ExVul

SMART CONTRACT AUDIT REPORT

DefAI Estate Smart Contract

SEPTEMBER 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **DefAI Estate** 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 Estate	03/09/2025 - 22/09/2025	Rust

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

Commit Hash
f77a7363f9405b26f757b6f5921ed5f9249ca2a7

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	4
MEDIUM	4
LOW	2
INFO	0

2.3 Key Findings

Severity	Findings Title	Status
HIGH	Estate and proposal do not correspond	Fixed
HIGH	Multi-signature calculation logic flaws	Fixed
HIGH	The close function does not perform permission verification	Fixed
HIGH	Premature closing leads to fund freezing	Fixed
MEDIUM	Multi-signature is invalid	Fixed
MEDIUM	Multi-signatures do not have deduplication	Fixed
MEDIUM	Lack of verification of the number of multi-signature members	Fixed
MEDIUM	Verification logic is inconsistent with the actual contract	Fixed
LOW	Administrator changes do not restrict who can execute	Fixed
LOW	asset_summary lacks update functionality	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Estate and proposal do not correspond

SEVERITY: HIGH **STATUS:** Fixed

PATH:

security-auditor/defai_estate/src/emergency.rs

DESCRIPTION:

When create_proposal is called, a proposal.target_estate is created to specify the estate affected by the proposal. However, in the actual ForceUnlockByMultisig, there is no verification of the estate and target_estate. This will result in two estates using the same multisig, then proposal A can also affect estate B.

```
//Archive/security-auditor/defai_estate/src/emergency.rs
#[derive(Accounts)]
pub struct ForceUnlockByMultisig<'info> {
    pub executor: Signer<'info>,

    #[account(
        mut,
        constraint = estate.is_locked @ EstateError::NotLocked,
        constraint = estate.multisig.is_some() @
   EstateError::NoMultisigAttached,
    )]
    pub estate: Account<'info, Estate>,

    #[account(
        mut,
        seeds = [b"emergency_lock", estate.key().as_ref()],
        bump = emergency_state.bump,
    )]
    pub emergency_state: Account<'info, EmergencyLockState>,

    #[account(
        constraint = multisig.key() == estate.multisig.unwrap() @
   EstateError::InvalidMultisig,
```

```
    )]
    pub multisig: Account<'info, crate::Multisig>,

    #[account(
        mut,
        constraint = proposal.multisig == multisig.key() @
    EstateError::InvalidProposal,
        constraint = proposal.executed @ EstateError::ProposalNotExecuted,
        constraint = matches!(proposal.action,
    crate::ProposalAction::EmergencyUnlock { .. }) @
    EstateError::InvalidProposalType,
    )]
    pub proposal: Account<'info, crate::Proposal>,

    pub clock: Sysvar<'info, Clock>,
}
```

IMPACT:

Proposal A affects estate B, causing a permission bypass issue.

RECOMMENDATIONS:

Added the function of checking target_estate and estate.

```
//Archive/security-auditor/defai_estate/src/emergency.rs
#[derive(Accounts)]
pub struct ForceUnlockByMultisig<'info> {
    ...

    #[account(
        mut,
        constraint = proposal.multisig == multisig.key() @
    EstateError::InvalidProposal,
        constraint = proposal.executed @ EstateError::ProposalNotExecuted,
        constraint = matches!(proposal.action,
    crate::ProposalAction::EmergencyUnlock { .. }) @
    EstateError::InvalidProposalType,
+       constraint = proposal.target_estate == estate.key() @
    EstateError::InvalidProposalEstate,
    )]
```

```
    pub proposal: Account<'info, crate::Proposal>,

    pub clock: Sysvar<'info, Clock>,
}
```

3.2 Multi-signature calculation logic flaws

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`security-auditor/defai_estate/src/emergency.rs`

DESCRIPTION:

There is no further constraint on the executor in the force_unlock_by_multisig function, which means anyone can call this function to perform the Unlock operation, causing the multi-signature mechanism to fail.

IMPACT:

An attacker can bypass the multi-signature verification mechanism and call force_unlock_by_multisig.

RECOMMENDATIONS:

Determine whether the executor is a proposer.

```
// security-auditor/defai_estate/src/emergency.rs
#[derive(Accounts)]
pub struct ForceUnlockByMultisig<'info> {
    pub executor: Signer<'info>,

    ...

    #[account(
        mut,
        constraint = proposal.multisig == multisig.key() @
    EstateError::InvalidProposal,
        constraint = proposal.executed @ EstateError::ProposalNotExecuted,
        constraint = matches!(proposal.action,
    crate::ProposalAction::EmergencyUnlock { .. }) @
    EstateError::InvalidProposalType,
+       constraint = proposal.proposer == executor.key() @
    EstateError::ProposerNotExecutor,
```

```
    )]
    pub proposal: Account<'info, crate::Proposal>,

    pub clock: Sysvar<'info, Clock>,
}
```

3.3 The close function does not perform permission verification

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`security-auditor/defai_estate/src/lib.rs`

DESCRIPTION:

The close_estate function allows closing an estate, but there is no permission check and it may be called arbitrarily.

```
// security-auditor/defai_estate/src/lib.rs
#[derive(Accounts)]
pub struct CloseEstate<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,

    #[account(
        mut,
        close = authority,
    )]
    pub estate: Account<'info, Estate>,
}
```

IMPACT:

The rent will be returned to any caller.

RECOMMENDATIONS:

Only estate.owner calls are allowed.

```
#[derive(Accounts)]
pub struct CloseEstate<'info> {
    #[account(mut)]
-   pub authority: Signer<'info>,
```

```
+   pub owner: Signer<'info>,

    #[account(
        mut,
-       close = authority,
+       close = owner,
+       has_one = owner
    )]
    pub estate: Account<'info, Estate>,
}
```

3.4 Premature closing leads to fund freezing

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`security-auditor/defai_estate/src/lib.rs`

DESCRIPTION:

When close is executed, it only checks whether claim_inheritance is fully called, but does not check whether the token and NFT are fully extracted.

```
// security-auditor/defai_estate/src/lib.rs
pub fn close_estate(ctx: Context<CloseEstate>) -> Result<()> {

    let estate = &ctx.accounts.estate;

    require!(estate.is_claimable, EstateError::NotClaimable);
    require!(
        estate.total_claims == estate.total_beneficiaries,
        EstateError::NotAllClaimed
    );

    msg!("Estate #{} closed", estate.estate_number);

    Ok(())
}
```

IMPACT:

If tokens and NFTs are not fully withdrawn, the funds will be frozen after closing.

RECOMMENDATIONS:

Check if estate_token_account and estate_nft_account are empty when executing close operation.

3.5 Multi-signature is invalid

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`security-auditor/defai_estate/src/lib.rs`

DESCRIPTION:

In the initialization of multisig, the threshold is incorrectly set to threshold > 0, which may make the threshold 1.

```
// security-auditor/defai_estate/src/lib.rs
pub fn initialize_multisig(
    ctx: Context<InitializeMultisig>,
    signers: Vec<Pubkey>,
    threshold: u8,
) -> Result<()> {
    ...
    require!(
        threshold > 0 && threshold as usize <= signers.len(),
        EstateError::InvalidThreshold
    );

    let multisig_key = ctx.accounts.multisig.key();

    ...
}
```

IMPACT:

When threshold == 1, multi-signature will be invalid.

RECOMMENDATIONS:

Fix the threshold judgment logic.

```
// security-auditor/defai_estate/src/lib.rs
pub fn initialize_multisig(
    ctx: Context<InitializeMultisig>,
    signers: Vec<Pubkey>,
    threshold: u8,
) -> Result<()> {
    ...
    require!(
-       threshold > 0 && threshold as usize <= signers.len(),
+       threshold > 1 && threshold as usize <= signers.len(),
        EstateError::InvalidThreshold
    );

    let multisig_key = ctx.accounts.multisig.key();

    ...
}
```

3.6 Multi-signatures do not have deduplication

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`security-auditor/defai_estate/src/lib.rs`

DESCRIPTION:

In initialize_multisig, a signer vector is passed in from outside as an allowed signature, but no deduplication is performed, which may lead to multi-signature verification later.

```
// security-auditor/defai_estate/src/lib.rs
pub fn initialize_multisig(
    ctx: Context<InitializeMultisig>,
    signers: Vec<Pubkey>,
    threshold: u8,
) -> Result<()> {
    require!(
        signers.len() >= MIN_SIGNERS && signers.len() <= MAX_SIGNERS,
        EstateError::InvalidSignerCount
    );
    require!(
        threshold > 0 && threshold as usize <= signers.len(),
        EstateError::InvalidThreshold
    );

    ...
}
```

IMPACT:

This may cause multi-signatures to be bypassed, resulting in logical errors.

RECOMMENDATIONS:

De-duplicate signers in initialize_multisig.

```
// security-auditor/defai_estate/src/lib.rs
pub fn initialize_multisig(
    ctx: Context<InitializeMultisig>,
    signers: Vec<Pubkey>,
    threshold: u8,
) -> Result<()> {
+   require!(
+       signers.iter().collect::<std::collections::HashSet<_>>().len() ==
   signers.len(),
+       EstateError::DuplicateSigner
+   );
    require!(
        signers.len() >= MIN_SIGNERS && signers.len() <= MAX_SIGNERS,
        EstateError::InvalidSignerCount
    );
    require!(
        threshold > 0 && threshold as usize <= signers.len(),
        EstateError::InvalidThreshold
    );

    ...
}
```

3.7 Lack of verification of the number of multi-signature members

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`security-auditor/defai_estate/src/emergency.rs`

DESCRIPTION:

In ForceUnlockByMultisig, there is no verification that the number of proposals is greater than or equal to the threshold. An attacker can pass in a proposal with insufficient multi-signature approvals to unlock.

IMPACT:

May cause the multi-signature function to fail.

RECOMMENDATIONS:

Add validation for approvals.

```
// security-auditor/defai_estate/src/emergency.rs
#[derive(Accounts)]
pub struct ForceUnlockByMultisig<'info> {
    ...
    #[account(
        mut,
        constraint = proposal.multisig == multisig.key() @
    EstateError::InvalidProposal,
        constraint = proposal.executed @ EstateError::ProposalNotExecuted,
        constraint = matches!(proposal.action,
    crate::ProposalAction::EmergencyUnlock { .. }) @
    EstateError::InvalidProposalType,

+       constraint = proposal.approvals.len() >= multisig.threshold as
    usize @ EstateError::NotEnoughApprovals,
    )]
    pub proposal: Account<'info, crate::Proposal>,
```

```
    pub clock: Sysvar<'info, Clock>,
}
```

3.8 Verification logic is inconsistent with the actual contract

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`security-auditor/defai_estate/src/lib.rs`

DESCRIPTION:

When calling update_beneficiaries to update the beneficiaries, the comment clearly states that either owner or multisig can perform this operation.

```
// security-auditor/defai_estate/src/lib.rs
pub fn update_beneficiaries(
    ctx: Context<UpdateBeneficiaries>,
    beneficiaries: Vec<Beneficiary>,
) -> Result<()> {
    ...
    // Check authorization - either owner or multisig
    let is_owner = ctx.accounts.owner.key() == estate.owner;
    let is_multisig = estate.multisig.is_some() &&
        ctx.accounts.owner.key() == estate.multisig.unwrap();

    require!(
        is_owner || is_multisig,
        EstateError::UnauthorizedAccess
    );
    ...
}
```

But in the account constraint, only estate.owner can use estate, which violates the design logic.

```
#[derive(Accounts)]
pub struct UpdateBeneficiaries<'info> {
    #[account(mut)]
    pub owner: Signer<'info>,

    #[account(
        mut,
```

```
        has_one = owner,
    )]
    pub estate: Account<'info, Estate>,
}
```

IMPACT:

This violates the design logic, resulting in multisig being unable to call this function.

RECOMMENDATIONS:

Delete the relevant checks in the account constraint, because there is already a more complete verification mechanism in the function.

```
#[derive(Accounts)]
pub struct UpdateBeneficiaries<'info> {
    #[account(mut)]
    pub owner: Signer<'info>,

    #[account(
        mut,
-       has_one = owner,
    )]
    pub estate: Account<'info, Estate>,
}
```

3.9 Administrator changes do not restrict who can execute

SEVERITY: LOW **STATUS:** Fixed

PATH:

`security-auditor/defai_estate/src/lib.rs`

DESCRIPTION:

Pending_admin is set after propose_admin_change, but the signer is not verified in accept_admin_change, which may allow others to impersonate and perform operations.

IMPACT:

No permission check means anyone can make changes to the admin process.

RECOMMENDATIONS:

Added Signer verification function to only allow pending_admin to call accept_admin_change.

```
// security-auditor/defai_estate/src/lib.rs
#[derive(Accounts)]
pub struct AcceptAdminChange<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,


-   #[account(mut)]
+   #[account(
+       mut,
+       constraint = multisig.pending_admin == signer.key()
+   )]
    pub multisig: Account<'info, Multisig>,
}
```

3.10 asset_summary lacks update functionality

SEVERITY: LOW **STATUS:** Fixed

PATH:

security-auditor/defai_estate/src/lib.rs

DESCRIPTION:

asset_summary is used to query and obtain relevant information about the current state. It is initialized by the seed containing estate.key to initialize the PDA and is subject to the account constraints of init. However, this will cause asset_summary to be unable to update.

```
// security-auditor/defai_estate/src/lib.rs
#[derive(Accounts)]
pub struct ScanEstateAssets<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,

    pub estate: Account<'info, Estate>,

    #[account(
        init,
        payer = authority,
        space = 8 + 32 + 8 + 8 + 4 + 4,
        seeds = [ASSET_SUMMARY_SEED, estate.key().as_ref()],
        bump
    )]
    pub asset_summary: Account<'info, AssetSummary>,

    pub system_program: Program<'info, System>,
}
```

IMPACT:

The data in asset_summary cannot be updated, which may cause the status to be inconsistent with the actual situation and affect normal operation.

RECOMMENDATIONS:

Use the correct init_if_needed constraint.

```
#[derive(Accounts)]
pub struct ScanEstateAssets<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,

    pub estate: Account<'info, Estate>,

    #[account(
-       init,
+       init_if_needed,
        payer = authority,
        space = 8 + 32 + 8 + 8 + 4 + 4,
        seeds = [ASSET_SUMMARY_SEED, estate.key().as_ref()],
        bump
    )]
    pub asset_summary: Account<'info, AssetSummary>,

    pub system_program: Program<'info, System>,
}
```

4. CONCLUSION

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

8. About Exvul Security

Premier Security for the Web3 Ecosystem

ExVul is a premier Web3 security firm committed to forging a secure and trustworthy decentralized ecosystem. Our elite team consists of security veterans from world-leading technology and blockchain security firms, including Huawei, YBB Captical, Qihoo 360, Amber, ByteDance, MoveBit, and PeckShield. Team member Nolan is ranked as a top-40 whitehat on Immunefi and is the platform's sole All-Star in the APAC region.

Our expertise covers the full spectrum of Web3 security. We conduct **meticulous smart contract audits**, having fortified thousands of projects on chains like Evm, Solana, Aptos, Sui etc. Our **Blockchain Protocol Audits** secure the core infrastructure of L1/L2 by uncovering deep-seated vulnerabilities. We also offer **comprehensive wallet audits** to protect user assets and provide **proactive web3 pentest**, enabling partners to neutralize threats before they strike.

Trusted by industry leaders, ExVul is the security partner for **OKX, Bitget, Cobo, Infini, Stacks, Aptos, Sui, CoreDAO, Sei** etc.

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec