Security Assessment Report

Defenders

Smart Contract and Frontend

March 5, 2024

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Defenders smart contract and the Defenders frontend.

The artifact of the audit was the source code of the following programs, excluding tests, in two zip files.

The initial audit focused on the following versions and revealed 15 issues or questions.

#	program	type	zip file md5sum
P1	Defenders smart contract	Solana	`50c22f4855bac2431a5741807f922add`
P2	Defenders frontend	Web2	`cd9fe9d6da4b195659fb7378e533ee0f`

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
DEFENDERS SMART CONTRACT		
[P1-C-1] Unconstrained new leaf owners and delegates	Critical	Resolved
[P1-C-2] Unconstrained compressed NFT mint	Critical	Resolved
[P1-M-1] Unconstrained dest_ata token account owner	Medium	Open
[P1-M-2] Unchecked amount in lock_solana and unlock_solana	Medium	Resolved
[P1-L-1] Bypass the TOTP	Low	Resolved
[P1-L-2] Integer overflow	Low	Resolved
[P1-I-1] CPI error handling	Info	Resolved
DEFENDERS FRONTEND		
[P2-M-1] Transaction unconfirmed, causing users unable to lock/unlock	Medium	Resolved
[P2-M-2] Missing confirmTransaction check in lockUnlockToken	Medium	Resolved
[P2-L-1] Use NFT whitelist to reduce security risks	Low	Acknowledged
[P2-L-2] Incorrect error handling In linkAuthAccountToWallet	Low	Resolved
[P2-L-3] Sanity check in the lock or unlock endpoints	Low	Resolved
[P2-I-1] Uncaught control flow	Info	Resolved
[P2-I-2] Hellomoon RPC should throw error friendly	Info	Resolved
[P2-I-3] Typos	Info	Open

Findings in Detail

DEFENDERS SMART CONTRACT

[P1-C-1] Unconstrained new leaf owners and delegates

The lock and unlock functionalities of the compressed NFTs are implemented using the "`Transfer`" instruction of the bubblegum smart contract, where the existing leaf node on the merkle tree will be replaced by the new leaf node.

```
/* mpl-bubblegum-0.7.0/src/lib.rs */
1194 | pub fn transfer<'info>(
1195 |     ctx: Context<'_, '_, '_, 'info, Transfer<'info>>,
1201 | ) -> Result<()> {
1214 |     let previous_leaf = LeafSchema::new_v0(
1215 |         asset_id,
1216 |         owner.key(),
1217 |         delegate.key(),
1218 |         nonce,
1219 |         data_hash,
1220 |         creator_hash,
1221 |     );
1222 |     // New leafs are instantiated with no delegate
1223 |     let new_leaf = LeafSchema::new_v0(
1224 |         asset_id,
1225 |         new_owner,
1226 |         new_owner,
1227 |         nonce,
1228 |         data_hash,
1229 |         creator_hash,
1230 |     );
```

1. Unconstrained new leaf owner account in lock_cnft()

When locking a compressed NFT, its "`leaf_owner`" and "`leaf_delegate`" on the merkle tree will be replaced by the "`new_leaf_owner`".

```
/* defenders/src/lib.rs */
913 | #[derive(Accounts)]
914 | pub struct LockCompressedNft<'info> {
915 |     #[account(mut)]
916 |     pub leaf_owner: Signer<'info>,
935 |     #[account(mut)]
```

```
936 |     pub leaf_delegate: Signer<'info>,
951 |     /// CHECK:
952 |     #[account(mut)]
953 |     pub new_leaf_owner: UncheckedAccount<'info>,
959 | }

181 | pub fn lock_cnft<'a, 'b, 'c, 'info>(
182 |     ctx: Context<'a, 'b, 'c, 'info, LockCompressedNft<'info>>,
188 | ) -> Result<()> {
228 |     accounts.extend(vec![
229 |         AccountMeta::new_readonly(ctx.accounts.tree_authority.key(), false),
230 |         AccountMeta::new_readonly(ctx.accounts.leaf_owner.key(), true),
231 |         AccountMeta::new_readonly(ctx.accounts.leaf_delegate.key(), false),
232 |         AccountMeta::new_readonly(ctx.accounts.new_leaf_owner.key(), false),
233 |         AccountMeta::new(ctx.accounts.merkle_tree.key(), false),
237 |     ]);
```

However, since the user-provided "`new_leaf_owner`" is unbounded, when it is the same as the "`leaf_owner`", the compressed NFT will be transferred to the same owner or the same delegate, and consequently, it will not be locked.

Consider adding a constraint on "`new_leaf_owner`" to ensure it's a PDA owned by this contract, such as the "`user_state`" account.

2. Unconstrained leaf owner and delegate accounts in unlock_cnft()

Based on the PDA seeds in lines 375-377, the signer "`leaf_owner`" is supposed to be "`user_state`", which is supposed to be the delegate account "`new_leaf_owner`" set in the "`lock_cnft()`".

```
/* src/lib.rs */
962 | pub struct UnlockCompressedNft<'info> {
968 |     /// CHECK
969 |     #[account(mut)]
970 |     pub leaf_owner: AccountInfo<'info>,
990 |     /// CHECK
991 |     #[account(mut)]
992 |     pub leaf_delegate: AccountInfo<'info>,

290 | pub fn unlock_cnft<'a, 'b, 'c, 'info>(
297 | ) -> Result<()> {
317 |     accounts.extend(vec![
318 |         AccountMeta::new_readonly(ctx.accounts.tree_authority.key(), false),
319 |         AccountMeta::new_readonly(ctx.accounts.leaf_owner.key(), true),
320 |         AccountMeta::new_readonly(ctx.accounts.leaf_delegate.key(), false),
321 |         AccountMeta::new_readonly(ctx.accounts.new_leaf_owner.key(), false),
```

```
322 |         AccountMeta::new(ctx.accounts.merkle_tree.key(), false),
323 |         AccountMeta::new_readonly(ctx.accounts.log_wrapper.key(), false),
324 |         AccountMeta::new_readonly(ctx.accounts.compression_program.key(), false),
325 |         AccountMeta::new_readonly(ctx.accounts.system_program.key(), false),
326 |     ]);
374 |     solana_program::program::invoke_signed(&instruction, &account_infos[..], &[&[
375 |         "user".as_bytes(),
376 |         ctx.accounts.payer.key().as_ref(),
377 |         &[user_state.bump]
378 |     ]])?;
```

However, the "`leaf_owner`" and "`leaf_delegate`" accounts are not constrained. As a result, it's possible to perform self-transfer in both "`lock_cnft`" and "`unlock_cnft`", and manipulate the lock states along the whole flow.

Consider adding constraints for both "`leaf_owner`" and "`leaf_delegate`" to ensure they can only be "`user_state`".

Resolution

Issue 1 has been resolved by checking the "`new_leaf_owner`".

Issue 2 has been fixed by validating the "`leaf_delegate`" and "`new_leaf_owner`".

DEFENDERS SMART CONTRACT

[P1-C-2] Unconstrained compressed NFT mint

When locking and unlocking compressed NFTs, "`nft_1`" is supposed to be the NFT token mint.

1. Unbounded nft_1 in lock_cnft()

Since "`nft_1`" is not included in the merkle tree leaf hash, it cannot be validated by the merkel tree in the bubblegum contract. As a result, it is not constrained.

In fact, before the compressed NFT is decompressed, its mint is not initialized.

```
/* defenders/src/lib.rs */
914 | pub struct LockCompressedNft<'info> {
929 |     /// CHECK
930 |     pub nft_1: AccountInfo<'info>,
931 |
932 |     #[account(init_if_needed, payer = leaf_owner, space = LockState::SPACE,
                  seeds = [b"lock", nft_1.key().as_ref()], bump)]
933 |     pub lock_state: Box<Account<'info, LockState>>,


181 | pub fn lock_cnft<'a, 'b, 'c, 'info>(
182 |     ctx: Context<'a, 'b, 'c, 'info, LockCompressedNft<'info>>,
183 |     root: [u8; 32],
184 |     data_hash: [u8; 32],
185 |     creator_hash: [u8; 32],
186 |     nonce: u64,
187 |     index: u32,
188 | ) -> Result<()> {
196 |     if lock_state.is_locked == 1{
197 |         return Err(ErrorCode::NftAlreadyLocked.into());
198 |     }
200 |     lock_state.is_locked = 1;
201 |     lock_state.owner = ctx.accounts.leaf_owner.key();
202 |     lock_state.mint = ctx.accounts.nft_1.key();
```

As a result, attackers can provide arbitrary mint without owning the corresponding NFT and craft a "`lock_state`" for any mint.

2. Unbounded nft_1 in unlock_cnft()

```
/* defenders/src/lib.rs */
962 | pub struct UnlockCompressedNft<'info> {
983 |     /// CHECK
984 |     pub nft_1: AccountInfo<'info>,
985 |
986 |     #[account(init_if_needed, payer = payer, space = LockState::SPACE, seeds = [b"lock",
↪  nft_1.key().as_ref()], bump,
                  constraint = lock_state.owner.key() == payer.key() && lock_state.mint.key() ==
                   ↪  nft_1.key() @ ErrorCode::InvalidAction)]
987 |     pub lock_state: Box<Account<'info, LockState>>,
```

Similarly, "`unlock_cnft()`" has the same issue. "`nft_1`" is not constrained too.

3. nft_1 constraints

Even when the compressed NFT is not decompressed and the mint has not been created yet, its key can be deterministically computed as follows.

```
/* mpl-bubblegum-0.7.0/src/lib.rs */
322 | pub struct DecompressV1<'info> {
340 |     #[account(
341 |         mut,
342 |         seeds = [
343 |             ASSET_PREFIX.as_ref(),
344 |             voucher.merkle_tree.as_ref(),
345 |             voucher.leaf_schema.nonce().to_le_bytes().as_ref(),
346 |         ],
347 |         bump
348 |     )]
349 |     pub mint: UncheckedAccount<'info>,
```

It's recommended to validate "`nft_1`" in both instructions.

Resolution

This issue has been resolved.

DEFENDERS SMART CONTRACT

[P1-M-1] Unconstrained dest_ata token account owner

The token account owner of "dest_ata" is not validated. It should be "user_state". As a result, it's possible to perform a self transfer so the fund will not be locked, or send tokens to a wrong owner by mistake.

```
/* defenders/src/lib.rs */
063 | pub fn lock_spl(ctx: Context<LockUnlockSpl>, amount: u64) -> Result<()> {
065 |     token::transfer(
066 |         CpiContext::new(
067 |             ctx.accounts.token_program.to_account_info().clone(),
068 |             anchor_spl::token::Transfer {
069 |                 from: ctx.accounts.from_ata.to_account_info(),
070 |                 to: ctx.accounts.dest_ata.to_account_info(),
071 |                 authority: ctx.accounts.owner.to_account_info(),
072 |             },
073 |         ),
074 |         amount
075 |     )?;
077 |     Ok(())
078 | }
```

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

The "dest_ata" token account owner check has been added. However, it is recommended to also add a delegate check to ensure that the SPL token transferred cannot be withdrawn by others.

```
if ctx.accounts.dest_ata.delegate != COption::None || ctx.accounts.dest_ata.delegated_amount > 0 {
    return Err(ErrorCode::WrongTokenAccount.into());
}
```

DEFENDERS SMART CONTRACT

[P1-M-2] Unchecked amount in lock_solana and unlock_solana

The transfer amount is not checked so it's possible to transfer out the rent. When it happens, the "owner" and "user_state" may be unexpectedly closed.

```
/* defenders/src/lib.rs */

080 | pub fn lock_solana(ctx: Context<LockUnlockSolana>, amount: u64) -> Result<()> {
081 |     let user_state = &mut ctx.accounts.user_state;
082 |     let fee_transfer = anchor_lang::solana_program::system_instruction::transfer(
083 |         ctx.accounts.owner.key,
084 |         &ctx.accounts.user_state.key(),
085 |         amount,
086 |     );
087 |     solana_program::program::invoke(
088 |         &fee_transfer,
089 |         &[
090 |             ctx.accounts.owner.clone().to_account_info(),
091 |             ctx.accounts.user_state.to_account_info().clone(),
092 |             ctx.accounts.system_program.to_account_info().clone(),
093 |         ],
094 |     )?;
095 |     Ok(())
096 | }

099 | pub fn unlock_solana(ctx: Context<LockUnlockSolana>, amount: u64) -> Result<()> {
100 |
101 |     let user_state = &mut ctx.accounts.user_state;
102 |     let user_wallet = &mut ctx.accounts.user_wallet;
103 |
104 |     **user_state.to_account_info().try_borrow_mut_lamports()? -= amount;
105 |     **user_wallet.to_account_info().try_borrow_mut_lamports()? += amount;
106 |
107 |     Ok(())
108 | }
```

Resolution

The rent exemption check was added. This issue has been resolved.

DEFENDERS SMART CONTRACT
[P1-L-1] Bypass the TOTP

The "totp" signature is needed in core functions. For example

```
/* src/lib.rs */
972 | #[derive(Accounts)]
973 | pub struct LockUnlockNft<'info> {
975 |     #[account(mut)]
976 |     pub owner: Signer<'info>,
979 |     #[account(mut, constraint = totp.key() == user_state.totp.key() @ ErrorCode::WrongTOTP)]
980 |     pub totp: Signer<'info>,
```

The "totp" is specified during initialization and approved by both the user and the totp key holder. The totp is supposed to be signed by the defender's backend.

```
/* programs/defenders/src/lib.rs */
033 | pub fn init_user_state(ctx: Context<InitUserState>, bump: u8) -> Result<()> {
035 |     let user_state = &mut ctx.accounts.user_state;
037 |     user_state.totp = ctx.accounts.totp.key();
038 |     user_state.owner = ctx.accounts.owner.key();
039 |     user_state.bump = bump;
041 |     Ok(())
042 | }

837 | #[derive(Accounts)]
838 | pub struct InitUserState<'info> {
839 |     /// CHECK
840 |     #[account(mut)]
841 |     pub owner: Signer<'info>,
842 |
843 |     /// CHECK
844 |     #[account(mut)]
845 |     pub totp: Signer<'info>,
846 |
847 |     #[account(init, payer = owner, space = UserState::SPACE, seeds = [b"user",
↪  owner.key().as_ref()], bump)]
848 |     pub user_state: Box<Account<'info, UserState>>,
849 |
850 |     pub system_program: Program<'info, System>,
851 | }
```

However, the public key of the TOTP is not constrained during initialization. Consequently, users can employ two private keys to sign transactions, enabling them to bypass the 'must use TOTP' restriction.

In contrast to the expected scenario where the TOTP is signed by the 'hellomoon' endpoint, this

provides attackers with increased flexibility to interact with the program.

Resolution

The team acknowledged the finding and clarified that this is intentional. This approach allows users to host their own authentication system and set up their own TOTP signer.

DEFENDERS SMART CONTRACT

[P1-L-2] Integer overflow

The "totp" signature is needed in core functions. For example

The overflow check is not enabled. As a result, the arithmetic operations may overflow and lead to unwanted results. Consider enabling overflow check in Cargo.toml or use checked operators like "checked_add" instead.

```
[profile.release]
overflow-checks = true
```

Resolution

This issue has been resolved.

DEFENDERS SMART CONTRACT
[P1-I-1] CPI error handling

The newly added handler missed "?" at line 57.

```
/* defenders/src/lib.rs */
034 | pub fn unlock_spl(ctx: Context<LockUnlockSpl>, amount: u64) -> Result<()> {
046 |     let transfer_token = anchor_spl::token::transfer(
047 |         CpiContext::new_with_signer(
048 |             ctx.accounts.token_program.to_account_info(),
049 |             anchor_spl::token::Transfer {
050 |                 from: ctx.accounts.from_ata.to_account_info(),
051 |                 to: ctx.accounts.dest_ata.to_account_info(),
052 |                 authority: ctx.accounts.user_state.to_account_info(),
053 |             },
054 |             &[&user_pda_signer]
055 |         ),
056 |         amount
057 |     );
060 |     Ok(())
061 | }
```

Right now, if there are errors in CPI, the transaction will fail (https://docs.solanalabs.com/proposals/return-data#note-on-returning-errors).

So, this won't lead to major issues. However, since other CPIs end with "?;", it may be a good idea to follow the same practice.

Resolution

This issue has been resolved.

DEFENDERS FRONTEND

[P2-M-1] Transaction unconfirmed, causing users unable to lock/unlock

The defenders frontend determines whether a transaction is successfully executed by checking the success of "sendRawTransaction".

```
/* components/AuthCodePopup.tsx */
097 | try {
098 |   let signedTx = await connection.sendRawTransaction(serializedTx);
099 |   successCount += 1;
100 |   console.log(signedTx);
101 | } catch (e) {
102 |   console.log(e);
103 |   errorCount += 1;
104 | }
```

Due to the absence of "confirmTransaction", it is likely that the transaction has not been confirmed on the blockchain, yet the "successCount" has been incremented. The user received a notification of a successful transaction, despite the fact that the lock/unlock operation was not executed successfully. Consequently, this will result in the user being unable to lock or unlock.

Recommendations

It is recommended to call "confirmTransaction()" after "connection.sendRawTransaction":

```
/* components/ImagePopup.tsx */
078 | let signedTx = await connection.sendRawTransaction(serializedTx);
/* components/AuthCodePopup.tsx */
206 | let signedTx = await connection.sendRawTransaction(serializedTx);
/* components/AuthCodePopup.tsx */
098 | let signedTx = await connection.sendRawTransaction(serializedTx);
```

Resolution

This issue has been fixed.

DEFENDERS FRONTEND

[P2-M-2] Missing confirmTransaction check in lockUnlockToken

The Lock/Unlock Token functions are added. However, in line 420, the check for the transaction execution result is missing.

```
/* defenders-ui/components/AuthCodePopup.tsx */
315 | const lockUnlockToken = async () => {
417 |     let signedTxn = await wallet.signTransaction!(recoveredTransaction);
418 |     const serializedTx = signedTxn.serialize();
419 |
420 |     let signedTx = await connection.sendRawTransaction(serializedTx);
421 |
422 |   forceRefresh();
423 |   togglePopup();
424 |
425 |   toast({
426 |     title: `Token locked success.`,
427 |     description: ``,
428 |     status: "success",
429 |     duration: 5000,
430 |     isClosable: true,
431 |   });
```

Resolution

This issue has been resolved.

DEFENDERS FRONTEND

[P2-L-1] Use NFT whitelist to reduce security risks

There are two functions in "`utils/transactions.ts`": "`getNfts`" and "`getCnfts`", which retrieve all NFTs and access the "`json_uri`" or the "`URI`" specified in each NFT.

```
/* utils/transactions.ts */
146 | async function getCnfts(wallet: string) {
148 |   const url = "https://rpc.hellomoon.io/...";
149 |   let response = await axios.post(url, {
150 |     jsonrpc: "2.0",
151 |     id: "1",
152 |     method: "getAssetsByOwner",
153 |     params: {
154 |       ownerAddress: wallet.toString(),
155 |       limit: 1000,
156 |       page: 1,
157 |     },
158 |   });
160 |   for (let i = 0; i < response.data.result.items.length; i++) {
161 |     let nft = response.data.result.items[i];
162 |     if (nft.content.links.image === "") {
163 |       console.log("ALERT MISSING DATA");
164 |       console.log(nft);
165 |
166 |       let json = await (
167 |         await fetch(nft.content.json_uri, {
168 |           cache: "force-cache",
169 |         })
170 |       ).json();
171 |
172 |       let newImage = json.image;
173 |       response.data.result.items[i].content.links.image = newImage;
174 |     }
175 |   }
177 |   return response.data.result.items;
178 | }
```

Due to the absence of restrictions on the scope of NFTs, as well as on the "`json_uri`" and "`URI`", attackers can exploit this vulnerability by transferring carefully crafted NFTs to users. This can lead users to inadvertently access malicious links.

Resolution

The team acknowledge this finding and plans to educate users about this risk.

DEFENDERS FRONTEND

[P2-L-2] Incorrect error handling In linkAuthAccountToWallet

The "`linkAuthAccountToWallet`" function queries the create-account interface in the background and performs JSON parsing on the return value.

```
/* components/ImagePopup.tsx */
033 | const linkAuthAccountToWallet = async () => {
034 |   console.log("will be linking with code: ", totpCode);
036 |   let txRes : any = null;
038 |   try {
039 |     let fetchIx = await fetch(
040 |       // @ts-ignore: Object is possibly 'null'.
041 |       `/api/defendersHandler/${wallet.publicKey.toString()}`,
042 |       {
043 |         method: "POST", // or 'PUT' or 'PATCH' depending on your API
044 |         headers: {
045 |           "Content-Type": "application/json",
046 |           // Add any other headers if needed
047 |         },
048 |         body: JSON.stringify({
049 |           endpoint: "create-account",
050 |           code: totpCode,
051 |           wallet: wallet.publicKey.toString(),
052 |         }),
053 |       }
054 |     );
055 |     txRes = await fetchIx.json();
056 |   } catch (e) {
057 |     console.log(e);
058 |     toast({
059 |       title: `Invalid authentication code provided.`,
060 |       description: ``,
061 |       status: "error",
062 |       duration: 5000,
063 |       isClosable: true,
064 |     });
065 |     return;
066 |   }
068 |   console.log(txRes);
070 |   const recoveredTransaction = Transaction.from(
071 |     Buffer.from(txRes.data, "base64")
072 |   );
```

If a JSON deserialization error occurs, it throws an error; otherwise, it parses "`txRes.data`" into a transaction. However, "`txRes.data`" may not necessarily be a serialized transaction. The type of data can be either "`string`" or "`transactionToSignEncoded`".

```
/* pages/api/defendersHandler/[wallet].ts */
106 |   if (!success) {
107 |     return res.send({
108 |       success: false,
109 |       data: "Failed to send transaction.",
110 |     });
111 |   }
112 |
113 |   return res.send({
114 |     success: true,
115 |     data: transactionToSignEncoded,
116 |   });
117 | };
```

Therefore, there is a special case, where the "`json()`" successes but the "`Transaction.from()`" fails. However, the error generated by "`Transaction.from`" is not caught.

Resolution

This issue has been fixed.

DEFENDERS FRONTEND

[P2-L-3] Sanity check in the lock or unlock endpoints

The frontend calls the hellomoon lock/unlock endpoints to organize transactions with a "`mints`" parameter, which is of the array type. The RPC endpoint should perform appropriate sanity checks on this parameter. Upon testing, it was observed that the endpoint does not verify whether the "`mints`" is empty, nor does it check for duplicate entries in the "`mints`" array.

PoC

Save the exploit code into poc.js and run "`node poc.js 2 <Your totp> <Your Solana Address>`". Lock requests with duplicate mints will succeed.

```
const fetch = require("node-fetch");
const url = "https://rest-api.hellomoon.io/v0/hello-moon/defenders/lock";
const headers = {
  "User-Agent":
    "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/117.0",
  Accept: "application/json",
  "Accept-Language": "en-US,en;q=0.5",
  "Accept-Encoding": "gzip, deflate, br",
  "Content-Type": "application/json",
  authorization: "Bearer 2aac76c6-9590-400a-bfbb-1411c9716810",
  Connection: "keep-alive",
  "Sec-Fetch-Dest": "empty",
  "Sec-Fetch-Mode": "cors",
  "Sec-Fetch-Site": "cross-site",
  TE: "trailers",
};

const code = process.argv[3];

const mints = [];
const mintsLength = parseInt(process.argv[2]);
for (let i = 0; i < mintsLength; i++) {
  mints.push({
    mint: "226gfidaPwtU3WbBQYRgXrxN6cNZMSAjjoBWa5N6TsEF",
    nftType: "PNFT",
  });
}

const data =
  '{"code":"' +
  code.toString() +
  '","wallet":"' + process.argv[4] + '","mints":' + JSON.stringify(mints) + '}';
```

```
console.time("fetch " + mintsLength.toString() + "time");
fetch(url, {
  method: "POST",
  headers: headers,
  body: data,
})
  .then(async (response) => {
    if(response.status === 200){
      console.timeEnd("fetch " + mintsLength.toString() + "time");
      console.log(await response.json());
      return;
    }else {
      console.log(response.statusText);
    }
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

We attempted to have 16 processes simultaneously execute lock RPC calls for 10,000 mints, with each process taking approximately 30 seconds. Concurrently, we simulated a regular user accessing the RPC within this 30-second window. The user was not blocked, indicating that this defect does not result in a DoS.

Resolution

This issue has been fixed.

DEFENDERS FRONTEND

[P2-I-1] Uncaught control flow

While this does not introduce security issues, some exceptional branches have not been caught and handled during the program's execution process.

1. Uncaught message

In "`transactions.ts`", the determination of whether a user is registered is made by checking if "`infos.data.message === 'Not Found'`". If the user is not registered, an "`infos`" object is returned, populated with default values for each of its members.

```
/* utils/transactions.ts */
054 | if(infos.data.message === 'Not Found'){
055 |   infos = {
056 |     data: {
057 |         lockedNfts: [],
058 |         defendersLocked: 0,
059 |         points: 0,
060 |         timeStaked: 0,
061 |         totp: ""
062 |
063 |     }
064 |   };
065 | }
```

However, the condition "`infos.data.message === 'Forbidden'`" should also be handled similarly. Otherwise, the "`infos`" object, lacking a default value, will cause an exception due to the absence of a "`.length`" member.

This issue will arise in the subsequent evaluation of "`infos.data.lockedNfts.length > 0`", potentially leading to a crash during execution.

It is recommended to change the condition in line 54:

```
if(infos.data.message === 'Not Found' || infos.data.message == 'Forbidden') {
```

2. json() error catch

At line 100 in "`defendersHandler/[wallet].ts`", the json function lacks exception handling.

```
/* pages/api/defendersHandler/[wallet].ts */
098 | let response = await fetch(url, options)
099 |   .then((response) => {
100 |     return response.json()})
101 |   .then(async (data) => {
102 |     transactionToSignEncoded = data;
103 |     console.log(data);
104 |   });
```

It is recommended to wrap it with a try-catch statement.

Resolution

This issue has been resolved.

DEFENDERS FRONTEND

[P2-I-2] Hellomoon RPC should throw error friendly

The defenders handler invokes the 'hellomoon' RPC, aiming to retrieve the transaction that requires signing.

```
/* pages/api/defendersHandler/[wallet].ts */
101 | let response = await fetch(url, options)
102 |   .then((response) => response.json())
103 |   .then(async (data) => {
104 |     transactionToSignEncoded = data;
105 |
106 |     console.log(data)
107 |   });
108 |
```

When parameter errors prevent the proper organization of transactions, the RPC should return a 500 error or other error messages. However, testing revealed that the defenders' hellomoon RPC, when encountering an error, does not return a 500 or other error codes. Instead, it returns HTTP 204. This results in a lack of appropriate prompts in the frontend.

Consider setting "`response.status`" to 500 when encountering an error in the hellomoon rpc. In addition, add a "`.catch`" statement at line 107 in "`pages/api/defendersHandler/[wallet].ts`" to handle network errors.

Resolution

This issue has been fixed.

Since the hellomoon is not in the audit scope, we recommended that all the return values for hellomoon RPC errors have been changed to 500.

DEFENDERS FRONTEND

[P2-I-3] Typos

At lines 124 and 260, should it be "`errorCount++`" instead of "`successCount--`"?

```
/* defenders-ui/components/AuthCodePopup.tsx */
111 | let successCount = 0;
112 | let errorCount = 0;
114 | const transactionPromises = signedTransactions.map((txn: { serialize: () => any; }) => {
115 |   const serializedTx = txn.serialize();
116 |   return connection.sendRawTransaction(serializedTx)
117 |         .then(signedTx => {
119 |             return connection.confirmTransaction(signedTx, "confirmed")
120 |                 .then(confirmed => {
121 |                     if (confirmed) {
122 |                         successCount++;
123 |                     } else {
124 |                         successCount--;
125 |                     }
126 |                 });
127 |         })
128 |         .catch(e => {
129 |             console.log(e);
130 |             errorCount++;
131 |             throw e;
132 |         });
133 | });

247 | let successCount = 0;
248 | let errorCount = 0;
250 | const transactionPromises = signedTransactions.map((txn: { serialize: () => any; }) => {
251 |   const serializedTx = txn.serialize();
252 |   return connection.sendRawTransaction(serializedTx)
253 |       .then(signedTx => {
255 |           return connection.confirmTransaction(signedTx, "confirmed")
256 |               .then(confirmed => {
257 |                   if (confirmed) {
258 |                       successCount++;
259 |                   } else {
260 |                       successCount--;
261 |                   }
262 |               });
263 |       })
264 |       .catch(e => {
265 |           console.log(e);
266 |           errorCount++;
267 |           throw e;
268 |       });
269 | });
```

In addition, at line 282, "`errorCount`" should be "`successCount`".

```
/* defenders-ui/components/AuthCodePopup.tsx */
280 | if (successCount > 0) {
281 |   toast({
282 |     title: `Processed ${errorCount} transactions successfuly.`,
283 |     description: ``,
284 |     status: "success",
285 |     duration: 5000,
286 |     isClosable: true,
287 |   });
288 | }
```

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

Appendix: Methodology and Scope of Work

The Sec3 (formerly Soteria) audit team, which consists of Computer Science professors and industrial researchers with extensive experience in smart contract security, program analysis, testing and formal verification, performed a comprehensive manual code review, software static analysis and penetration testing.

Assisted by the Sec3 Scanner developed in-house, the audit team particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Customer Company Name (the "Client"). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

Founded by leading academics in the field of software security and senior industrial veterans, Sec3 (formerly Soteria) is a leading blockchain security company. We are also building sophisticated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.

Sec3™