ExVul

SMART CONTRACT AUDIT REPORT

Jupbot Smart Contract

AUGUST 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **Jupbot** to review smart contract implementation. The assessment was conducted in accordance with our systematic approach to evaluate potential security issues based upon customer requirement. The report provides detailed recommendations to resolve the issue and provide additional suggestions or recommendations for improvement.

The outcome of the assessment outlined in chapter 3 provides the system's owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the underlying technical issue.

1.1 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10] which is the gold standard in risk assessment using the following risk models:

- **Likelihood**: represents how likely a particular vulnerability is to be uncovered and exploited in the wild.
- **Impact**: measures the technical loss and business damage of a successful attack.
- **Severity**: determine the overall criticality of the risk.

Likelihood can be: High, Medium and Low and impact are categorized into: High, Medium, Low, Informational. Severity is determined by likelihood and impact and can be classified into five categories accordingly: Critical, High, Medium, Low, Informational shown in table 1.1.

Likelihood \ IMPACT	Informational	Low	Medium	High
High	INFO	MEDIUM	HIGH	CRITICAL
Medium	INFO	LOW	MEDIUM	HIGH
Low	INFO	LOW	LOW	MEDIUM

Table 1.1 Overall Risk Severity

To evaluate the risk, we will be going through a list of items, and each would be labelled with a severity category. The audit was performed with a systematic approach guided by a comprehensive assessment list carefully designed to identify known and impactful security issues. If our tool or analysis does not identify any issue, the contract can be considered safe regarding the assessed item. For any discovered issue, we might further deploy contracts on our private test environment and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.2.

- **Basic Coding Bugs**: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- **Code and business security testing**: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- **Additional Recommendations**: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Category	Assessment Item
Basic Coding Assessment	• Apply Verification Control • Authorization Access Control • Forged Transfer Vulnerability • Forged Transfer Notification • Numeric Overflow • Transaction Rollback Attack • Transaction Block Stuffing Attack • Soft Fail Attack • Hard Fail Attack • Abnormal Memo • Abnormal Resource Consumption • Secure Random Number

Advanced Source Code Scrutiny	• Asset Security • Cryptography Security • Business Logic Review • Source Code Functional Verification • Account Authorization Control • Sensitive Information Disclosure • Circuit Breaker • Blacklist Control • System API Call Analysis • Contract Deployment Consistency Check • Abnormal Resource Consumption
Additional Recommendations	• Semantic Consistency Checks • Following Other Best Practices

Table 1.2: The Full List of Assessment Items

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development.

2. FINDINGS OVERVIEW

2.1 Project Info And Contract Address

Project Name	Audit Time	Language
Jupbot	07/08/2025 - 12/08/2025	Rust

Repository
https://github.com/dextools-io/jupbot-solana-sc.git

Commit Hash
2b7141ac0f2138feb56c7b0cce6929c5d52c5691

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	0
MEDIUM	1
LOW	5
INFO	2

2.3 Key Findings

Severity	Findings Title	Status
MEDIUM	Executor Fees Can Be Claimed by an Unrelated Executor Upon Closing an Order	Fixed
LOW	Inconsistent Refund Amount Information in Order Closure Event	Fixed
LOW	Unbounded Executor Fee Configuration Risk	Fixed
LOW	Order ID Reuse After Order Closure	Fixed
LOW	close_order Will Fail if User Lacks a Destination Token Account	Fixed
LOW	Executor Controls Swap Direction Without User Consent	Fixed
INFO	Missing Change Validation in Fee Recipients Update	Acknowledge
INFO	Swap Events Missing User Identifier for Frontend Differentiation	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Executor Fees Can Be Claimed by an Unrelated Executor Upon Closing an Order

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`close_order.rs::close_order_handler()`

DESCRIPTION:

In close_order.rs, the permission model allows any registered Executor in the system to close any user's order. When an executor calls close_order, they can claim a refund_amount for themselves, which is directly deducted from the prepaid SOL stored in the order_state account's balance.

The vulnerable code section:

```
// close_order.rs
#[account(
    // ...
    constraint = /* ... */
       jupbot_state.executors.contains(&initiator.key())
)]
pub initiator: Signer<'info>,

if jupbot_state
    .executors
    .contains(&ctx.accounts.initiator.key())
    && refund_amount > 0
{
    // ...
    ctx.accounts.order_state.sub_lamports(refund_amount)?;
    ctx.accounts.initiator.add_lamports(refund_amount)?;
}
```

The contract does not verify whether the executor closing the order has ever serviced that specific order. An executor who has never performed any swaps for an order can preemptively call close_order once the order is completed (swap_count is zero) to claim the final fee reserved for closing the order.

IMPACT:

Unfair Fee Distribution: The user's prepaid fees are not distributed to the executor who actually performed the work.

RECOMMENDATIONS:

1. Track Contributor in OrderState: Add a last_executor: Option field to the OrderState struct.
2. Update Swap Logic: After each successful swap (specifically within the after_swap hook), update the order_state.last_executor to the public key of the current executor.
3. Tighten close_order Permissions: Modify the refund logic in close_order.rs to include an additional check, ensuring that only the order's user or the last_executor is eligible to claim a refund when closing the order.

```
let is_last_executor = if let Some(last_exec_key) =
    ctx.accounts.order_state.last_executor {
     last_exec_key == ctx.accounts.initiator.key()
} else {
     false
};

if is_last_executor && refund_amount > 0 {
     // ... allow refund
}
```

3.2 Inconsistent Refund Amount Information in Order Closure Event

SEVERITY: LOW **STATUS:** Fixed

PATH:

`close_order.rs::close_order_handler()`

DESCRIPTION:

The close_order_handler function emits incorrect refund amount information in the OrderClosed event when the initiator is not an authorized executor. The event always uses the input refund_amount parameter regardless of whether the actual refund occurred, leading to misleading event data.

```
if jupbot_state
    .executors
    .contains(&ctx.accounts.initiator.key())
    && refund_amount > 0
{
    require_gte!(
        jupbot_state.executor_fee_lamports,
        refund_amount,
        JupbotErrorCode::RefundExecutorAmountOutOfBounds
    );
    ctx.accounts.order_state.sub_lamports(refund_amount)?;
    ctx.accounts.initiator.add_lamports(refund_amount)?;
}

emit!(crate::events::OrderClosed {
    initiator: ctx.accounts.initiator.key(),
    order_id: ctx.accounts.order_state.order_id,
    token_a_amount,
    token_b_amount,
    order_state: ctx.accounts.order_state.clone().into_inner(),
    refund_amount,  // Always shows input amount, not actual refund
});
```

When a non-executor closes the order, no refund occurs but the event still reports the input refund_amount, creating inconsistency between actual state and event information.

IMPACT:

Misleading event logs can cause frontend applications to display incorrect refund information and complicate audit trails.

RECOMMENDATIONS:

Track the actual refund amount and use it in the event emission to ensure consistency:

```
let actual_refund_amount = if jupbot_state
    .executors
    .contains(&ctx.accounts.initiator.key())
    && refund_amount > 0
{
    require_gte!(
        jupbot_state.executor_fee_lamports,
        refund_amount,
        JupbotErrorCode::RefundExecutorAmountOutOfBounds
    );
    ctx.accounts.order_state.sub_lamports(refund_amount)?;
    ctx.accounts.initiator.add_lamports(refund_amount)?;
    refund_amount
} else {
    0
};

emit!(crate::events::OrderClosed {
    initiator: ctx.accounts.initiator.key(),
    order_id: ctx.accounts.order_state.order_id,
    token_a_amount,
    token_b_amount,
    order_state: ctx.accounts.order_state.clone().into_inner(),
    refund_amount: actual_refund_amount,
});
```

3.3 Unbounded Executor Fee Configuration Risk

SEVERITY: LOW **STATUS:** Fixed

PATH:

`update_fees.rs::update_fees_handler()`

DESCRIPTION:

The update_fees_handler function allows the admin to set executor_fee_lamports to any arbitrary value without upper bounds, creating potential economic risks for users.

```
if let Some(executor_fee_lamports) = executor_fee_lamports {
    require_neq!(
        executor_fee_lamports,
        old_executor_fee_lamports,
        JupbotErrorCode::NoChange
    );
    jupbot_state.executor_fee_lamports = executor_fee_lamports;  // No
       upper bound check
}
```

The executor fee is charged per swap operation and order closure, calculated as:

```
let executor_fee = ctx
    .accounts
    .jupbot_state
    .executor_fee_lamports
    .checked_mul((swap_count + 1) as u64) // +1 for order closing
    .ok_or(JupbotErrorCode::Overflow)?;
```

IMPACT:

Admin can set unlimited executor fees, potentially making the service economically unviable and creating governance risks for users.

RECOMMENDATIONS:

Implement reasonable upper bounds for the executor fee to protect users from excessive charges:

```
const MAX_EXECUTOR_FEE_LAMPORTS: u64 = 10_000; // 0.1 SOL max per
    operation

pub fn update_fees_handler(
    ctx: &mut Context<UpdateFees>,
    order_fee_bps: Option<u32>,
    executor_fee_lamports: Option<u64>,
) -> Result<()> {
    let jupbot_state = &mut ctx.accounts.jupbot_state;

    require!(
        order_fee_bps.is_some() || executor_fee_lamports.is_some(),
        JupbotErrorCode::NoChange
    );

    let old_order_fee = jupbot_state.order_fee;
    if let Some(order_fee_bps) = order_fee_bps {
        let order_fee = Fee::from_basis_points(order_fee_bps)?;
        require_neq!(order_fee, old_order_fee, JupbotErrorCode::NoChange);
        jupbot_state.order_fee = order_fee;
    }

    let old_executor_fee_lamports = jupbot_state.executor_fee_lamports;
    if let Some(executor_fee_lamports) = executor_fee_lamports {
        require_lte!(
            executor_fee_lamports,
            MAX_EXECUTOR_FEE_LAMPORTS,
            JupbotErrorCode::MaxFeeExceeded
        );
        require_neq!(
            executor_fee_lamports,
            old_executor_fee_lamports,
            JupbotErrorCode::NoChange
        );
        jupbot_state.executor_fee_lamports = executor_fee_lamports;
    }

    emit!(crate::events::FeesUpdated {
        old_order_fee,
        old_executor_fee_lamports,
        order_fee: jupbot_state.order_fee,
```

```
        executor_fee_lamports: jupbot_state.executor_fee_lamports,
    });

    Ok(())
}
```

3.4 Order ID Reuse After Order Closure

SEVERITY: LOW **STATUS:** Fixed

PATH:

`create_order.rs::create_order_handler()`

DESCRIPTION:

The create_order_handler function allows users to reuse the same order_id after closing an order, leading to duplicate order IDs in events and potential confusion in indexing systems.

```
#[account(
    init,
    payer = user,
    space = 8 + OrderState::INIT_SPACE,
    seeds = [
        crate::constants::ORDER_STATE_SEED,
        user.key().as_ref(),
        order_id.to_le_bytes().as_ref()  // Same seeds can be reused
    ],
    bump,
)]
pub order_state: Box<Account<'info, OrderState>>,
```

When an order is closed, the order_state account is closed and its lamports are returned to the user. The same order_id can then be used to create a new order with identical PDA seeds, resulting in duplicate order IDs in events.

IMPACT:

Duplicate order IDs in events can cause confusion in frontend applications and indexing systems, making it difficult to distinguish between different order instances with the same ID.

RECOMMENDATIONS:

Add a unique identifier to distinguish between order instances, such as a creation timestamp or user-specific order sequence number:

```
#[event]
pub struct OrderCreated {
    pub order_id: u128,
    pub creation_time: i64,  // Add unique timestamp
    pub token_a_amount: u64,
    pub order_state: crate::states::OrderState,
    pub executor_fee: u64,
    pub swap_count: u8,
}
```

3.5 close_order Will Fail if User Lacks a Destination Token Account

SEVERITY: LOW **STATUS:** Fixed

PATH:

`close_order.rs`

DESCRIPTION:

The code doesn't account for users who don't have a pre-existing account for Token B. If a swap executes and the order tries to send Token B back to the user, the transaction will fail because the destination account is missing.

```
#[account(
    mut,
    associated_token::mint = token_b,
    associated_token::authority = user,
    associated_token::token_program = token_b_program,
)]
pub user_token_b_ata: Box<InterfaceAccount<'info, TokenAccount>>,

// ...

crate::transfer!(
    ctx.accounts.pda_token_b_ta,   // from
    ctx.accounts.user_token_b_ata, // to (destination)
    // ...
);
```

IMPACT:

It creates a Denial of Service (DoS) condition, preventing the user from withdrawing their assets through the protocol's standard procedure.

RECOMMENDATIONS:

Option 1: In the close_order instruction, add the init_if_needed constraint for user_token_b_ata:

```
// file: close_order.rs
#[account(
    init_if_needed,
    payer = initiator,
    mut,
    associated_token::mint = token_b,
    associated_token::authority = user,
    // ...
)]
pub user_token_b_ata: Box<InterfaceAccount<'info, TokenAccount>>,
```

Option 2: In the create_order instruction, proactively create the user_token_b_ata for the user in advance:

```
// file: create_order.rs
#[derive(Accounts)]
pub struct CreateOrder<'info> {
    // ... other accounts

    #[account(
        init_if_needed,
        payer = user,
        associated_token::mint = token_b,
        associated_token::authority = user,
    )]
    pub user_token_b_ata: Account<'info, TokenAccount>,

    // ... other accounts
}
```

3.6 Executor Controls Swap Direction Without User Consent

SEVERITY: LOW **STATUS:** Fixed

PATH:

`swap.rs::swap_handler()`

DESCRIPTION:

Executors control swap direction via reverse_swap parameter, allowing them to execute trades opposite to user intent. Users create orders expecting A→B but executors can force B→A.

```
// swap.rs
pub fn swap_handler(
    ctx: &mut Context<Swap>,
    swap_args: okx_dex::SwapArgs,
    okx_order_id: u64,
    reverse_swap: bool,  // A to B or B to A
    refund_executor_amount: u64,
) -> Result<()> {

// utils/hooks.rs
let (source, destination) = if reverse_swap {
    (pda_token_b_ta, pda_token_a_ta)  // B to A
} else {
    (pda_token_a_ta, pda_token_b_ta)  // A to B
};
```

Scenario: 1. User creates order: USDC to SOL (buy SOL) 2. Executor sets reverse_swap = true 3. Actual execution: SOL to USDC (sell SOL) 4. Result: User loses SOL, opposite to intent

IMPACT:

Executors can execute trades opposite to user intentions, leading to potential financial losses and loss of user control over their own trading strategies.

RECOMMENDATIONS:

Implement deterministic swap direction based on order state rather than executor input.

Option 1: Store Swap Direction in OrderState

```
// states/order_state.rs
pub struct OrderState {
    pub version: OrderVersion,
    pub user: Pubkey,
    pub order_id: u128,
    pub creation_time: i64,
    pub bump: u8,
    pub swap_count: u8,
    pub token_a: Pubkey,
    pub token_b: Pubkey,
    pub token_a_pda_ta_bump: u8,
    pub token_b_pda_ta_bump: u8,
    pub swap_directions: Vec<bool>,  // true = reverse, false = forward
    pub current_swap_index: u8,
}

// create_order.rs
let swap_directions = vec![false; swap_count as usize];
let order_state = OrderState {
    swap_directions,
    current_swap_index: 0,
};

// hooks.rs
let reverse_swap =
   order_state.swap_directions[order_state.current_swap_index as usize];
```

Option 2: Remove Reverse Swap Entirely

```
// swap.rs
pub fn swap_handler<'info>(
    ctx: &mut Context<'_, '_, '_, 'info, Swap<'info>>,
    swap_args: okx_dex::SwapArgs,
    okx_order_id: u64,
    refund_executor_amount: u64,
) -> Result<()> {
    // Always execute A to B
    let reverse_swap = false;
```

3.7 Missing Change Validation in Fee Recipients Update

SEVERITY: INFO **STATUS:** Acknowledge

PATH:

`update_fee_recipients.rs::update_fee_recipients_handler()`

DESCRIPTION:

The update_fee_recipients_handler function lacks validation to ensure the new fee recipients are actually different from the existing ones. Unlike other update operations in the codebase (e.g., update_fees.rs), this function does not verify whether the provided new fee recipients represent an actual change.

```
pub fn update_fee_recipients_handler(
    ctx: &mut Context<UpdateFeeRecipients>,
    new_fee_recipients: Vec<crate::utils::FeeRecipient>,
) -> Result<()> {
    let jupbot_state = &mut ctx.accounts.jupbot_state;

    new_fee_recipients.validate()?;

    emit!(crate::events::FeeRecipientsUpdated {
        old_fee_recipients: jupbot_state.fee_recipients.clone(),
        new_fee_recipients: new_fee_recipients.clone(),
    });

    jupbot_state.fee_recipients = new_fee_recipients;

    Ok(())
}
```

IMPACT:

While this doesn't pose a security risk, it allows unnecessary transactions and events that don't change state, potentially wasting gas and creating noise in event logs.

RECOMMENDATIONS:

Add change validation before updating fee recipients to maintain consistency with other update operations:

```
pub fn update_fee_recipients_handler(
    ctx: &mut Context<UpdateFeeRecipients>,
    new_fee_recipients: Vec<crate::utils::FeeRecipient>,
) -> Result<()> {
    let jupbot_state = &mut ctx.accounts.jupbot_state;

    new_fee_recipients.validate()?;

    // Add change validation
    require_neq!(
        new_fee_recipients,
        jupbot_state.fee_recipients,
        JupbotErrorCode::NoChange
    );

    emit!(crate::events::FeeRecipientsUpdated {
        old_fee_recipients: jupbot_state.fee_recipients.clone(),
        new_fee_recipients: new_fee_recipients.clone(),
    });

    jupbot_state.fee_recipients = new_fee_recipients;

    Ok(())
}
```

3.8 Swap Events Missing User Identifier for Frontend Differentiation

SEVERITY: INFO **STATUS:** Fixed

PATH:

`events.rs::SwapExecuted`

DESCRIPTION:

All swap execution events (SwapExecuted, ProxySwapExecuted, SwapV3Executed, SwapTobV3Executed) only include order_id without user identification. Since order_id is user-provided and not globally unique, different users can use the same order_id, making it impossible for frontend applications to accurately distinguish between different users' orders.

```
#[event]
pub struct SwapExecuted {
    pub order_id: u128,
    pub executor: Pubkey,
    pub swap_count: u8,
    pub fee_amount: u64,
}
```

The order_id is generated from user input during order creation:

```
// create_order.rs
seeds = [
    crate::constants::ORDER_STATE_SEED,
    user.key().as_ref(),
    order_id.to_le_bytes().as_ref()
],
```

IMPACT:

Frontend applications cannot accurately track user-specific swap events when multiple users use the same order_id, leading to potential confusion and incorrect user interface displays.

RECOMMENDATIONS:

Add user: Pubkey field to all swap events to enable proper user identification:

```
#[event]
pub struct SwapExecuted {
    pub order_id: u128,
    pub user: Pubkey,        // Add user identifier
    pub executor: Pubkey,
    pub swap_count: u8,
    pub fee_amount: u64,
}

emit_cpi!(crate::events::SwapExecuted {
    order_id: swap_accounts.order_state.order_id,
    user: swap_accounts.order_state.user,    // Include user
    executor: swap_accounts.executor.key(),
    swap_count: swap_accounts.order_state.swap_count,
    fee_amount,
});
```

4. CONCLUSION

In this audit, we thoroughly analyzed **Jupbot** smart contract implementation. The problems found are described and explained in detail in Section 3. The problems found in the audit have been communicated to the project leader. We therefore consider the audit result to be **PASSED**.

To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5. APPENDIX

5.1 Basic Coding Assessment

5.1.1 Apply Verification Control

Description	The security of apply verification
Result	Not found
Severity	CRITICAL

5.1.2 Authorization Access Control

Description	Permission checks for external integral functions
Result	Not found
Severity	CRITICAL

5.1.3 Forged Transfer Vulnerability

Description	Assess whether there is a forged transfer notification vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.4 Transaction Rollback Attack

Description	Assess whether there is transaction rollback attack vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.5 Transaction Block Stuffing Attack

Description	Assess whether there is transaction blocking attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.6 Soft Fail Attack Assessment

Description	Assess whether there is soft fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.7 Hard Fail Attack Assessment

Description	Examine for hard fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.8 Abnormal Memo Assessment

Description	Assess whether there is abnormal memo vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.9 Abnormal Resource Consumption

Description	Examine whether abnormal resource consumption in contract processing
Result	Not found
Severity	CRITICAL

5.1.10 Random Number Security

Description	Examine whether the code uses insecure random number
Result	Not found
Severity	CRITICAL

5.2 Advanced Code Scrutiny

5.2.1 Cryptography Security

Description	Examine for weakness in cryptograph implementation
Result	Not found
Severity	HIGH

5.2.2 Account Permission Control

Description	Examine permission control issue in the contract
Result	Not found
Severity	MEDIUM

5.2.3 Malicious Code Behavior

Description	Examine whether sensitive behavior present in the code
Result	Not found
Severity	MEDIUM

5.2.4 Sensitive Information Disclosure

Description	Examine whether sensitive information disclosure issue present in the code
Result	Not found
Severity	MEDIUM

5.2.5 System API

Description	Examine whether system API application issue present in the code
Result	Not found
Severity	LOW

6. 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 Services Agreement, or the scope of services, and terms and conditions provided to 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 without ExVul's prior written consent.

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 ExVul 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. ExVul's position is that each company and individual are responsible for their own due diligence and continuous security. ExVul'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.

7. REFERENCES

[1] MITRE. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/definitions/191.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[3] MITRE. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/definitions/400.html.

[4] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

[5] MITRE. CWE-684: Protection Mechanism Failure. https://cwe.mitre.org/data/definitions/693.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE CATEGORY: Resource Management Errors. https://cwe.mitre.org/data/definitions/399.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec