VRust

Security Assessment

O2Lab VRust Team

11/02/2022 20:52:10

Contents

Summary

This report has been prepared for O2Lab VRust Team to discover issues and vulnerabilities in the source code of the O2Lab VRust Team project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques. The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	O2Lab VRust Team
Platform	Ethereum
Language	Solana
Crate	nft_bridge
GitHub Location	https://github.com/parasol-aser/vrust
sha256	Unknown

Audit Summary

Delivery Date	11/02/2022
Audit Methodology	Static Analysis
Key Components	

Vulnerability Summary

Vulnerability Level	Total
Critical	7
Major	0
Medium	0
Minor	0
Informational	0
Discussion	0

Findings

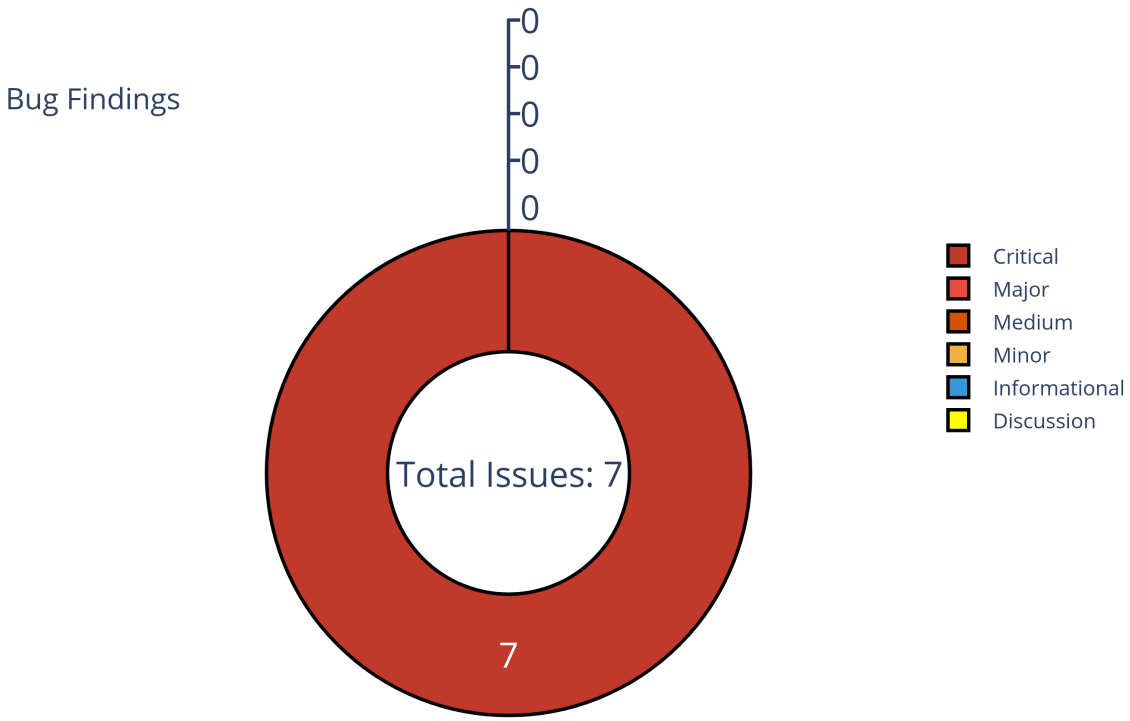

Figure 1: Findings

Finding Statistic

Category	Count
MissingKeyCheck	1
CrossProgramInvocation	6

ID	Category	Severity	Status
0	MissingKeyCheck	Critical	UnResolved
1	CrossProgramInvocation	Critical	UnResolved
2	CrossProgramInvocation	Critical	UnResolved
3	CrossProgramInvocation	Critical	UnResolved
4	CrossProgramInvocation	Critical	UnResolved
5	CrossProgramInvocation	Critical	UnResolved
6	CrossProgramInvocation	Critical	UnResolved

Issue: 0: MissingKeyCheck

Category	Severity	Status
MissingKeyCheck	Critical	UnResolved

- Location

/home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/processors/peel.rs:211:22:211:50

```
211 ctx.info().lamports.borrow()
212
```

- Code Context

– Function Definition:

```
192 fn peel<I>(ctx: &'c mut Context<'a, 'b, 'c, I>) -> Result<Self>
193
```

Vulnerability at Line: 202

```
197         }
198
199         // If we're initializing the type, we should emit system/rent as
            ↪  deps.
200         let (initialized, data): (bool, T) = match IsInitialized {
201             AccountState::Uninitialized => {
202                 if **ctx.info().lamports.borrow() != 0 {
203                     return
                        ↪  Err(SolitaireError::AlreadyInitialized(*ctx.info().key));
204                 }
205                 (false, T::default())
206             }
207
```

Other Use Case for Variable: ctx.info().lamports.borrow()

```
211                 if **ctx.info().lamports.borrow() == 0 {
```

- Call Stack

```
fn entrypoint(){// /home/yifei/.cargo/registry/src/github.com-
↪  1ecc6299db9ec823/solana-program-1.7.0/src/entrypoint.rs:46:9: 53:10
↪  }
    fn instruction::solitaire(){//
    ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macros.rs
    ↪  108:14 }
        fn instruction::dispatch(){//
        ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macro
        ↪  99:14 }
            fn instruction::CompleteNative::execute(){//
            ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/m
            ↪  74:22 }
                fn <api::complete_transfer::CompleteNative<'b> as
                ↪  solitaire::FromAccounts<'a, 'b, 'c>>::from(){//
                ↪  program/src/api/complete_transfer.rs:42:10: 42:22 }
                    fn <bridge::vaa::ClaimableVAA<'b, T> as
                    ↪  solitaire::Peel<'a, 'b, 'c>>::peel(){//
                    ↪  /home/yifei/open/vrust/examples2/wormhole/solana/bridge/prog
                    ↪  148:22 }
                        fn <bridge::vaa::ClaimableVAA<'b, T> as
                        ↪  solitaire::FromAccounts<'a, 'b,
                        ↪  'c>>::from(){//
                        ↪  /home/yifei/open/vrust/examples2/wormhole/solana/bridge/
                        ↪  148:22 }
        fn <bridge::PayloadMessage<'b, T> as solitaire::Peel<'a, 'b,
        ↪  'c>>::peel(){//
        ↪  /home/yifei/open/vrust/examples2/wormhole/solana/bridge/program/src/vaa.rs:11
        ↪  124:6 }
            fn <solitaire::Data<'b, T, IsInitialized> as
            ↪  solitaire::Peel<'a, 'b, 'c>>::peel(){//
            ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/p
            ↪  236:6 }

```

- description:
- link:
- alleviation:

Issue: 1: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
program/src/api/complete_transfer.rs
```

- Code Context

```
81  pub fn complete_native(
82      ctx: &ExecutionContext,
83      accs: &mut CompleteNative,
84      _data: CompleteNativeData,
85  ) -> Result<()> {
86      // Verify the chain registration
87      let derivation_data: EndpointDerivationData = (&*accs).into();
88      accs.chain_registration
89          .verify_derivation(ctx.program_id, &derivation_data)?;
90
91      // Verify that the custody account is derived correctly
92      let derivation_data: CustodyAccountDerivationData = (&*accs).into();
93      accs.custody
94          .verify_derivation(ctx.program_id, &derivation_data)?;
95
96      // Verify mints
97      if *accs.mint.info().key != accs.custody.mint {
98          return Err(InvalidMint.into());
99      }
100     if *accs.custody_signer.key != accs.custody.owner {
101         return Err(WrongAccountOwner.into());
102     }
103
104     // Verify VAA
105     // Please refer to transfer.rs for why the token id is used to store
        ↪ the mint
106     if accs.vaa.token_address != [1u8; 32] {
```

```
107         return Err(InvalidMint.into());
108     }
109     let mut token_id_bytes = [0u8; 32];
110     accs.vaa.token_id.to_big_endian(&mut token_id_bytes);
111     if token_id_bytes != accs.mint.info().key.to_bytes() {
112         return Err(InvalidMint.into());
113     }
114     if accs.vaa.token_chain != CHAIN_ID_SOLANA {
115         return Err(InvalidChain.into());
116     }
117     if accs.vaa.to_chain != CHAIN_ID_SOLANA {
118         return Err(InvalidChain.into());
119     }
120     if accs.vaa.to != accs.to.info().key.to_bytes() {
121         return Err(InvalidRecipient.into());
122     }
123
124     // Prevent vaa double signing
125     accs.vaa.verify(ctx.program_id)?;
126     accs.vaa.claim(ctx, accs.payer.key)?;
127
128     if !accs.to.is_initialized() {
129         let associated_addr =
          ↪  spl_associated_token_account::get_associated_token_address(
130             accs.to_authority.info().key,
131             accs.mint.info().key,
132         );
133         if *accs.to.info().key != associated_addr {
134             return Err(InvalidAssociatedAccount.into());
135         }
136         // Create associated token account
137         let ix =
          ↪  spl_associated_token_account::create_associated_token_account(
138             accs.payer.info().key,
139             accs.to_authority.info().key,
140             accs.mint.info().key,
141         );
142         invoke(&ix, ctx.accounts)?;
143     } else if *accs.mint.info().key != accs.to.mint {
144         return Err(InvalidMint.into());
145     }
146
```

```
147     // Transfer tokens
148     let transfer_ix = spl_token::instruction::transfer(
149         &spl_token::id(),
150         accs.custody.info().key,
151         accs.to.info().key,
152         accs.custody_signer.key,
153         &[],
154         1,
155     )?;
156     invoke_seeded(&transfer_ix, ctx, &accs.custody_signer, None)?;
157 
158     Ok(())
159 }
160 
```

- Call Stack

```
1  fn entrypoint(){// /home/yifei/.cargo/registry/src/github.com-
   ↪  1ecc6299db9ec823/solana-program-1.7.0/src/entrypoint.rs:46:9: 53:10
   ↪  }
2      fn instruction::solitaire(){//
       ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macros.rs
       ↪  108:14 }
3          fn instruction::dispatch(){//
           ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macro
           ↪  99:14 }
4              fn instruction::CompleteNative::execute(){//
               ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/m
               ↪  74:22 }
5                  fn api::complete_transfer::complete_native(){//
                   ↪  program/src/api/complete_transfer.rs:81:1: 159:2 }
6 
```

- description:
- link:
- alleviation:

Issue: 2: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
program/src/api/complete_transfer.rs
```

- Code Context

```
212 pub fn complete_wrapped(
213     ctx: &ExecutionContext,
214     accs: &mut CompleteWrapped,
215     _data: CompleteWrappedData,
216 ) -> Result<()> {
217     use bstr::ByteSlice;
218 
219     // Verify the chain registration
220     let derivation_data: EndpointDerivationData = (&*accs).into();
221     accs.chain_registration
222         .verify_derivation(ctx.program_id, &derivation_data)?;
223 
224     // Verify mint
225     let derivation_data: WrappedDerivationData = (&*accs).into();
226     accs.mint
227         .verify_derivation(ctx.program_id, &derivation_data)?;
228 
229     // Verify VAA
230     if accs.vaa.to_chain != CHAIN_ID_SOLANA {
231         return Err(InvalidChain.into());
232     }
233     if accs.vaa.to != accs.to.info().key.to_bytes() {
234         return Err(InvalidRecipient.into());
235     }
236 
237     accs.vaa.verify(ctx.program_id)?;
238     accs.vaa.claim(ctx, accs.payer.key)?;
```

```
239
240     // Initialize the NFT if it doesn't already exist
241     if !accs.meta.is_initialized() {
242         // Create mint account
243         accs.mint
244             .create(&((&*accs).into()), ctx, accs.payer.key, Exempt)?;
245
246         // Initialize mint
247         let init_ix = spl_token::instruction::initialize_mint(
248             &spl_token::id(),
249             accs.mint.info().key,
250             accs.mint_authority.key,
251             None,
252             0,
253         )?;
254         invoke_signed(&init_ix, ctx.accounts, &[])?;
255
256         // Create meta account
257         accs.meta
258             .create(&((&*accs).into()), ctx, accs.payer.key, Exempt)?;
259
260         // Populate meta account
261         accs.meta.chain = accs.vaa.token_chain;
262         accs.meta.token_address = accs.vaa.token_address;
263         accs.meta.token_id = accs.vaa.token_id.0;
264     }
265
266     if !accs.to.is_initialized() {
267         let associated_addr =
            ↪ spl_associated_token_account::get_associated_token_address(
268             accs.to_authority.info().key,
269             accs.mint.info().key,
270         );
271         if *accs.to.info().key != associated_addr {
272             return Err(InvalidAssociatedAccount.into());
273         }
274         // Create associated token account
275         let ix =
            ↪ spl_associated_token_account::create_associated_token_account(
276             accs.payer.info().key,
277             accs.to_authority.info().key,
278             accs.mint.info().key,
```

```
279         );
280         invoke_signed(&ix, ctx.accounts, &[])?;
281     } else if *accs.mint.info().key != accs.to.mint {
282         return Err(InvalidMint.into());
283     }
284 
285     // Mint tokens
286     let mint_ix = spl_token::instruction::mint_to(
287         &spl_token::id(),
288         accs.mint.info().key,
289         accs.to.info().key,
290         accs.mint_authority.key,
291         &[],
292         1,
293     )?;
294     invoke_seeded(&mint_ix, ctx, &accs.mint_authority, None)?;
295 
296     Ok(())
297 }
298 
```

- Call Stack

```
1  fn entrypoint(){// /home/yifei/.cargo/registry/src/github.com-
   ↪  1ecc6299db9ec823/solana-program-1.7.0/src/entrypoint.rs:46:9: 53:10
   ↪  }
2      fn instruction::solitaire(){//
       ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macros.rs
       ↪  108:14 }
3          fn instruction::dispatch(){//
           ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macro
           ↪  99:14 }
4              fn instruction::CompleteWrapped::execute(){//
               ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/m
               ↪  74:22 }
5                      fn api::complete_transfer::complete_wrapped(){//
                       ↪  program/src/api/complete_transfer.rs:212:1: 297:2 }
6  
```

- description:
- link:

- alleviation:

Issue: 3: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
program/src/api/complete_transfer.rs
```

- Code Context

```
212 pub fn complete_wrapped(
213     ctx: &ExecutionContext,
214     accs: &mut CompleteWrapped,
215     _data: CompleteWrappedData,
216 ) -> Result<()> {
217     use bstr::ByteSlice;
218 
219     // Verify the chain registration
220     let derivation_data: EndpointDerivationData = (&*accs).into();
221     accs.chain_registration
222         .verify_derivation(ctx.program_id, &derivation_data)?;
223 
224     // Verify mint
225     let derivation_data: WrappedDerivationData = (&*accs).into();
226     accs.mint
227         .verify_derivation(ctx.program_id, &derivation_data)?;
228 
229     // Verify VAA
230     if accs.vaa.to_chain != CHAIN_ID_SOLANA {
231         return Err(InvalidChain.into());
232     }
233     if accs.vaa.to != accs.to.info().key.to_bytes() {
234         return Err(InvalidRecipient.into());
235     }
236 
237     accs.vaa.verify(ctx.program_id)?;
238     accs.vaa.claim(ctx, accs.payer.key)?;
```

```
239
240     // Initialize the NFT if it doesn't already exist
241     if !accs.meta.is_initialized() {
242         // Create mint account
243         accs.mint
244             .create(&((&*accs).into()), ctx, accs.payer.key, Exempt)?;
245
246         // Initialize mint
247         let init_ix = spl_token::instruction::initialize_mint(
248             &spl_token::id(),
249             accs.mint.info().key,
250             accs.mint_authority.key,
251             None,
252             0,
253         )?;
254         invoke_signed(&init_ix, ctx.accounts, &[])?;
255
256         // Create meta account
257         accs.meta
258             .create(&((&*accs).into()), ctx, accs.payer.key, Exempt)?;
259
260         // Populate meta account
261         accs.meta.chain = accs.vaa.token_chain;
262         accs.meta.token_address = accs.vaa.token_address;
263         accs.meta.token_id = accs.vaa.token_id.0;
264     }
265
266     if !accs.to.is_initialized() {
267         let associated_addr =
          ↪  spl_associated_token_account::get_associated_token_address(
268             accs.to_authority.info().key,
269             accs.mint.info().key,
270         );
271         if *accs.to.info().key != associated_addr {
272             return Err(InvalidAssociatedAccount.into());
273         }
274         // Create associated token account
275         let ix =
          ↪  spl_associated_token_account::create_associated_token_account(
276             accs.payer.info().key,
277             accs.to_authority.info().key,
278             accs.mint.info().key,
```

```
279         );
280         invoke_signed(&ix, ctx.accounts, &[])?;
281     } else if *accs.mint.info().key != accs.to.mint {
282         return Err(InvalidMint.into());
283     }
284 
285     // Mint tokens
286     let mint_ix = spl_token::instruction::mint_to(
287         &spl_token::id(),
288         accs.mint.info().key,
289         accs.to.info().key,
290         accs.mint_authority.key,
291         &[],
292         1,
293     )?;
294     invoke_seeded(&mint_ix, ctx, &accs.mint_authority, None)?;
295 
296     Ok(())
297 }
298 
```

- Call Stack

```
1  fn entrypoint(){// /home/yifei/.cargo/registry/src/github.com-
   ↪  1ecc6299db9ec823/solana-program-1.7.0/src/entrypoint.rs:46:9: 53:10
   ↪  }
2      fn instruction::solitaire(){//
       ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macros.rs
       ↪  108:14 }
3          fn instruction::dispatch(){//
           ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macro
           ↪  99:14 }
4              fn instruction::CompleteWrapped::execute(){//
               ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/m
               ↪  74:22 }
5                      fn api::complete_transfer::complete_wrapped(){//
                       ↪  program/src/api/complete_transfer.rs:212:1: 297:2 }
6  
```

- description:
- link:

- alleviation:

Issue: 4: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
program/src/api/transfer.rs
```

- Code Context

```
290 pub fn transfer_wrapped(
291     ctx: &ExecutionContext,
292     accs: &mut TransferWrapped,
293     data: TransferWrappedData,
294 ) -> Result<()> {
295     // Verify that the from account is owned by the from_owner
296     if &accs.from.owner != accs.from_owner.key {
297         return Err(WrongAccountOwner.into());
298     }
299 
300     // Verify mints
301     if accs.mint.info().key != &accs.from.mint {
302         return Err(TokenBridgeError::InvalidMint.into());
303     }
304 
305     // Verify that meta is correct
306     let derivation_data: WrappedMetaDerivationData = (&*accs).into();
307     accs.wrapped_meta
308         .verify_derivation(ctx.program_id, &derivation_data)?;
309 
310     // Burn tokens
311     let burn_ix = spl_token::instruction::burn(
312         &spl_token::id(),
313         accs.from.info().key,
314         accs.mint.info().key,
315         accs.authority_signer.key,
316         &[],
```

```
317         1,
318     )?;
319     invoke_seeded(&burn_ix, ctx, &accs.authority_signer, None)?;
320 
321     // Pay fee
322     let transfer_ix = solana_program::system_instruction::transfer(
323         accs.payer.key,
324         accs.fee_collector.key,
325         accs.bridge.config.fee,
326     );
327 
328     invoke(&transfer_ix, ctx.accounts)?;
329 
330     // Enfoce wrapped meta to be uninitialized.
331     let derivation_data: WrappedMetaDerivationData = (&*accs).into();
332     accs.wrapped_meta
333         .verify_derivation(ctx.program_id, &derivation_data)?;
334 
335     // Token must have metadata
336     if accs.spl_metadata.data_is_empty() {
337         return Err(TokenNotNFT.into());
338     }
339 
340     let derivation_data: SplTokenMetaDerivationData = (&*accs).into();
341     accs.spl_metadata
342         .verify_derivation(&spl_token_metadata::id(), &derivation_data)?;
343 
344     if *accs.spl_metadata.owner != spl_token_metadata::id() {
345         return Err(WrongAccountOwner.into());
346     }
347 
348     let metadata: Metadata =
349         Meta-
            ↪ data::from_account_info(accs.spl_metadata.info()).ok_or(InvalidMetadata)?;
350 
351     // Post message
352     let payload = PayloadTransfer {
353         token_address: accs.wrapped_meta.token_address,
354         token_chain: accs.wrapped_meta.chain,
355         token_id: U256(accs.wrapped_meta.token_id),
356         to: data.target_address,
357         to_chain: data.target_chain,
```

```
358         symbol: metadata.data.symbol,
359         name: metadata.data.name,
360         uri: metadata.data.uri,
361     };
362     let params = (
363         bridge::instruction::Instruction::PostMessage,
364         PostMessageData {
365             nonce: data.nonce,
366             payload: payload.try_to_vec()?,
367             consistency_level: ConsistencyLevel::Finalized,
368         },
369     );
370 
371     let ix = Instruction::new_with_bytes(
372         accs.config.wormhole_bridge,
373         params.try_to_vec()?.as_slice(),
374         vec![
375             AccountMeta::new(*accs.bridge.info().key, false),
376             AccountMeta::new(*accs.message.key, true),
377             AccountMeta::new_readonly(*accs.emitter.key, true),
378             AccountMeta::new(*accs.sequence.key, false),
379             AccountMeta::new(*accs.payer.key, true),
380             AccountMeta::new(*accs.fee_collector.key, false),
381             AccountMeta::new_readonly(*accs.clock.info().key, false),
382             AccountMeta::new_readonly(solana_program::system_program::id(),
                ↪  false),
383             AccountMeta::new_readonly(solana_program::sysvar::rent::ID,
                ↪  false),
384         ],
385     );
386     invoke_seeded(&ix, ctx, &accs.emitter, None)?;
387 
388     Ok(())
389 }
390 
```

- Call Stack

```
1  fn entrypoint(){// /home/yifei/.cargo/registry/src/github.com-
   ↪  1ecc6299db9ec823/solana-program-1.7.0/src/entrypoint.rs:46:9: 53:10
   ↪  }
2      fn instruction::solitaire(){//
       ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macros.rs
       ↪  108:14 }
```

```
3         fn instruction::dispatch(){//
          ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macro
          ↪  99:14 }
4             fn instruction::TransferWrapped::execute(){//
               ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/m
               ↪  74:22 }
5                 fn api::transfer::transfer_wrapped(){//
                   ↪  program/src/api/transfer.rs:290:1: 389:2 }
6
```

- description:
- link:
- alleviation:

Issue: 5: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
program/src/api/transfer.rs
```

- Code Context

```
119 pub fn transfer_native(
120     ctx: &ExecutionContext,
121     accs: &mut TransferNative,
122     data: TransferNativeData,
123 ) -> Result<()> {
124     // Verify that the custody account is derived correctly
125     let derivation_data: CustodyAccountDerivationData = (&*accs).into();
126     accs.custody
127         .verify_derivation(ctx.program_id, &derivation_data)?;
128 
129     let derivation_data: SplTokenMetaDerivationData = (&*accs).into();
130     accs.spl_metadata
131         .verify_derivation(&spl_token_metadata::id(), &derivation_data)?;
132 
133     // Verify mints
134     if accs.from.mint != *accs.mint.info().key {
135         return Err(TokenBridgeError::InvalidMint.into());
136     }
137 
138     // Token must have metadata
139     if accs.spl_metadata.data_is_empty() {
140         return Err(TokenNotNFT.into());
141     }
142 
143     if *accs.spl_metadata.owner != spl_token_metadata::id() {
144         return Err(WrongAccountOwner.into());
145     }
```

```
146
147     // Verify that the token is not a wrapped token
148     if let COption::Some(mint_authority) = accs.mint.mint_authority {
149         if mint_authority == MintSigner::key(None, ctx.program_id) {
150             return Err(TokenBridgeError::TokenNotNative.into());
151         }
152     }
153
154     if !accs.custody.is_initialized() {
155         accs.custody
156             .create(&(&*accs).into(), ctx, accs.payer.key, Exempt)?;
157
158         let init_ix = spl_token::instruction::initialize_account(
159             &spl_token::id(),
160             accs.custody.info().key,
161             accs.mint.info().key,
162             accs.custody_signer.key,
163         )?;
164         invoke_signed(&init_ix, ctx.accounts, &[])?;
165     }
166
167     // Transfer tokens
168     let transfer_ix = spl_token::instruction::transfer(
169         &spl_token::id(),
170         accs.from.info().key,
171         accs.custody.info().key,
172         accs.authority_signer.key,
173         &[],
174         1,
175     )?;
176     invoke_seeded(&transfer_ix, ctx, &accs.authority_signer, None)?;
177
178     // Pay fee
179     let transfer_ix = solana_program::system_instruction::transfer(
180         accs.payer.key,
181         accs.fee_collector.key,
182         accs.bridge.config.fee,
183     );
184     invoke(&transfer_ix, ctx.accounts)?;
185
186     let metadata: Metadata =
187         Meta-
          ↪ data::from_account_info(accs.spl_metadata.info()).ok_or(InvalidMetadata)?;
```

```
188
189     // Post message
190     // Given there is no tokenID equivalent on Solana and each distinct
        ↪  token address is translated
191     // into a new contract on EVM based chains (which is costly), we use a
        ↪  static token_address
192     // and encode the mint in the token_id.
193     let payload = PayloadTransfer {
194         token_address: [1u8; 32],
195         token_chain: 1,
196         to: data.target_address,
197         to_chain: data.target_chain,
198         symbol: metadata.data.symbol,
199         name: metadata.data.name,
200         uri: metadata.data.uri,
201         token_id: U256::from_big_endian(&accs.mint.info().key.to_bytes()),
202     };
203     let params = (
204         bridge::instruction::Instruction::PostMessage,
205         PostMessageData {
206             nonce: data.nonce,
207             payload: payload.try_to_vec()?,
208             consistency_level: ConsistencyLevel::Finalized,
209         },
210     );
211
212     let ix = Instruction::new_with_bytes(
213         accs.config.wormhole_bridge,
214         params.try_to_vec()?.as_slice(),
215         vec![
216             AccountMeta::new(*accs.bridge.info().key, false),
217             AccountMeta::new(*accs.message.key, true),
218             AccountMeta::new_readonly(*accs.emitter.key, true),
219             AccountMeta::new(*accs.sequence.key, false),
220             AccountMeta::new(*accs.payer.key, true),
221             AccountMeta::new(*accs.fee_collector.key, false),
222             AccountMeta::new_readonly(*accs.clock.info().key, false),
223             AccountMeta::new_readonly(solana_program::system_program::id(),
                ↪  false),
224             AccountMeta::new_readonly(solana_program::sysvar::rent::ID,
                ↪  false),
225         ],
```

```
226     );
227     invoke_seeded(&ix, ctx, &accs.emitter, None)?;
228 
229     Ok(())
230 }
231 
```

- Call Stack

```
1  fn entrypoint(){// /home/yifei/.cargo/registry/src/github.com-
   ↪  1ecc6299db9ec823/solana-program-1.7.0/src/entrypoint.rs:46:9: 53:10
   ↪  }
2      fn instruction::solitaire(){//
       ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macros.rs
       ↪  108:14 }
3          fn instruction::dispatch(){//
           ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macro
           ↪  99:14 }
4              fn instruction::TransferNative::execute(){//
               ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/m
               ↪  74:22 }
5                  fn api::transfer::transfer_native(){//
                   ↪  program/src/api/transfer.rs:119:1: 230:2 }
6 
```

- description:
- link:
- alleviation:

Issue: 6: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
program/src/api/transfer.rs
```

- Code Context

```
119 pub fn transfer_native(
120     ctx: &ExecutionContext,
121     accs: &mut TransferNative,
122     data: TransferNativeData,
123 ) -> Result<()> {
124     // Verify that the custody account is derived correctly
125     let derivation_data: CustodyAccountDerivationData = (&*accs).into();
126     accs.custody
127         .verify_derivation(ctx.program_id, &derivation_data)?;
128 
129     let derivation_data: SplTokenMetaDerivationData = (&*accs).into();
130     accs.spl_metadata
131         .verify_derivation(&spl_token_metadata::id(), &derivation_data)?;
132 
133     // Verify mints
134     if accs.from.mint != *accs.mint.info().key {
135         return Err(TokenBridgeError::InvalidMint.into());
136     }
137 
138     // Token must have metadata
139     if accs.spl_metadata.data_is_empty() {
140         return Err(TokenNotNFT.into());
141     }
142 
143     if *accs.spl_metadata.owner != spl_token_metadata::id() {
144         return Err(WrongAccountOwner.into());
145     }
```

```
146
147     // Verify that the token is not a wrapped token
148     if let COption::Some(mint_authority) = accs.mint.mint_authority {
149         if mint_authority == MintSigner::key(None, ctx.program_id) {
150             return Err(TokenBridgeError::TokenNotNative.into());
151         }
152     }
153
154     if !accs.custody.is_initialized() {
155         accs.custody
156             .create(&(&*accs).into(), ctx, accs.payer.key, Exempt)?;
157
158         let init_ix = spl_token::instruction::initialize_account(
159             &spl_token::id(),
160             accs.custody.info().key,
161             accs.mint.info().key,
162             accs.custody_signer.key,
163         )?;
164         invoke_signed(&init_ix, ctx.accounts, &[])?;
165     }
166
167     // Transfer tokens
168     let transfer_ix = spl_token::instruction::transfer(
169         &spl_token::id(),
170         accs.from.info().key,
171         accs.custody.info().key,
172         accs.authority_signer.key,
173         &[],
174         1,
175     )?;
176     invoke_seeded(&transfer_ix, ctx, &accs.authority_signer, None)?;
177
178     // Pay fee
179     let transfer_ix = solana_program::system_instruction::transfer(
180         accs.payer.key,
181         accs.fee_collector.key,
182         accs.bridge.config.fee,
183     );
184     invoke(&transfer_ix, ctx.accounts)?;
185
186     let metadata: Metadata =
187         Meta-
          ↪ data::from_account_info(accs.spl_metadata.info()).ok_or(InvalidMetadata)?;
```

```
188
189     // Post message
190     // Given there is no tokenID equivalent on Solana and each distinct
        ↪  token address is translated
191     // into a new contract on EVM based chains (which is costly), we use a
        ↪  static token_address
192     // and encode the mint in the token_id.
193     let payload = PayloadTransfer {
194         token_address: [1u8; 32],
195         token_chain: 1,
196         to: data.target_address,
197         to_chain: data.target_chain,
198         symbol: metadata.data.symbol,
199         name: metadata.data.name,
200         uri: metadata.data.uri,
201         token_id: U256::from_big_endian(&accs.mint.info().key.to_bytes()),
202     };
203     let params = (
204         bridge::instruction::Instruction::PostMessage,
205         PostMessageData {
206             nonce: data.nonce,
207             payload: payload.try_to_vec()?,
208             consistency_level: ConsistencyLevel::Finalized,
209         },
210     );
211
212     let ix = Instruction::new_with_bytes(
213         accs.config.wormhole_bridge,
214         params.try_to_vec()?.as_slice(),
215         vec![
216             AccountMeta::new(*accs.bridge.info().key, false),
217             AccountMeta::new(*accs.message.key, true),
218             AccountMeta::new_readonly(*accs.emitter.key, true),
219             AccountMeta::new(*accs.sequence.key, false),
220             AccountMeta::new(*accs.payer.key, true),
221             AccountMeta::new(*accs.fee_collector.key, false),
222             AccountMeta::new_readonly(*accs.clock.info().key, false),
223             AccountMeta::new_readonly(solana_program::system_program::id(),
                ↪  false),
224             AccountMeta::new_readonly(solana_program::sysvar::rent::ID,
                ↪  false),
225         ],
```

```
226     );
227     invoke_seeded(&ix, ctx, &accs.emitter, None)?;
228 
229     Ok(())
230 }
231 
```

- Call Stack

```
1  fn entrypoint(){// /home/yifei/.cargo/registry/src/github.com-
   ↪  1ecc6299db9ec823/solana-program-1.7.0/src/entrypoint.rs:46:9: 53:10
   ↪  }
2      fn instruction::solitaire(){//
       ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macros.rs
       ↪  108:14 }
3          fn instruction::dispatch(){//
           ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/macro
           ↪  99:14 }
4              fn instruction::TransferNative::execute(){//
               ↪  /home/yifei/open/vrust/examples2/wormhole/solana/solitaire/program/src/m
               ↪  74:22 }
5                  fn api::transfer::transfer_native(){//
                   ↪  program/src/api/transfer.rs:119:1: 230:2 }
6 
```

- description:
- link:
- alleviation:

Appendix

Copied from https://leaderboard.certik.io/projects/aave

Finding Categories

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Checksum Calculation Method

The “Checksum” field in the “Audit Scope” section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux “sha256sum” command against the target file.

Disclaimer

Copied from https://leaderboard.certik.io/projects/aave

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK’s prior written consent in each instance.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.