ExVul

SMART CONTRACT AUDIT REPORT

DefAI Staking Smart Contract

AUGUST 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **DefAI Staking** 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
DefAI	21/07/2025 - 11/08/2025	Rust

Repository
https://github.com/defaiza/audit.git

Commit Hash
bbf88147743821d60bdbcf335e25e25e8ac9159f441

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	3
MEDIUM	2
LOW	1
INFO	0

2.3 Key Findings

Severity	Findings Title	Status
HIGH	Unvalidated Accounts in fund_escrow Lead to State Pollution and Protocol Insolvency	Fixed
HIGH	Unvalidated Accounts in unstake_tokens Allow Penalty Interception	Fixed
HIGH	Unvalidated Escrow Account in compound_rewards Leads to State Pollution	Fixed
MEDIUM	Combination of State Update Leads to Permanent Loss of Funds	Fixed
MEDIUM	Penalty logic in unstake_tokens can be bypassed	Fixed
LOW	Initialization Functions Lack Authority Validation	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Unvalidated Accounts in fund_escrow Lead to State Pollution and Protocol Insolvency

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`defai_staking/src/lib.rs::fund_escrow`

DESCRIPTION:

The fund_escrow function, which allows anyone to add funds to the reward pool, fails to apply strict constraints to the escrow_token_account and defai_mint accounts passed into its context, FundEscrow.

An attacker can exploit this by passing in a malicious token (evil_mint) they created, along with its corresponding token account (ATA). Although the genuine reward_escrow PDA cannot be forged, the program will erroneously add the amount of the malicious tokens to the total_balance state of the genuine reward_escrow account. This leads to a pollution of the protocol's state.

The vulnerable code section:

```
#[derive(Accounts)]
pub struct FundEscrow<'info> {
    #[account(mut)]
    pub reward_escrow: Account<'info, RewardEscrow>,

    #[account(mut)]
    pub escrow_token_account: InterfaceAccount<'info, TokenAccount>, //
        Attacker controlled

    #[account(mut)]
    pub funder_token_account: InterfaceAccount<'info, TokenAccount>, //
        Attacker controlled

    #[account(mut)]
    pub funder: Signer<'info>,
```

```
    pub defai_mint: InterfaceAccount<'info, Mint>, // Attacker controlled
    pub token_program: Interface<'info, TokenInterface>,
}

pub fn fund_escrow(ctx: Context<FundEscrow>, amount: u64) -> Result<()> {
    // The transfer succeeds because all mints match the attacker's
       evil_mint
    transfer_checked(transfer_ctx, amount,
       ctx.accounts.defai_mint.decimals)?;

    // Critical vulnerability: The amount of malicious tokens is added to
       the state of the genuine escrow account
    let escrow = &mut ctx.accounts.reward_escrow;
    escrow.total_balance =
       escrow.total_balance.checked_add(amount).unwrap();

    Ok(())
}
```

IMPACT:

This is a critical vulnerability that allows an attacker to destroy the protocol's economic model through state pollution, ultimately enabling the theft of all users' stake funds. The maliciously inflated reward_escrow.total_balance invalidates all functions that rely on this value for security checks.

RECOMMENDATIONS:

It is imperative to add strict account constraints to the FundEscrow context to ensure that all incoming accounts match the authoritative addresses stored in the program's state:

```
#[derive(Accounts)]
pub struct FundEscrow<'info> {
    // 1. Bring in ProgramState to access authoritative addresses
    pub program_state: Account<'info, ProgramState>,

    #[account(
        mut,
        // 2. Ensure reward_escrow is the correct PDA
        seeds = [b"reward-escrow", program_state.key().as_ref()],
        bump
    )]
```

```
    pub reward_escrow: Account<'info, RewardEscrow>,

    #[account(
        mut,
        // 3. Ensure escrow_token_account is the correct ATA and is owned
            by reward_escrow
        seeds = [b"escrow-vault", program_state.key().as_ref()],
        bump,
        token::authority = reward_escrow,
        token::mint = defai_mint, // 4. Ensure the ATA's mint matches the
            provided mint
    )]
    pub escrow_token_account: InterfaceAccount<'info, TokenAccount>,

    #[account(mut)]
    pub funder_token_account: InterfaceAccount<'info, TokenAccount>,

    #[account(mut)]
    pub funder: Signer<'info>,

    #[account(
        // 5. Ensure the provided mint is the official one from
            ProgramState
        constraint = defai_mint.key() == program_state.defai_mint @
            StakingError::InvalidMint
    )]
    pub defai_mint: InterfaceAccount<'info, Mint>,

    pub token_program: Interface<'info, TokenInterface>,
}
```

3.2 Unvalidated Accounts in unstake_tokens Allow Penalty Interception

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`defai_staking/src/lib.rs::unstake_tokens`

DESCRIPTION:

The unstake_tokens function, when processing penalties generated from early withdrawals, fails to validate whether the receiving account for the penalty (escrow_token_account) is the official reward pool address. The function trusts the account address provided by the user, which allows any user performing an early unstake to supply an account they control, thereby intercepting the penalty fee that should have gone to the protocol.

The vulnerable code section:

```
#[derive(Accounts)]
pub struct UnstakeTokens<'info> {
    // ...
    #[account(mut)]
    pub reward_escrow: Account<'info, RewardEscrow>, //  Not validated as
       the official PDA

    #[account(mut)]
    pub escrow_token_account: InterfaceAccount<'info, TokenAccount>, //
       Attacker-controlled
    // ...
}

if penalty > 0 {
    let transfer_penalty_ctx = CpiContext::new_with_signer(
        // ...
        TransferChecked {
            from: ctx.accounts.stake_vault.to_account_info(),      //
               Source: The real public vault
            to: ctx.accounts.escrow_token_account.to_account_info(), //
               Destination: Attacker's personal account
            authority: ctx.accounts.stake_vault.to_account_info(),  //
               Authority: The real public vault
```

```
            mint: ctx.accounts.defai_mint.to_account_info(),
        },
        signer, // Signature: The real vault's PDA signature
    );
    transfer_checked(transfer_penalty_ctx, penalty,
       ctx.accounts.defai_mint.decimals)?;
    // ...
}
```

IMPACT:

This vulnerability allows users to directly steal penalty fees that are intended as protocol revenue. Any user executing an early unstake can exploit this method to intercept their own penalty payment. This breaks the economic model designed to discourage short-term speculation and replenish the reward pool.

RECOMMENDATIONS:

It is essential to add strict account constraints to the UnstakeTokens context to ensure that the provided reward_escrow and escrow_token_account are the official, unique PDA accounts of the protocol:

```
#[derive(Accounts)]
pub struct UnstakeTokens<'info> {
    pub program_state: Account<'info, ProgramState>,

    #[account(
        mut,
        seeds = [b"reward-escrow", program_state.key().as_ref()],
        bump = program_state.escrow_bump
    )]
    pub reward_escrow: Account<'info, RewardEscrow>,

    #[account(
        mut,
        seeds = [b"escrow-vault", program_state.key().as_ref()],
        bump,
        token::authority = reward_escrow,
        token::mint = defai_mint
    )]
    pub escrow_token_account: InterfaceAccount<'info, TokenAccount>,
```

```
    // ... other accounts
}
```

3.3 Unvalidated Escrow Account in compound_rewards Leads to State Pollution

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`defai_staking/src/lib.rs::compound_rewards`

DESCRIPTION:

The compound_rewards function has a flaw where it fails to validate the address of the incoming reward_escrow account. It only checks if the account is mutable but does not use a PDA constraint to verify that it is the official, program-controlled escrow account.

This allows a caller to pass in a self-created, fake reward_escrow account containing false data to execute the compound instruction.

The vulnerable code section:

```
#[derive(Accounts)]
pub struct CompoundRewards<'info> {
    // ...
    #[account(mut)] // Address of reward_escrow is not validated here
    pub reward_escrow: Account<'info, RewardEscrow>,
    // ...
}

// The check passes because it uses the balance from the fake account
require!(
    ctx.accounts.reward_escrow.total_balance >= total_unclaimed,
    StakingError::InsufficientEscrowBalance
);

// The attacker's stake and the global stake are incorrectly increased
user_stake.staked_amount = user_stake.staked_amount
    .checked_add(total_unclaimed).unwrap();

program_state.total_staked = program_state.total_staked
    .checked_add(total_unclaimed).unwrap();
```

```
// The balance deduction is applied to the fake account; the real escrow
    is untouched
let escrow = &mut ctx.accounts.reward_escrow;
escrow.total_balance =
    escrow.total_balance.checked_sub(total_unclaimed).unwrap();
```

IMPACT:

This vulnerability pollutes the protocol's core state. An attacker can create a fake RewardEscrow account with an arbitrarily large balance and use it to artificially inflate their staked amount and the global total_staked value, while the real reward pool remains untouched.

RECOMMENDATIONS:

It is recommended to add a strict PDA constraint to the reward_escrow account within the CompoundRewards context:

```
#[derive(Accounts)]
pub struct CompoundRewards<'info> {
    #[account(mut)]
    pub program_state: Account<'info, ProgramState>,

    #[account(
        mut,
        // Add this constraint to ensure it's the official PDA
        seeds = [b"reward-escrow", program_state.key().as_ref()],
        bump = program_state.escrow_bump
    )]
    pub reward_escrow: Account<'info, RewardEscrow>,

    // ...
}
```

3.4 Combination of State Update Leads to Permanent Loss of Funds

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`defai_staking/src/lib.rs::stake_tokens and update_defai_mint`

DESCRIPTION:

The protocol has a critical design flaw: the core stake_tokens function lacks address validation for the Vault account, while the update_defai_mint function allows an administrator to change the official token recorded in the program's state. This combination creates a scenario where funds can be permanently locked.

The vulnerable code sections:

```
#[derive(Accounts)]
pub struct StakeTokens<'info> {
    // ...
    #[account(mut)] // Missing PDA validation allows any vault address
    pub stake_vault: InterfaceAccount<'info, TokenAccount>,
    // ...
}

// ... in stake_tokens function
let transfer_ctx = CpiContext::new(
    ctx.accounts.token_program.to_account_info(),
    TransferChecked {
        to: ctx.accounts.stake_vault.to_account_info(), // Can point to
            the new_vault
        authority: ctx.accounts.user.to_account_info(), // Authority is
            the user
        // ...
    },
);
```

And in the withdrawal function:

```
// ... in unstake_tokens function
```

```
let program_state_key = ctx.accounts.program_state.key(); // This key is
    constant
let seeds = &[
    b"stake-vault",
    program_state_key.as_ref(), // This part of the seed is constant
    &[ctx.accounts.program_state.vault_bump],
];
let signer = &[&seeds[..]]; // The signer always represents the original
    vault

let transfer_ctx = CpiContext::new_with_signer(
    // ...
    signer, // This signature cannot authorize operations on the new_vault
);
```

IMPACT:

All funds deposited into the new vault will be permanently locked and unrecoverable. During a routine operation intended to upgrade the protocol to support a new token, this vulnerability leads to a catastrophic outcome where users can stake into new vaults but cannot withdraw their funds.

RECOMMENDATIONS:

1. **Enforce Account Validation**: In all functions that interact with funds, a PDA constraint must be added to strictly validate the vault address.
2. **Remove Risky Function**: Completely remove the update_defai_mint function. A token migration should be handled through a comprehensive solution that includes a fund migration strategy, not a simple state update.

```
#[derive(Accounts)]
pub struct StakeTokens<'info> {
    pub program_state: Account<'info, ProgramState>,

    #[account(
        mut,
        seeds = [b"stake-vault", program_state.key().as_ref()],
        bump = program_state.vault_bump,
        token::mint = program_state.defai_mint,
        token::authority = program_state
    )]
```

```
    pub stake_vault: InterfaceAccount<'info, TokenAccount>,
    // ...
}
```

3.5 Penalty logic in unstake_tokens can be bypassed

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`defai_staking/src/lib.rs::calculate_unstake_penalty and stake_tokens`

DESCRIPTION:

The calculate_unstake_penalty function determines the penalty based on the stake_timestamp (the time of the initial stake). If a user adds more funds via stake_tokens after their initial stake, the new funds are considered as “old” as the first deposit because the stake_timestamp is only set once and not updated on subsequent stakes.

The vulnerable code sections:

```
fn calculate_unstake_penalty(
    stake_timestamp: i64,
    current_timestamp: i64,
    amount: u64,
) -> Result<u64> {
    let days_staked = (current_timestamp - stake_timestamp) / 86400;

    let penalty_bps = if days_staked < 30 {
        200  // 2%
    } else if days_staked < 90 {
        100  // 1%
    } else {
        0    // No penalty
    };

    Ok((amount as u128 * penalty_bps as u128 / BASIS_POINTS as u128) as
       u64)
}

pub fn stake_tokens(/* ... */) -> Result<()> {
    // ...
    if user_stake.owner == Pubkey::default() {
        // New user stake
        user_stake.stake_timestamp = clock.unix_timestamp;
```

```
    } else {
        // Existing user adds to stake - timestamp not updated!
        user_stake.staked_amount =
            user_stake.staked_amount.checked_add(amount).unwrap();
    }
    // ...
    Ok(())
}
```

IMPACT:

This vulnerability allows an attacker to bypass the early unstake penalty by first staking a small amount to “age” their account. After waiting 90 days to gain penalty-free status, they can then deposit a much larger sum, earn short-term rewards on this large capital, and immediately withdraw everything with zero penalty, completely nullifying the economic model designed to encourage long-term holding.

RECOMMENDATIONS:

Add a last_stake_timestamp field to the UserStake struct. This timestamp will be updated every time a user adds to their stake, and the penalty calculation should use this most recent timestamp:

```
#[account]
pub struct UserStake {
    pub owner: Pubkey,
    pub staked_amount: u64,
    pub stake_timestamp: i64,
    pub last_stake_timestamp: i64, // New field
    // ... other fields
}

pub fn stake_tokens(/* ... */) -> Result<()> {
    let clock = Clock::get()?;

    if user_stake.owner == Pubkey::default() {
        // New user stake
        user_stake.stake_timestamp = clock.unix_timestamp;
        user_stake.last_stake_timestamp = clock.unix_timestamp;
    } else {
        // Existing user adds to stake - update last_stake_timestamp
        user_stake.staked_amount =
```

```
            user_stake.staked_amount.checked_add(amount).unwrap();
        user_stake.last_stake_timestamp = clock.unix_timestamp;
    }
    // ...
    Ok(())
}

fn calculate_unstake_penalty(
    last_stake_timestamp: i64, // Use last_stake_timestamp instead
    current_timestamp: i64,
    amount: u64,
) -> Result<u64> {
    let days_staked = (current_timestamp - last_stake_timestamp) / 86400;
    // ... rest of the logic remains the same
}
```

3.6 Initialization Functions Lack Authority Validation

SEVERITY: LOW **STATUS:** Fixed

PATH:

`defai_staking/src/lib.rs::initialize_program and initialize_escrow`

DESCRIPTION:

The critical initialization functions initialize_program and initialize_escrow lack validation for the caller's identity. They accept any signer as the authority without verifying if the signer is the intended protocol administrator.

The vulnerable code sections:

```
#[derive(Accounts)]
pub struct InitializeProgram<'info> {
    #[account(
        init,
        payer = authority, // payer can be anyone
        // ...
    )]
    pub program_state: Account<'info, ProgramState>,

    #[account(mut)]
    pub authority: Signer<'info>, // Lacks validation for the authority's
       identity
}

#[derive(Accounts)]
pub struct InitializeEscrow<'info> {
    #[account(
        mut,
        seeds = [b"program-state"],
        bump
    )]
    pub program_state: Account<'info, ProgramState>,

    #[account(mut)]
```

```
    pub authority: Signer<'info>, // Lacks validation for the authority's
        identity
}
```

IMPACT:

The direct impact is low as these functions are intended to be called only once. However, it introduces potential risks like initialization front-running where a malicious user could front-run the actual project administrator and call the initialize_program function first, causing the protocol to be initialized with an incorrect admin address.

RECOMMENDATIONS:

Add proper authority validation to these functions to ensure only the legitimate protocol administrator can perform these critical initializations:

```
// Define the expected admin public key as a constant
const EXPECTED_ADMIN: Pubkey =
    solana_program::pubkey!("YourAdminPublicKeyHere");

#[derive(Accounts)]
pub struct InitializeProgram<'info> {
    #[account(
        init,
        payer = authority,
        // ... other constraints
    )]
    pub program_state: Account<'info, ProgramState>,

    #[account(
        mut,
        constraint = authority.key() == EXPECTED_ADMIN @
            StakingError::UnauthorizedAdmin
    )]
    pub authority: Signer<'info>,
}

#[derive(Accounts)]
pub struct InitializeEscrow<'info> {
    #[account(
```

```
        mut,
        seeds = [b"program-state"],
        bump,
        constraint = program_state.authority == authority.key() @
            StakingError::UnauthorizedAdmin
    )]
    pub program_state: Account<'info, ProgramState>,

    #[account(mut)]
    pub authority: Signer<'info>,
}
```

4. CONCLUSION

In this audit, we thoroughly analyzed **DefAI Staking** 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