Least Authority
PRIVACY MATTERS

Data Lake Token + Vesting Smart Contracts

Security Audit Report

Data Lake

Final Audit Report: 5 December 2022

Table of Contents

Overview

Background

Data Lake has requested that Least Authority perform a security audit of their smart contracts.

Project Dates

- **November 16 - 22:** Initial Code Review *(Completed)*
- **November 23:** Delivery of Initial Audit Report *(Completed)*
- **December 2-5:** Verification Review *(Completed)*
- **December 5:** Delivery of Final Audit Report *(Completed)*

Review Team

- Mukesh Jaiswal, Security Researcher and Engineer
- Ahmad Jawid Jamiulahmadi, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Data Lake smart contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- Data Lake Token App:
  https://github.com/Data-Lake-LLC/token-app

Specifically, we examined the Git revisions for our initial review:

- Token: 557cffd33bad80a5d48ac5141f4c50f90fe1fbaa

For the verification, we examined the Git revision:

fc627684cb4c84ac2605edcc0cc809a0f6c7dd56

For the review, these repositories were cloned for use during the audit and for reference in this report:

- Data Lakes Repository:
  https://github.com/LeastAuthority/Data-Lakes-Repository

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Medical Data for Research (Data Lake Website):
  https://data-lake.co/researchers-medical-data-for-research
- Data-Lake-Whitepaper-1.8-1.pdf *(shared with Least Authority via email on October 31, 2022)*
- Smart Contracts architecture.pdf *(shared with Least Authority via email on October 31, 2022)*

- Vesting logic.pdf *(shared with Least Authority via email on October 31, 2022)*
- Vesting Smart Contracts.pdf *(shared with Least Authority via email on October 31, 2022)*
- Data Lake Token App Introduction.pdf *(shared with Least Authority via email on October 31, 2022)*

In addition, this audit report references the following documents:
- Solidity Documentation Recommendations: https://docs.soliditylang.org/en/v0.8.17/contracts.html#errors-and-the-revert-statement

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS)/security exploits that would impact the intended use of the contracts or disrupt their execution;
- Vulnerabilities in the smart contracts' code;
- Protection against malicious attacks and other ways to exploit contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

The Data Lake smart contract suite composes the on-chain component of the Data Lake system, which aims to create an incentivized mechanism to store medical patient consent authorization data on-chain. In addition to the storage functionality, the smart contracts create the system token and facilitate token transactions and data flow transfer.

Our team performed a comprehensive review of the smart contracts, identified issues in the design and the implementation, and provided recommendations to address those issues. The nature of the issues and suggestions identified by our team, and detailed in this report, suggest a hasty implementation.

System Design

Our team found that security has been considered in the design of the Data Lakes smart contracts as demonstrated by the implementation of functionality to halt system operations if needed, and to remove participants and their tokens from the system at will. We found, however, that there is no check implemented to prevent the `TokenVesting` smart contract from being blacklisted by accident (Issue C).

In addition, there are liberal controls implemented on the vesting component of the system. All of this powerful functionality is controlled by a single owner address, a single point of failure to the system and its users. We recommend that the ownership of the contract be further secured (Issue E).

The smart contracts use the `Transfer` function, which is insufficiently secure, given the availability of safe libraries that prevent a transaction from failing silently (Issue F).

Code Quality

Our team performed a manual review of the Data Lake smart contracts and identified implementation errors (Issue A, Issue B, Issue G) and areas that can be improved by better adherence to best practice (Suggestion 3, Suggestion 4, Suggestion 7). We recommend improving the function and variable naming convention and removing unused variables to improve the readability of the code.

We also recommend that custom errors be used to improve the efficiency of the system and to reduce gas costs (Suggestion 6), as well as optimizing the use of for loops to reduce gas consumption (Suggestion 5).

Tests

Our team found that sufficient test coverage has been implemented for both repositories, which helps check for implementation errors that could lead to security vulnerabilities.

Documentation

The project documentation provided for this review included a helpful description of the architecture of the system. However, we found instances of the coded implementation being inconsistent with the documentation. We recommend including a more detailed description of the implementation, its components, and functions (Suggestion 2).

Code Comments

We found that while some functions have code comments describing the intended behavior, other functions are not described correctly. We recommend that code comments be consistent and describe all security-critical functions and components in accordance with NatSpec guidelines (Suggestion 1).

Scope

The scope of this review was sufficient and included all the on-chain security-critical components of the Data Lake system.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Incorrect Implementation of deleteBeneficiary	Resolved
Issue B: Incorrect Implementation of addVestingScheduleAllocation and removeVestingScheduleAllocation	Resolved
Issue C: BlackList Manager Can Blacklist TokenVesting Smart Contract	Resolved
Issue D: claimTokens Does Not Deduct the vestingSchedulesTotalAmount by the Amount Claimed (Known Issue)	Resolved
Issue E: Centralized Ownership of Smart Contracts	Resolved

Issue F: Unchecked Return Value from the ERC-20 Token Transfer	Resolved
Issue G: Missing Zero Check for terms in createVestingSchedule Function	Resolved
Suggestion 1: Update Code Comments to Reflect the Actual Implementation	Resolved
Suggestion 2: Update Documentation to Reflect the Implementation	Unresolved
Suggestion 3: Change Misleading Naming of Functions	Resolved
Suggestion 4: Set TGE Timestamp in the Constructor	Resolved
Suggestion 5: Cache the Length in for Loops	Resolved
Suggestion 6: Use Custom Errors to Save Gas	Resolved
Suggestion 7: Use an Updated and Non-Floating Pragma Version	Resolved

Issue A: Incorrect Implementation of deleteBeneficiary

Location

`contracts/TokenVesting.sol#L395-L403`

Synopsis

In the `deleteBeneficary` function, there is no check to test if `_beneficiary` is not a zero address. There is also no check to verify if the beneficiary being deleted actually exists. In both cases, the function loops through the entire `vestingSchedulesNames` array needlessly. Additionally, if the `_beneficiary` does exist, the function loops through the `vestingScheduleNames` array rather than those related to the particular beneficiary only.

In the zero address case, a new zero address beneficiary is added to the mapping of beneficiaries for every `vestingScheduleName`, if it is the first time calling with a zero address. Otherwise, it is an unnecessary update of a previously added zero address.

For the non-existing address case, a new beneficiary is added to the mapping of beneficiaries for every `vestingScheduleName` with that address if it is the first time calling with a non-existing address. Otherwise, it is an unnecessary update of a previously added non-existing address.

Additionally, when there is no beneficiary for any `vestingSchedulesNames[i]`, it redundantly performs operations, and another beneficiary may be added for `vestingSchedulesNames[i]`.

Impact

In the first two cases, an incorrect event is emitted at the end of the function because it did not actually delete anything but rather added to the mapping. In every case, it results in the further consumption, and eventual loss, of gas. Additionally, the incorrect event results in misinterpretation of the state of the contract for external entities (whoever is watching for that event).

Preconditions

This issue is possible if the `_beneficiary` is a zero address or if there is no beneficiary with such an address. It is also possible if the beneficiary has no allocation in a `VestingSchedule`, which has `vestingSchedulename = vestingSchedulesNames[i]`.

Mitigation

We recommend implementing a check that the `_beneficiary` is not a zero address. Additionally, as suggested by the Data Lake team during the audit, we recommend finding the list of active vesting schedule names, checking the length of the list, and reverting if the length of the list is not greater than zero.

Moreover, we recommend only looping through those `vestingScheduleNames`, which are related to the particular beneficiary using active vesting schedule names to avoid redundancy. The implementation could be as follows:

```
require(_beneficiary != address(0x0));

string[] memory scheduleNames =
getBeneficiaryActiveScheduleNames(_beneficiary);

uint32 activeSchedulesLength = scheduleNames.length;

require(activeSchedulesLength > 0);

for (uint32 i = 0; i < activeSchedulesLength; i++) {

            uint256 unreleasedAmount =
getBeneficiaryUnreleasedAmount(_beneficiary, vestingSchedulesNames[i]);

            beneficiaries[_beneficiary][scheduleNames[i]].allocatedAmount -=
unreleasedAmount;

            vestingSchedules[scheduleNames[i]].allocatedAmount -=
unreleasedAmount;

        }
```

Status

The recommended check has been implemented in the `deleteBeneficary` function.

Verification

Resolved.

Issue B: Incorrect Implementation of addVestingScheduleAllocation and removeVestingScheduleAllocation

Location

[contracts/TokenVesting.sol#L290-L294]

[contracts/TokenVesting.sol#L301-L305]

Synopsis

The above-referenced functions are missing necessary checks and adjustments or execute incorrect adjustments:

1. The `addVestingScheduleAllocation` function does not check if the contract has enough unused balance to dedicate to the particular vesting schedule: `require(getUnusedAmount() >= _amount)`. As a result, the amount of balance dedicated to all vesting schedules could exceed the smart contract's balance, rendering the smart contract unable to pay for all vesting schedules;
2. The `removeVestingScheduleAllocation` function does not check if the schedule has enough unallocated tokens to deduct from: `require(getScheduleUnallocatedAmount(_name) >= _amount)`. If the amount deducted is more than a particular vesting schedule's unallocated amount, the smart contract would lose track of the `vestingSchedulesTotalAmount`, assuming the amount allocated to beneficiaries is less than what it actually is, which may result in it being unable to pay for all vesting schedules;
3. Both functions do not check if a `VestingSchedule` function with the particular `_name` argument exists. In the case of `addVestingScheduleAllocation`, the operation would pass and a new `VestingSchedule` would be added to the `vestingSchedules` mapping, with `vestingSchedule.totalAmount = _amount`. In the case of `removeVestingScheduleAllocation`, it would revert due to an unpredicted underflow: `vestingSchedules[_name].allocatedAmount -= _amount`
4. Both functions incorrectly adjust `vestingSchedules[_name].allocatedAmount` instead of `vestingSchedules[_name].totalAmount`. This might lead to the track of the actual `vestingSchedule.allocated` amount being lost, resulting in incorrect calculations in the functions `getScheduleLockedAmount` and `getScheduleUnallocatedAmount`. Consequently, the `claimTokens` function could behave unexpectedly; and
5. Both functions are missing the necessary adjustments of `vestingSchedulesTotalAmount` by the amount increased or decreased:
   `vestingSchedulesTotalAmount += _amount;`

   `vestingSchedulesTotalAmount -= _amount;`

   In both increase and decrease cases, the `vestingSchedulesTotalAmount` would indicate an incorrect state of the amount reserved for all vesting schedules, resulting in the `getUnusedAmount` function incorrectly calculating the remaining balance to be allocated for vesting schedules.

Mitigation

We recommend implementing these functions as follows:

`addVestingScheduleAllocation:`

```
function addVestingScheduleAllocation(uint256 _amount, string calldata _name)
external onlyOwner {

        require(!paused, 'TokenVesting: Contract is paused!');

        require(_amount != 0, 'TokenVesting: amount must be bigger than
zero');

        //This line checks if the vesting schedule is valid (exists)
        require(isVestingScheduleValid(_name), "Vesting schedule is not 
valid");



        //Add this check to ensure the contract has enough unused balance

        require(getUnusedAmount() >= _amount, "_amount cannot be greater the 
unused balance");



       //Add the following line and remove the previous line to adjust 
vestingSchedule's totalAmount instead of allocatedAmount.

        vestingSchedules[_name].totalAmount += _amount;



        //Add this line to perform necessary adjustment of 
vestingSchedulesTotalAmount

        vestingSchedulesTotalAmount += _amount;

    }

RemoveVestingScheduleAllocation:

function removeVestingScheduleAllocation(uint256 _amount, string calldata 
_name) external onlyOwner {

        require(!paused, 'TokenVesting: Contract is paused!');

        require(_amount != 0, 'TokenVesting: amount must be bigger than 
zero');

        //This line checks if the vesting schedule is valid (exists)

        require(isVestingScheduleValid(_name), "Vesting schedule is not 
valid");



        //Add this check to ensure the schedule has enough amount of 
unallocated tokens to deduct from

        require(getScheduleUnallocatedAmount(_name) >= _amount, 'TokenVesting: 
amount cannot be greater than vesting schedule unallocated amount');

        //Add the following line and remove the previous line to adjust 
vestingSchedule's totalAmount instead of allocatedAmount.

        vestingSchedules[_name].totalAmount -= _amount;

        //Add this line to perform necessary adjustment of 
vestingSchedulesTotalAmount
```

```
        vestingSchedulesTotalAmount -= _amount;
    }
```

Status
The referenced functions have been reimplemented according to the recommendation.

Verification
Resolved.

Issue C: BlackList Manager Can Blacklist TokenVesting Smart Contract

Location
[contracts/access/BlacklistManager.sol#L68-L74]

Synopsis
In the `addToBlacklist` function, there is no check to verify if the address being blacklisted is not the `TokenVesting` smart contract.

Impact
If this happens for any reason, the `TokenVesting` smart contract would not be able to send $LAKE tokens until it is removed from the blacklist.

Mitigation
We recommend adding a check in the `addToBlackList` function to identify that the blacklisted address is not the `TokenVesting` smart contract address.

Status
A check has been implemented to the `addToBlacklist` function.

Verification
Resolved.

Issue D: claimTokens Does Not Deduct the vestingSchedulesTotalAmount by the Amount Claimed (Known Issue)

Location
[contracts/TokenVesting.sol#L482-L507]

Synopsis
The Data Lake developer team raised this issue during the audit and noted that the allocated tokens, which had been claimed, were not deducted from the `vestingSchedulesTotalAmount`.

Impact
If the allocated tokens, which were claimed, were not deducted from the `vestingSchedulesTotalAmount`, this could prevent the creation of new vesting schedules and/or allocations.

Remediation

The Data Lake team resolved this issue by deducting the `vestingSchedulesTotalAmount` by the amount claimed.

Status

The Data Lake smart contracts team resolved the issue.

Verification

Resolved.

Issue E: Centralized Ownership of Smart Contracts

Location

Example (non-exhaustive):

`contracts/TokenVesting.sol#L831-L835`

Synopsis

The smart contracts in scope, which include the functions for changing the owner, are heavily dependent on the owner.

Impact

If the owner is set incorrectly, the functionality that depends on it will break and negatively impact the protocol, rendering it almost unusable.

Mitigation

We recommend that a two-step process be used whereby in the first step, a new owner is proposed, and in the second step, the previous owner remains in charge until the newly appointed owner claims the owner rule.

Status

The recommended mitigation has been implemented.

Verification

Resolved.

Issue F: Unchecked Return Value from the ERC-20 Token Transfer

Location

`contracts/tokenvesting#L141`

Synopsis

The return value from the token transfer should be checked to verify that the transaction has been executed successfully.

Impact

This issue could result in the ERC-20 Token transfer failing silently, thereby affecting the token accounting process.

Mitigation

We recommend the use of `SafeTransfer` from the OpenZepplin library.

Status

The use of the recommended library has been implemented.

Verification

Resolved.

Issue G: Missing Zero Check for terms in createVestingSchedule Function

Location

`contracts/TokenVesting.sol#L188-L219`

Synopsis

In the `createVestingSchedule` function, there is no check to verify if the `terms` argument being passed is not zero. Consequently, a `vestingSchedule` with zero `terms` could be added.

Impact

This would result in incorrect calculations in every function which uses `vestingSchedule.terms`. The following are two prominent cases:

First, the `isVestingScheduleFinished` function would immediately return `true` after the `vestingSchedule.cliff` time has passed without taking the `vestingSchedule.duration` into account.

Second, the `getPassedVestings` function will always revert due to a division by zero.

Preconditions

The passed `terms` argument to the `createVestingSchedule` function is zero.

Mitigation

We recommend adding a zero check for `terms` in the `createVestingSchedule` function.

Status

A zero check has been implemented in the `createVestingSchedule` function as recommended.

Verification

Resolved.

Suggestions

Suggestion 1: Update Code Comments to Reflect the Actual Implementation

Location

Examples (Non-exhaustive):

`contracts/LakeToken.sol#L30-L43`

contracts/access/BlacklistManager.sol#L20-L22

contracts/access/BlacklistManager.sol#L28-L30

contracts/access/BlacklistManager.sol#L35-L38

contracts/access/BlacklistManager.sol#L44-L47

Synopsis
There are many instances of functions with comments that do not reflect the actual implementation of the functions.

Mitigation
We recommend updating code comments to describe functions as implemented.

Status
The code comments have been updated as recommended.

Verification
Resolved.

Suggestion 2: Update Documentation to Reflect the Implementation

Synopsis
The project documentation provided for this review was not up to date, and our team found many discrepancies between the provided documentation and the actual implementation.

Mitigation
We recommend updating the documentation to reflect the actual implementation.

Status
The documentation has not been updated as recommended, as of the verification phase of this review.

Verification
Unresolved.

Suggestion 3: Change Misleading Naming of Functions

Location
contracts/TokenVesting.sol#L290-L294

contracts/TokenVesting.sol#L301-L305

Synopsis
The naming of the above-referenced functions is misleading and does not clearly describe their intended behavior.

Mitigation

We recommend that `addVestingScheduleAllocation` be changed to `increaseVestingScheduleTotalAmount` and `removeVestingScheduleAllocation` be changed to `decreaseVestingScheduleTotalAmount`.

Status

The names of the referenced functions have been updated as recommended.

Verification

Resolved.

Suggestion 4: Set TGE Timestamp in the Constructor

Location

[`contracts/TokenVesting.sol#L174-L178`]

Synopsis

Since the `setTgeTimestamp` function is only called once during the deployment, there is no need for such a function, and it can be set in the constructor.

Mitigation

We recommend removing the redundant function as well as the `TgeTimestampChanged` event declaration and only setting the `tgeTimestamp` function in the constructor.

Status

The `tgeTimestamp` function is now only set in the constructor.

Verification

Resolved.

Suggestion 5: Cache the Length in for Loops

Location

Example (non-exhaustive):

`Contracts/TokenVesting#L583`

Synopsis

In many sections of the code, the "for loop" is implemented so that the Solidity compiler will always read the length of the array during each iteration. As a result:

1. If it is a `storage` array, this is an extra `sload` operation (100 additional extra gas, according to EIP-2929, for each iteration except for the first);
2. If it is a `memory` array, this is an extra `mload` operation (3 additional gas for each iteration except for the first); and
3. If it is a `calldata` array, this is an extra `calldataload` operation (3 additional gas for each iteration except for the first).

Mitigation
We recommend avoiding these extra costs by caching the array length (in stack), as follows:

```
uint length = arr.length;

for (uint i = 0; i < length; i++) {

    // do something that doesn't change arr.length

}
```

In the above example, the `sload`, `mload`, or `calldataload` operation is only called once and is subsequently replaced by a more efficient dupN instruction. Even though `mload`, `calldataload`, and dupN have the same gas cost, `mload` and `calldataload` need an additional dupN to put the offset in the stack, i.e., an extra 3 gas.

This optimization is especially important in the case of a storage array or lengthy loops.

Status
The suggested optimization has been implemented.

Verification
Resolved.

Suggestion 6: Use Custom Errors to Save Gas

Location
[Contract/TokenVesting#L523]

Synopsis
The codebase currently uses `require` and `revert` string messages for error handling. However, using a `revert` with a custom `Error` in transactions, instead of a string error message, considerably optimizes gas costs, according to [Solidity documentation recommendations].

Mitigation
We recommend using a `revert` with a custom `Error` to handle errors.

Status
Custom errors have been implemented as recommended.

Verification
Resolved.

Suggestion 7: Use an Updated and Non-Floating Pragma Version

Synopsis
Smart contracts in the project have their pragma set to ^0.8.0. Compiling with different versions of the compiler might lead to unexpected results. In addition, older versions of the Solidity compiler contain bugs that have been fixed in more recent versions of the compiler, including up-to-date security patches.

Mitigation

In order to maintain consistency and to prevent unexpected behavior, we recommended that the Solidity compiler version be pinned by removing "^" and using the latest version of the Solidity compiler.

Status

The Solidity compiler pragma has been updated and pinned as recommended.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, and zero-knowledge protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.