure the external contract being called follows best practices for reentrancy protection.

Remediation

RISK ACCEPTED: According to the **InFlux Technologies team**, they wanted the library to throw the exact error message. Handling of error and validation should happen on a client using the library.

Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.4 SENSITIVE INFORMATION IN ENV VARS

// MEDIUM

Description

Sensitive information, such as API keys and private keys, were being stored directly in environmental variables.
While environment variables are commonly used to configure applications, storing highly sensitive data in them poses security risks.
Unauthorized users or malicious insiders who gain access to the system or CI/CD pipeline may retrieve these sensitive variables, leading to potential data leaks, unauthorized access to external services, and compromised application security.
The repository code used sensitive private keys stored in a `.env` file, which are accessed programmatically via `process.env.PRIVATE_KEY` clauses or similar.
Since the repository is public, there is also a risk that developers may inadvertently copy these sensitive values directly into production environments. This practice can lead to unauthorized access and compromise of digital assets. If environment variables are not properly managed or are leaked, it can result in exposure of private keys.

Impact

Storing private keys or other sensitive information inside environment variables can lead to severe security risks in case of exposure, such as:

- Unauthorized access to blockchain wallets or smart contract deployments.
- Financial loss due to stolen assets.
- Compromise of cryptographic keys, leading to broader security breaches.

Proof of Concept

Listed below are some examples where sensitive information was being stored inside environmental variables.

- `examples/account-deployment/factory-create-account-method-call-deployment.ts`

```
15  async function factoryCallCreateSmartAccount() {
16    const privKey1 = process.env.PRIVATE_KEY as Hex
17    const schnorrSigner1 = createSchnorrSigner(privKey1)
18    const publicKey1 = schnorrSigner1.getPubKey()
19
20    let privKey2
21    do privKey2 = randomBytes(32)
22    while (!secp256k1.privateKeyVerify(privKey2))
23    const schnorrSigner2 = createSchnorrSigner(privKey2)
24    const publicKey2 = schnorrSigner2.getPubKey()
25
26    const salt = "this is salt"
27
```

```
28
29    const publicKeys = [publicKey1, publicKey2]
30
31    const combinedAddresses = getAllCombinedAddrFromKeys(publicKeys, 2)
32
33    const provider = new JsonRpcProvider(process.env.ALCHEMY_RPC_URL)
34    const wallet = new Wallet(process.env.PRIVATE_KEY, provider)
35
36    const factoryAddress = deployments[CHAIN.id]?.MultiSigSmartAccountFac
37
38    const smartAccountAddress = await predictAccountAddrOnchain(factoryAc
      console.log("Smart Account Address:", smartAccountAddress)
```

- examples/sign_3_of_3/sign-3_of_3.ts

```
26  async function main() {
27    /**
28     * Wallet to cover initial transaction costs/prefund smart account
29     */
30    const provider = new JsonRpcProvider(process.env.ALCHEMY_RPC_URL)
31    const wallet = new Wallet(process.env.PRIVATE_KEY, provider)
32
33    /**
34     * Precondition:
35     * 3 Participants sharing they Public Key's
36     */
37
38    /**
39     * Multi Sig participant #1
40     */
41    const privKey1 = process.env.PRIVATE_KEY as Address
42    const schnorrSigner1 = createSchnorrSigner(privKey1)
43
44    const publicKey1 = schnorrSigner1.getPubKey()
```

- examples/user-operation/transfer-native/transfer-native.ts

```
29  async function main() {
30    /**
31     * Wallet to cover initial transaction costs/prefund smart account
32     */
33    const provider = new JsonRpcProvider(process.env.ALCHEMY_RPC_URL)
34    const wallet = new Wallet(process.env.PRIVATE_KEY, provider)
35
36    /**
37     * Requirements
```

```
38      * Eth/Matic: 0.06
39      */
40
41     const walletBalance = await provider.getBalance(wallet.address)
42     if (walletBalance < parseEther("0.06")) throw new Error("Not enough r
43
44     const privKey1 = process.env.PRIVATE_KEY as Address
45     const schnorrSigner1 = createSchnorrSigner(privKey1)
46
47     const publicKey1 = schnorrSigner1.getPubKey()
```

Score

Impact: 4

Likelihood: 3

Recommendation

- **Remove sensitive keys** from `.env` files in public repositories.
- **Use environment-specific secrets management tools** (e.g., AWS Secrets Manager, HashiCorp Vault or hardware key management (HSM)).
- **Ensure the** `.env.example` file includes placeholders or comments explaining proper key usage, without actual secrets.
- **Implement CI/CD checks** to prevent committing sensitive data to public repositories.

Remediation

RISK ACCEPTED: According to the **InFlux Technologies team**, they wanted to accept the risk of this issue.

Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.5 VULNERABLE THIRD-PARTY DEPENDENCIES

// MEDIUM

Description

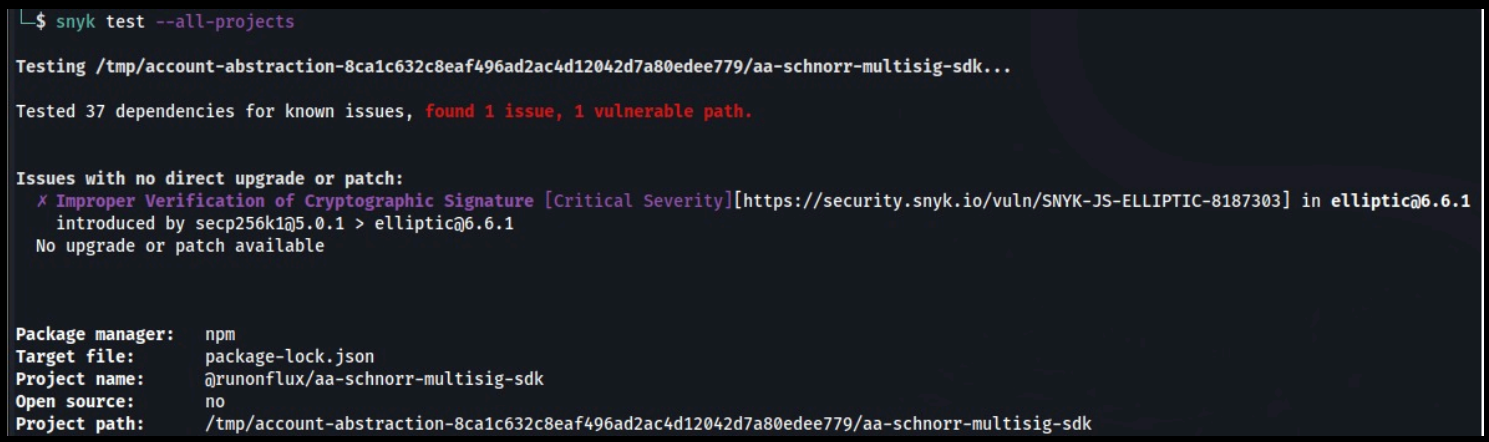
The scoped repository used multiple third-party dependencies. Several of the assessed dependencies had public-known vulnerabilities, many of them with CRITICAL or HIGH severity, that may pose a high risk to the global application security level.

Impact

Using vulnerable third-party libraries can result in security vulnerabilities in the project that can be exploited by attackers. This can result in data breaches, theft of sensitive information, and other security issues.

Proof of Concept

- `snyk test --all-projects` command output.



Score

Impact: 4

Likelihood: 3

Recommendation

Update all affected packages to its latest version.
It is strongly recommended to perform an automated analysis of the dependencies from the birth of the project and if they contain any security issues. Developers should be aware of this and apply any necessary mitigation measures to protect the affected application.

Remediation

RISK ACCEPTED: According to the **InFlux Technologies team**, they wanted to accept the risk of this issue.

Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.6 HARDCODED TRANSACTION COST

// MEDIUM

Description

The `initTransactionCost` variable was hardcoded in the source code examples, making it static and unresponsive to real-time gas price fluctuations on the blockchain network. This approach did not account for dynamic network conditions, potentially leading to incorrect transaction execution or unexpected failures due to insufficient gas provision.

Impact

The major security concern about this issue was the use of a hardcoded transaction value (`initTransactionCost`), which could lead to incorrect execution or gas manipulation. This value must be calculated dynamically.

- **Economic Denial of Service (EDoS):** If gas prices surge unexpectedly, a hardcoded gas limit may lead to transaction failures or delays.
- **Manipulation Risk:** Attackers could deliberately increase network congestion, forcing the hardcoded value to be insufficient, causing critical operations to fail.
- **Reduced Flexibility:** The contract cannot adapt to real-time gas price changes, increasing operational risk.

This vulnerability severity was lowered from HIGH to MEDIUM because all the hardcoded values for the transaction cost were found in the `examples` folder of the project. However, this bad practice should be removed, according to the best security practices.

Proof of Concept

- `examples/account-deployment/user-operation-init-code-deployment.ts`

```
51  const smartAccountAdddress = await predictAccountAddrOnchain(factoryAdd
52    console.log("Smart Account Address:", smartAccountAdddress)
53
54    const initTransactionCost = parseUnits("0.05", 18)
55    const addBalanceToSmartAccountTransaction = await wallet.sendTransact
56    await addBalanceToSmartAccountTransaction.wait()
```

- `examples/account-deployment/user-operation-init-code-deployment.ts`

```
125 const smartAccountAdddress = await predictAccountAddrOnchain(factoryAdd
126   console.log("Smart Account Address:", smartAccountAdddress)
127
128   const initTransactionCost = parseUnits("0.05", 18)
129   const addBalanceToSmartAccountTransaction = await wallet.sendTransact
130   await addBalanceToSmartAccountTransaction.wait()
```

- examples/sign_3_of_3/sign-3_of_3.ts

```
94  /**
95     * Prefund smart account
96     */
97    const initTransactionCost = parseUnits("0.05", 18)
98    const addBalanceToSmartAccountTransaction = await wallet.sendTransact
99    await addBalanceToSmartAccountTransaction.wait()
```

- examples/user-operation/transfer-erc20/transfer-erc20.ts

```
90  /**
91    * Prefund smart account
92    */
93   const initTransactionCost = parseUnits("0.05", 18)
94   const addBalanceToSmartAccountTransaction = await wallet.sendTransacti
95   await addBalanceToSmartAccountTransaction.wait()
```

- examples/user-operation/transfer-native/transfer-native.ts

```
79  /**
80     * Prefund smart account
81     */
82    const initTransactionCost = parseUnits("0.06", 18)
83    const addBalanceToSmartAccountTransaction = await wallet.sendTransact
84    await addBalanceToSmartAccountTransaction.wait()
```

Score

Impact: 4

Likelihood: 2

Recommendation

Similarly to other functions, consider relying on calls to specific contracts to determine the gas estimation. Alternatively, consider using APIs (e.g., blockchain node RPC endpoints or third-party services) as this is the more common approach for gas estimation.

- Replace the hardcoded `initTransactionCost` with a dynamic calculation based on the current gas price using on-chain data or a reliable oracle.
- Implement a buffer margin to account for sudden price spikes in congested conditions.
- Test the contract under varying gas price scenarios to ensure resilience.

Remediation

NOT APPLICABLE: Finally agreed with the **InFlux Technologies team** that this issue was not applicable.

Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.7 UNSPECIFIED DEFAULT HASH FUNCTION

// MEDIUM

Description

The optional parameter `hashFn` defaulted to `null`, relying on external callers to define the hash function, which introduced a cryptographic risk of using an insecure hash.

Impact

If an insecure hash function (e.g., MD5) is provided or `null` is used, it can compromise the integrity of the signatures.

Proof of Concept

Listed below, there are some examples of code snippets where the Hash Function was set to `null` by default.

- `src/signers/SchnorrSigner.ts`

```
69  signMessage(msg: string, hashFn: HashFunction | null = null): Signature
70      return Schnorrkel.sign(this.#privateKey, msg, hashFn)
71    }
```

- `src/signers/Schnorrkel.ts`

```
159 multiSigSign(
160     privateKey: Key,
161     msg: string,
162     publicKeys: Key[],
163     publicNonces: PublicNonces[],
164     hashFn: HashFunction | null = null
165   ): SignatureOutput {
166     const combinedPublicKey = Schnorrkel.getCombinedPublicKey(publicKey
167     const mappedPublicNonce = this.getMappedPublicNonces(publicNonces)
168     const mappedNonces = this.getMappedNonces()
169
170     const musigData = _multiSigSign(
171       mappedNonces,
172       combinedPublicKey.buffer,
173       privateKey.buffer,
174       msg,
175       publicKeys.map((key) => key.buffer),
176       mappedPublicNonce,
177       hashFn
178     )
```

- src/signers/Schnorrkel.ts

```
234 static verify(
235     signature: SchnorrSignature,
236     msg: string,
237     finalPublicNonce: FinalPublicNonce,
238     publicKey: Key,
239     hashFn: HashFunction | null = null
240   ): boolean {
241     return _verify(signature.buffer, msg, finalPublicNonce.buffer, publ
242   }
```

Score

Impact: 3

Likelihood: 4

Recommendation

- Set a cryptographically secure default hash function, such as SHA-256:

```
const defaultHashFunction = (msg: string) => createHash("sha256").update(ms
```

Remediation

SOLVED: The **InFlux Technologies team** solved this issue by removing the null value and also by adding the "`_hashMessage`" (solidityPackedKeccak256) as default.

SchnorrSigner.ts 4

account-abstraction-8ca1c632c8eaf496ad2ac4d12042d7a80edee779 > aa-schnorr-multisig-sdk > src > signers > SchnorrSigner.ts

```
11  export class SchnorrSigner extends Schnorrkel {
69
70    signMessage(msg: string, hashFn: HashFunction = _hashMessage): SignatureOutput {
71      return Schnorrkel.sign(this.#privateKey, msg, hashFn)
72    }
73
74    signHash(hash: string): SignatureOutput {
75      return Schnorrkel.signHash(this.#privateKey, hash)
76    }
77  }
78
```

Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.8 LACK OF KEY VALIDATION

// MEDIUM

Description

The method `getCombinedPublicKey` combined public keys without validating whether they were valid points on the curve. This may allow a *rogue-key* attack, where a malicious signer can construct a key pair that reveals the private key.
In a multi-signature scheme with aggregate signatures, several participants combine their public keys and collectively sign a message. The *rogue-key* attack occurs when a malicious attacker introduces a manipulated public key that invalidates the security of the scheme.
In this case, if a malicious actor managed to provide a rogue key during the execution process of the whole multi-signature scheme, the attack would be successful as there was no validation on the keys.

Impact

A rogue participant could manipulate the key combination process, potentially revealing the private key of honest signers.

- Total security breach: The attacker can sign alone on behalf of the entire group.
- Unilateral control of funds: In the context of smart contracts or multi-sig wallets, this could allow the attacker to transfer funds without the cooperation of the rest of the participants.

Proof of Concept

- `src/signers/Schnorrkel.ts`

```
102 | static getCombinedPublicKey(publicKeys: Key[]): Key {
103 |     if (publicKeys.length < 2) throw new Error("At least 2 public keys
104 |
105 |     const bufferPublicKeys = publicKeys.map((publicKey) => publicKey.bu
106 |     const L = _generateL(bufferPublicKeys)
107 |
108 |     const modifiedKeys = bufferPublicKeys.map((publicKey) => {
109 |       return secp256k1.publicKeyTweakMul(publicKey, _aCoefficient(publi
110 |     })
111 |
112 |     return new Key(Buffer.from(secp256k1.publicKeyCombine(modifiedKeys)
113 |   }
```

Score

Impact: 3

Likelihood: 4

Recommendation

- Validate that all public keys are valid points on the `secp256k1` curve before combining.
- Reject zero keys and invalid points explicitly.
- Example Fix:

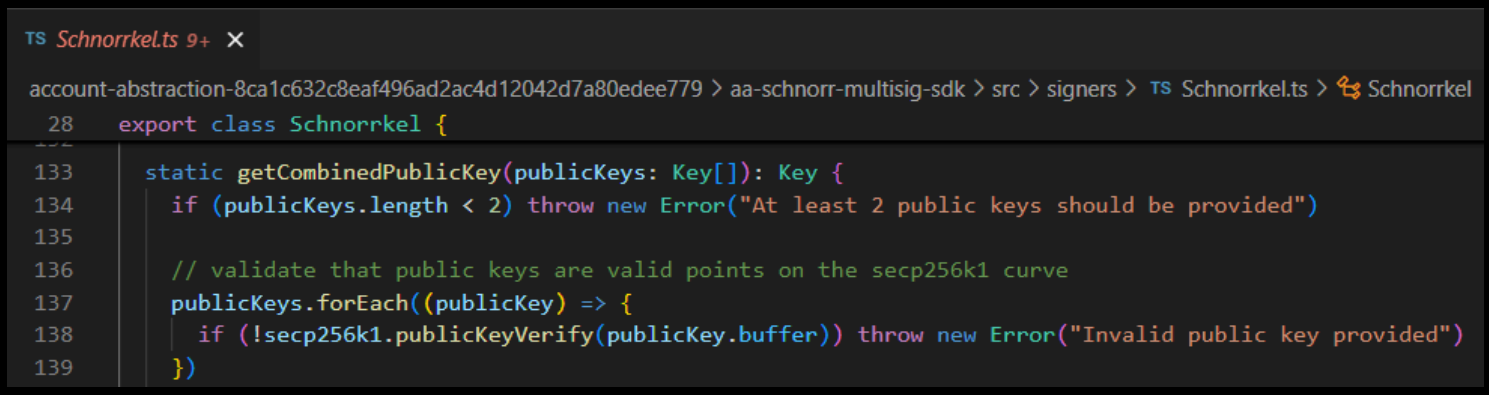
```
if (!secp256k1.publicKeyVerify(publicKey)) {
   throw new Error("Invalid public key provided");
}
```

To prevent a Rogue Key Attack:

- **Explicit verification of public keys**: Each participant must validate the public keys of the others before combining them.
- **Proof-of-knowledge protocols**: Implement a scheme such as *MuSig2* or *MuSig-DN*, which requires proof of knowledge of the associated private key before aggregation.

Remediation

SOLVED: The **InFlux Technologies team** followed Halborn recommendations to solve this issue.



Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.9 POTENTIAL NONCE REUSAGE (KEY LEAKAGE RISK)

// LOW

Description

The `Schnorrkel` class stored nonces in a private field `#nonces`. Although *nonces* were cleared after use with the `clearNonces` method, the process relied on correct execution flow. If *nonces* were reused or not properly cleared, it may lead to private key leakage.

Impact

Reusing *nonces* in Schnorr signatures can completely reveal the private key due to the mathematical properties of the algorithm.

Proof of Concept

- `src/signers/Schnorrkel.ts`

```
24  export class Schnorrkel {
25    #nonces: Nonces = {}
26
27    private _setNonce(privateKey: Buffer): string {
28      const { publicNonceData, privateNonceData, hash } = _generatePublic
29
30      const mappedPublicNonce: PublicNonces = {
31        kPublic: new Key(Buffer.from(publicNonceData.kPublic)),
32        kTwoPublic: new Key(Buffer.from(publicNonceData.kTwoPublic)),
33      }
34
35      const mappedPrivateNonce: Pick<NoncePairs, "k" | "kTwo"> = {
36        k: new Key(Buffer.from(privateNonceData.k)),
37        kTwo: new Key(Buffer.from(privateNonceData.kTwo)),
38      }
39
40      this.#nonces[hash] = { ...mappedPrivateNonce, ...mappedPublicNonce
41      return hash
42    }
```

```
61  private clearNonces(privateKey: Key): void {
62      const x = privateKey.buffer
63      const hash = _hashPrivateKey(x)
64
65      // eslint-disable-next-line @typescript-eslint/no-dynamic-delete
66
67
```

```
        delete this.#nonces[hash]
      }
```

Score

Impact: 2

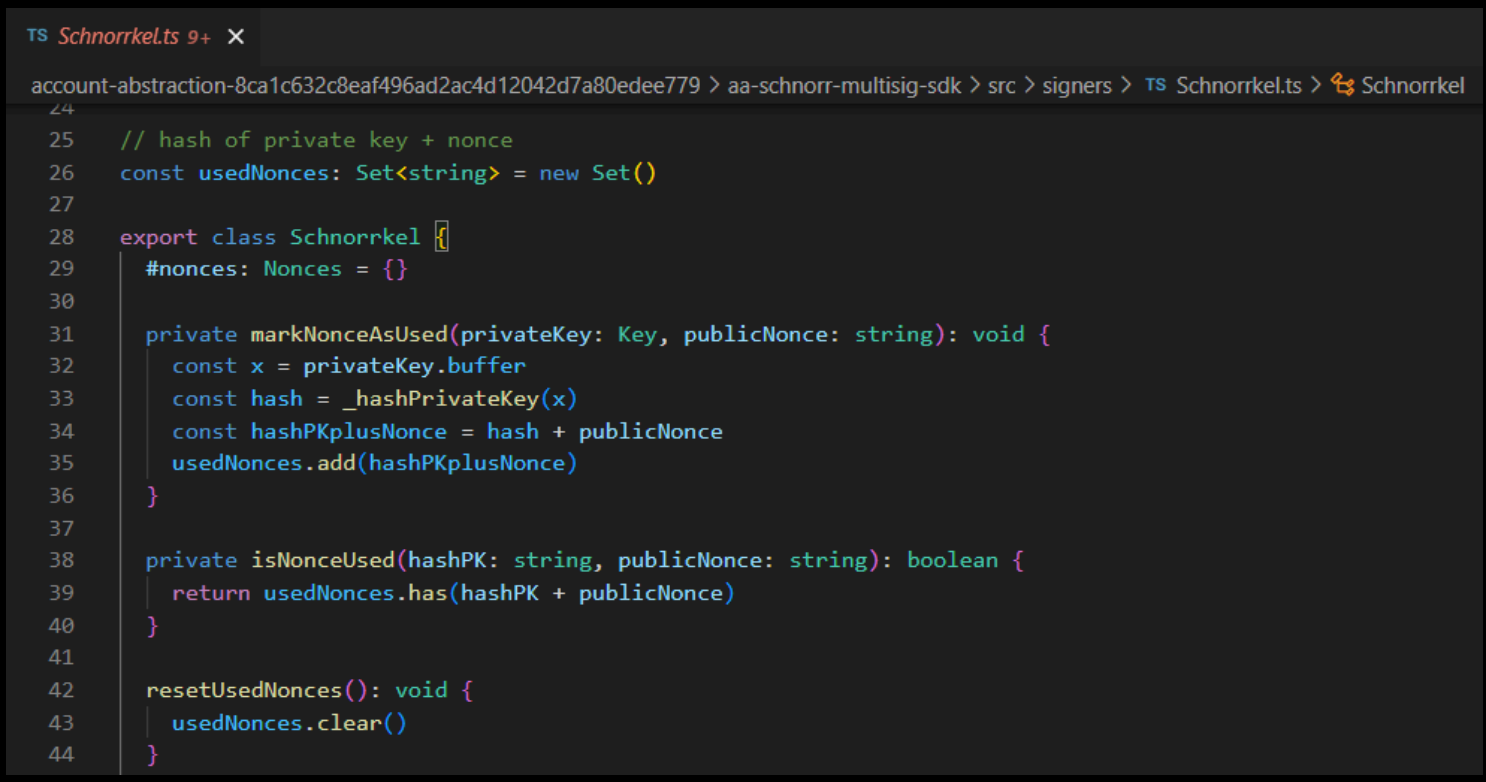
Likelihood: 2

Recommendation

- Implement a nonce-tracking mechanism to prevent reuse explicitly, even if an error occurs during the signing process.
- Consider enforcing a policy where each nonce is stored in a non-reusable bucket system.

Remediation

SOLVED: The **InFlux Technologies team** solved this issue by adding a final block with more cleaning and also a general bucket to remember all nonces used added as well.



Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17

7.10 LIBRARY USAGE RECOMMENDATION

// INFORMATIONAL

Description

The external `schnorrkel` library was not being used. Instead, the *Schnorr* implementation was based on the `Schnorrkel` class defined within the project itself. `secp256k1` version `5.0.1` was being used.
As mentioned, `secp256k1` is being used for *Schnorr signatures*. Although functional, the `schnorrkel` library is often used with `Curve25519` for better multi-signature capabilities and robustness and security. **This is a point to review if the security standard of the project requires a higher level of security against quantum attacks.**

Impact

While not a direct vulnerability, `secp256k1` has a weaker theoretical resistance to rogue attacks compared to `Curve25519`.

Score

Impact: 1

Likelihood: 1

Recommendation

- Consider migrating to the Curve25519 or a library specifically designed for Schnorr signatures such as `dalek` or `schnorrkel`.

Remediation

ACKNOWLEDGED: According to the **InFlux Technologies team**: "Acknowledged, great suggestion. However, we won't be migrating to it now or anytime soon.".

Remediation Hash

https://github.com/RunOnFlux/account-abstraction/pull/17
