Competitive Security Assessment

Hajime

Mar 27th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3′s Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.

• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.

• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.

• Verify the code base is compliant with the most up-to-date industry standards and security best practices.

• Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

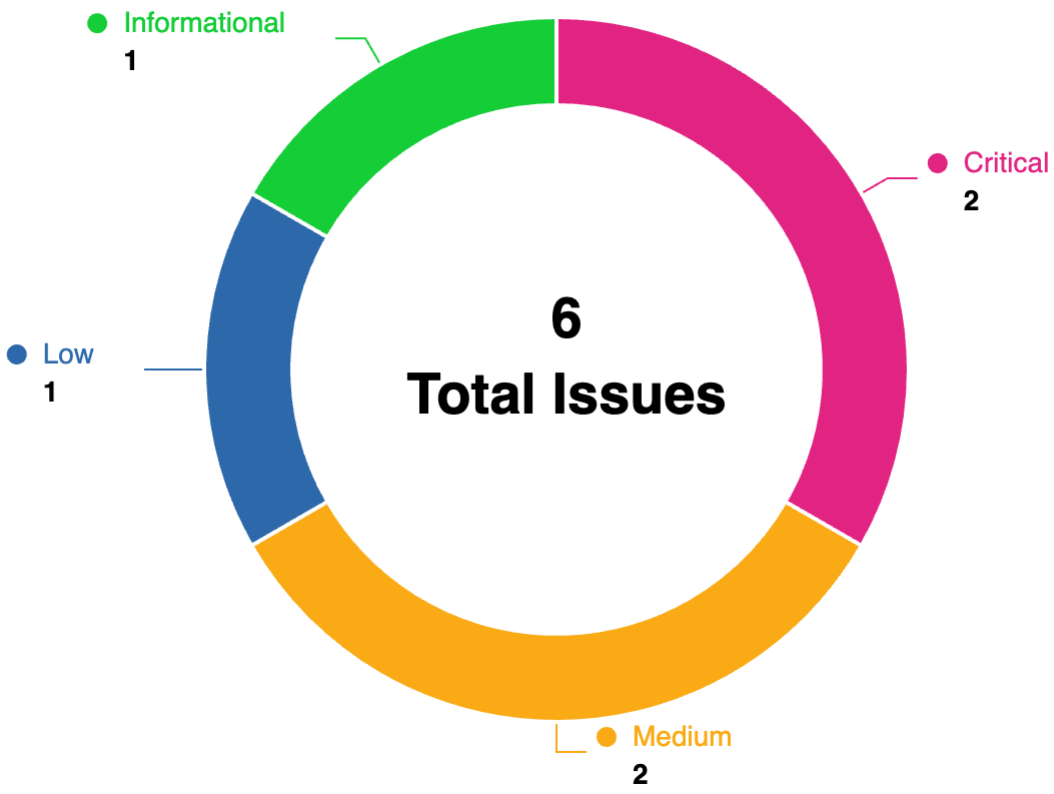
Overview

Project Name	Hajime
Language	Rust
Codebase	• https://github.com/flybot-aki/HajimeBot-Program • audit version - zip file • final version - 548864e6f55b7cd4e3abb2d708dc8e99c625af7f
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
./programs/hajime-ticket/src/lib.rs	013e9ffd2a5b38dca2594394277cf55c9597070a1636c29614f6c2c2205228c2

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
HJM-1	Incorrect use of `CpiContext` in `buy_nft`	Logical	Critical	Fixed	biakia
HJM-2	Inadequate Validation of `payment_token_account` in the `buy_nft` Function Leads to Potential NFT Acquisition Bypass	Privilege Related	Critical	Fixed	BradMoonUESTC
HJM-3	Potential front-run attack	Logical	Medium	Fixed	biakia
HJM-4	Mismatched instruction names	Logical	Medium	Fixed	biakia
HJM-5	Different payment tokens have the same price	Logical	Low	Fixed	biakia
HJM-6	Missing Emit Events	Code Style	Informational	Fixed	biakia

HJM-1:Incorrect use of `CpiContext` in `buy_nft`

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	biakia

Code Reference

- code/programs/hajime-ticket/src/lib.rs#L115-L149

```
115: pub fn buy_nft(ctx: Context<BuyNFT>, token_addr: Pubkey) -> Result<()> {
116:         msg!("start buy");
117:
118:         let token_state = &mut ctx.accounts.token_state;
119:
120:         // make PDA as signer
121:         let (_, account_nonce) = Pubkey::find_program_address(
122:             &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes()],
123:             ctx.program_id,
124:         );
125:         let seeds = &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes(), &[account_nonce]];
126:
127:         let destination = &ctx.accounts.payment_token_account;
128:         let source = &ctx.accounts.user_token_account;
129:         let token_program = &ctx.accounts.token_program;
130:         let authority = &ctx.accounts.user;
131:
132:         if source.mint != destination.mint || !token_state.allow_tokens.contains(&source.mint)
{
133:             return Err(ProgramError::InvalidArgument.into());
134:         }
135:
136:         // Transfer tokens from taker to initializer
137:         let cpi_accounts = Transfer {
138:             from: source.to_account_info().clone(),
139:             to: destination.to_account_info().clone(),
140:             authority: authority.to_account_info().clone(),
141:         };
142:         let cpi_program = token_program.to_account_info();
143:
144:         transfer(
145:             CpiContext::new_with_signer(cpi_program, cpi_accounts, &[seeds]),
146:             token_state.price,
147:         )?;
148:
149:         msg!("print new edition");
```

Description

biakia: In **`buy_nft`**, the signer will transfer some tokens to the **`payment_token_account`** to buy the NFT. It will use a **`Transfer`** CPI:

```
        let destination = &ctx.accounts.payment_token_account;
        let source = &ctx.accounts.user_token_account;
        let token_program = &ctx.accounts.token_program;
        let authority = &ctx.accounts.user;
        // Transfer tokens from taker to initializer
        let cpi_accounts = Transfer {
            from: source.to_account_info().clone(),
            to: destination.to_account_info().clone(),
            authority: authority.to_account_info().clone(),
        };
        let cpi_program = token_program.to_account_info();

        transfer(
            CpiContext::new_with_signer(cpi_program, cpi_accounts, &[seeds]),
            token_state.price,
        )?;
```

The **`authority`** should be a signed account to sign the CPI. The **`from`** is the source token account which is owned by **`authority`**. The **`to`** is the destination token account. In anchor framework, both user account and PDA can sign a CPI. The difference is that if a user account signs a CPI, you should use **`CpiContext::new(cpi_program, cpi_accounts)`** and if a PDA signs a CPI, you should use **`CpiContext::new_with_signer(cpi_program, cpi_accounts, seeds)`**. In **`buy_nft`**, the **`authority`** is a user account. The issue here is that the **`CpiContext`** is incorrect, causing the CPI fail to be called.
Reference: https://book.anchor-lang.com/anchor_in_depth/PDAs.html

Recommendation

biakia: Consider following fix:

```
transfer(
            CpiContext::new(cpi_program, cpi_accounts),
            token_state.price,
        )?;
```

Client Response

biakia: Fixed - fix by this https://github.com/flybot-aki/HajimeBot-Program/commit/631edb4c6f5aba384eb98e518d2309cfee985a39

HJM-2:Inadequate Validation of `payment_token_account` in the `buy_nft` Function Leads to Potential NFT Acquisition Bypass

Category	Severity	Client Response	Contributor
Privilege Related	Critical	Fixed	BradMoonUESTC

Code Reference

- code/programs/hajime-ticket/src/lib.rs#L120-L134

```
120: // make PDA as signer
121:         let (_, account_nonce) = Pubkey::find_program_address(
122:             &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes()],
123:             ctx.program_id,
124:         );
125:         let seeds = &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes(), &[account_nonce]];
126:
127:         let destination = &ctx.accounts.payment_token_account;
128:         let source = &ctx.accounts.user_token_account;
129:         let token_program = &ctx.accounts.token_program;
130:         let authority = &ctx.accounts.user;
131:
132:         if source.mint != destination.mint || !token_state.allow_tokens.contains(&source.mint)
{
133:             return Err(ProgramError::InvalidArgument.into());
134:         }
```

Description

BradMoonUESTC: The **`buy_nft`** function is intended to facilitate the purchase of NFTs (Non-Fungible Tokens) by transferring tokens from a buyer's token account to a seller's payment token account. However, a critical security vulnerability has been identified due to the lack of proper validation on the **`payment_token_account`**.
In the smart contract, the **`SetPayment`** struct correctly defines **`payment_token_account`** as a **`TokenAccount`** but lacks explicit constraints to ensure its validity and ownership:

```
pub struct SetPayment<'info> {
    ...
    pub payment_token_account: Account<'info, TokenAccount>,
}
```

While the **`set_payment`** function includes a check to ensure the **`payment_token_account`**'s owner matches the program's PDA account, identified by a specific prefix and the token address, this critical validation is absent in the **`buy_nft`** function:

```
let (account, _) = Pubkey::find_program_address(
    &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes()],
    ctx.program_id,
);
...
if payment.owner != account {
    return Err(ProgramError::IllegalOwner.into());
}
```

This oversight in the `buy_nft` function allows an attacker to use any `payment_token_account`, including ones they own, to bypass the intended security checks and potentially acquire NFTs without proper authorization or transfer of funds. The absence of a check against the `payment_token_account`'s owner in the `buy_nft` function's logic is a significant security flaw:

```
pub struct BuyNFT<'info> {
    ...
    #[account(mut)]
    pub payment_token_account: Account<'info, TokenAccount>,
    ...
}
```

Recommendation

BradMoonUESTC: To mitigate this vulnerability and ensure that only valid, program-owned `payment_token_account`s can be used in NFT transactions, it is recommended to add an ownership validation check in the `buy_nft` function. This can be achieved by verifying that the `payment_token_account`'s owner matches the expected program-derived account (PDA). The corrected implementation should include the following adjustment:

```
pub fn buy_nft(ctx: Context<BuyNFT>, token_addr: Pubkey) -> Result<()> {
    ...
    let (account, account_nonce) = Pubkey::find_program_address(
        &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes()],
        ctx.program_id,
    );
    ...
    let destination = &ctx.accounts.payment_token_account;
    ...
    if destination.owner != account {
        return Err(ProgramError::IllegalOwner.into());
    }
    ...
}
```

Client Response

BradMoonUESTC: Fixed - fix by this https://github.com/flybot-aki/HajimeBot-Program/commit/7c9519371d0ac4db3b99b3e246a921f2eb295f08

HJM-3:Potential front-run attack

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	biakia

Code Reference

- code/programs/hajime-ticket/src/lib.rs#L82-L87
- code/programs/hajime-ticket/src/lib.rs#L115-L147

```
82: pub fn set_price(ctx: Context<SetPrice>, _token_addr: Pubkey, new_price: u64) -> Result<()> {
83:         let token_state = &mut ctx.accounts.token_state;
84:         token_state.price = new_price;
85:
86:         Ok(())
87:     }
```

```
115: pub fn buy_nft(ctx: Context<BuyNFT>, token_addr: Pubkey) -> Result<()> {
116:         msg!("start buy");
117:
118:         let token_state = &mut ctx.accounts.token_state;
119:
120:         // make PDA as signer
121:         let (_, account_nonce) = Pubkey::find_program_address(
122:             &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes()],
123:             ctx.program_id,
124:         );
125:         let seeds = &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes(), &[account_nonce]];
126:
127:         let destination = &ctx.accounts.payment_token_account;
128:         let source = &ctx.accounts.user_token_account;
129:         let token_program = &ctx.accounts.token_program;
130:         let authority = &ctx.accounts.user;
131:
132:         if source.mint != destination.mint || !token_state.allow_tokens.contains(&source.mint)
{
133:             return Err(ProgramError::InvalidArgument.into());
134:         }
135:
136:         // Transfer tokens from taker to initializer
137:         let cpi_accounts = Transfer {
138:             from: source.to_account_info().clone(),
139:             to: destination.to_account_info().clone(),
140:             authority: authority.to_account_info().clone(),
141:         };
142:         let cpi_program = token_program.to_account_info();
143:
144:         transfer(
145:             CpiContext::new_with_signer(cpi_program, cpi_accounts, &[seeds]),
146:             token_state.price,
147:         )?;
```

Description

biakia: The **`buy_nft`** function allows the user to pay for NFT using the allowed token. The price is the **`token_state.price`**. This price can be changed by the function **`set_price`**. There is a potential front-run attack:

1. The price is 1 USDT now and Bob sends a transaction to buy the NTF
2. The admin front-run Bob's transaction and calls `set_price` to update the price to 10 USDT
3. Bob's transaction now is executed and he pays 10 USDT instead of 1 USDT to buy this NFT

Recommendation

biakia: Consider passing a price in the function `buy_nft` and check whether it is the same with `token_state.price`:

```
pub fn buy_nft(ctx: Context<BuyNFT>, token_addr: Pubkey, price: u64) -> Result<()> {
       msg!("start buy");

       let token_state = &mut ctx.accounts.token_state;
       if price!=token_state.price {
          return Err(ProgramError::InvalidArgument.into());
       }
```

Client Response

biakia: Fixed - fix by this https://github.com/flybot-aki/HajimeBot-Program/commit/3c96a0530a2e1703d752eca16f88522570fbc6a5

HJM-4:Mismatched instruction names

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	biakia

Code Reference

- code/programs/hajime-ticket/src/lib.rs#L82
- code/programs/hajime-ticket/src/lib.rs#L293-L309

```
82: pub fn set_price(ctx: Context<SetPrice>, _token_addr: Pubkey, new_price: u64) -> Result<()> {
```

```
293: #[derive(Accounts)]
294: #[instruction(token_addr: Pubkey)]
295: pub struct SetPrice<'info> {
296:     pub authority: Signer<'info>,
297:     #[account(
298:         seeds = [b"owner"],
299:         bump,
300:         has_one = authority
301:     )]
302:     pub owner_state: Account<'info, OwnerState>,
303:     #[account(
304:         mut,
305:         seeds = [NFT_STATE_PREFIX, &token_addr.key().as_ref()],
306:         bump,
307:     )]
308:     pub token_state: Account<'info, TokenState>,
309: }
```

Description

biakia: In function `set_price`, the name of the second argument is `_token_addr`:

```
pub fn set_price(ctx: Context<SetPrice>, _token_addr: Pubkey, new_price: u64) -> Result<()> {
        let token_state = &mut ctx.accounts.token_state;
        token_state.price = new_price;

        Ok(())
    }
```

However, in anchor struct `SetPrice`, the name of the `instruction` attribute is `token_addr`:

```
#[derive(Accounts)]
#[instruction(token_addr: Pubkey)]
pub struct SetPrice<'info> {
```

As per the document(https://github.com/coral-xyz/anchor/blob/7c424ee58a9525567ffadb18161396ba23a987db/docs/src/pages/docs/account-constraints.md?plain=1#L8), you can access the instruction's arguments with the `#[instruction(..)]` attribute. If the name of the argument is mismatched with the `#[instruction(..)]` attribute, the argument parsing in anchor framework may fail.

Recommendation

biakia: Consider following fix:

```
pub fn set_price(ctx: Context<SetPrice>, token_addr: Pubkey, new_price: u64) -> Result<()> {
        let token_state = &mut ctx.accounts.token_state;
        token_state.price = new_price;

        Ok(())
    }
```

Client Response

biakia: Fixed - fix by this https://github.com/flybot-aki/HajimeBot-Program/commit/00b3df5a3c4add89176d650945c2aa906079d30b

HJM-5:Different payment tokens have the same price

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	biakia

Code Reference

- code/programs/hajime-ticket/src/lib.rs#L115-L147

```
115: pub fn buy_nft(ctx: Context<BuyNFT>, token_addr: Pubkey) -> Result<()> {
116:         msg!("start buy");
117:
118:         let token_state = &mut ctx.accounts.token_state;
119:
120:         // make PDA as signer
121:         let (_, account_nonce) = Pubkey::find_program_address(
122:             &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes()],
123:             ctx.program_id,
124:         );
125:         let seeds = &[NFT_ACCOUNT_PREFIX, &token_addr.to_bytes(), &[account_nonce]];
126:
127:         let destination = &ctx.accounts.payment_token_account;
128:         let source = &ctx.accounts.user_token_account;
129:         let token_program = &ctx.accounts.token_program;
130:         let authority = &ctx.accounts.user;
131:
132:         if source.mint != destination.mint || !token_state.allow_tokens.contains(&source.mint)
{
133:             return Err(ProgramError::InvalidArgument.into());
134:         }
135:
136:         // Transfer tokens from taker to initializer
137:         let cpi_accounts = Transfer {
138:             from: source.to_account_info().clone(),
139:             to: destination.to_account_info().clone(),
140:             authority: authority.to_account_info().clone(),
141:         };
142:         let cpi_program = token_program.to_account_info();
143:
144:         transfer(
145:             CpiContext::new_with_signer(cpi_program, cpi_accounts, &[seeds]),
146:             token_state.price,
147:         )?;
```

Description

biakia: The `buy_nft` function allows the user to pay for NFT using the allowed token:

```
if source.mint != destination.mint || !token_state.allow_tokens.contains(&source.mint) {
            return Err(ProgramError::InvalidArgument.into());
        }

        ...
        transfer(
            CpiContext::new_with_signer(cpi_program, cpi_accounts, &[seeds]),
            token_state.price,
        )?;
```

The problem here is that all payment tokens use the same price. Imagine now that there are two payment tokens, one is `WSOL` and the other is `USDT`, and the price is 1. The user who chooses to use `USDT` as payment will spend less than the user who chooses to use `WSOL`.

Recommendation

biakia: Consider setting different prices for different tokens.

Client Response

biakia: Fixed - fix by this https://github.com/flybot-aki/HajimeBot-Program/commit/548864e6f55b7cd4e3abb2d708dc8e99c625af7f

HJM-6:Missing Emit Events

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	biakia

Code Reference

- code/programs/hajime-ticket/src/lib.rs#L25
- code/programs/hajime-ticket/src/lib.rs#L34
- code/programs/hajime-ticket/src/lib.rs#L82
- code/programs/hajime-ticket/src/lib.rs#L89

```
25: pub fn init_auth(ctx: Context<InitAuth>) -> Result<()> {
```

```
34: pub fn change_auth(ctx: Context<ChangeAuth>, new_auth: Pubkey) -> Result<()> {
```

```
82: pub fn set_price(ctx: Context<SetPrice>, _token_addr: Pubkey, new_price: u64) -> Result<()> {
```

```
89: pub fn set_payment(ctx: Context<SetPayment>, token_addr: Pubkey) -> Result<()> {
```

Description

biakia: The following functions affect the status of sensitive variables and should be able to emit events as notifications:

1. init_auth
2. change_auth
3. set_price
4. set_payment

Recommendation

biakia: Consider adding events for sensitive actions and emit them in the above mentioned functions.

Client Response

biakia: Fixed - fix by this https://github.com/flybot-aki/HajimeBot-Program/commit/3580281c5d95ca131863f503086c7deaea699032

Disclaimer

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 Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. 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 Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. 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.