contact@testmachine.ai

TestMachine + Legion

Predator Campaign 1

21 November 2024

1. / Overview 3

2. / How Does TestMachine Work? 3

3. / Campaign Structure 6

4. / Findings 7

4.1. Token Custody Risks 7

4.2. Overflow Panics 9

5. / Engagement Metrics 10

5.1. Function Calls 11

5.2. Transaction Counts 13

5.3. Reward Function 15

5.4. Observation Space 16

5.5. Unique Storage Mutations 17

6. / Future Efforts 18

6.1. Deeper Engagement with Stages of Sale Instances 19

6.2. Continue Evaluation of Token Custody Risks 19

6.3. Custom Value Drain Invariants 19

7. / Legal 20

Summary

1. / Overview

From October 21, 2024 to November 21, 2024, TestMachine's Predator software conducted a security assessment of the Legion protocol with an emphasis on its sale instances and integration with ERC20 tokens.

To do this, Predator ran 2 types of campaigns with distinct objectives:

1. Qualify technical risks due to Legion sale instances' integrations with tokens
2. Meaningfully engage with simulated sale instances as much as possible with pre-selected generalized invariants and informational findings

The Legion sale contracts (LegionFixedPriceSale, LegionPreLiquidSale, and LegionSealedBidAuction) and related dependencies (testnet ERC20 bid and ask tokens picked by TestMachine) were configured in our high-fidelity testing environment. After onboarding, the Predator algorithm initiated its testing campaign, using reinforcement learning guided fuzzing to perform penetration testing. As Predator receives feedback from the environment, it adjusts its internal policy to subsequently generate actions that will lead to novel states. Predator periodically re-checks the invariants against the environment to see if any are broken.

Our tools did not uncover any serious bugs, vulnerabilities, or other unauthorized privilege escalation in Legion's sale contracts.

Key Predator Results

- 10,536,085 transactions executed across 5 campaigns;
- Blacklisting, Minting, and Management Invariants triggered on a chosen AskToken (https://sepolia.arbiscan.io/address/0x20D2A6D53eE43b6758AB16D3F4949A5110876B70#code), with severe implications on a Legion sale if hypothetical project admin is malicious or project has common ERC20 functionality that can impact token custody;
- Solidity panic exception code 0x11 (overflow resulting in reverted transactions) encountered on potential inputs to LegionFixedPriceSale's publishSaleResults and LegionPreLiquidSale's withdrawExcessCapital functions. Upon further review, these findings were determined to be informational;
- Future campaigns will more deeply engage with stages of sale instances and attempt to drain tokens from funded vesting contracts.

Methodology

2. / How Does TestMachine Work?

Most existing smart contract security solutions are either based solely around static analysis of code or employ a rudimentary fuzzing campaign based on superficial metrics. TestMachine breaks the mold with Predator, an AI-driven solution that harnesses the power of deep reinforcement learning to dynamically adapt penetration testing to the unique needs of your protocol.

Predator treats experimental runs, or *campaigns*, as a Markov Decision Process. Individual smart contract transactions are actions that cause an observable change in the environment. A carefully curated reward function evaluates the "value" of environment states as they pertain to Predator's goal: breaking contract invariants. Signals from the reward function update Predator's policy, allowing it to choose actions and develop plans with more acuity.

At the conclusion of each campaign, we generate a comprehensive report detailing our findings, including any identified vulnerabilities, their potential impact, and recommendations for remediation. We also provide a range of visual aids and analytical tools, such as function call histograms and state transition graphs, to help Legion better understand the results of our testing process.

TestMachine's Predator platform offers unparalleled insights into the security and reliability of smart contracts. Our methodology enables us to identify and mitigate vulnerabilities that would be missed by traditional auditing techniques, providing the Legion team with the highest level of assurance and protection for their digital assets.

Table 1: Summary of Key Campaign Metrics

Transactions	Predator conducted 10,536,085 transactions.
Invariant Break	Blacklist, Management, Minting broken in context of AskToken

Engagement	Predator had most success probing the following contracts: LegionFixedPriceSale LegionPreLiquidSale LegionSealedBidAuction MockBidToken AskToken
Security	Predator's results provide a deeper level of confidence in the tested contracts.

3. / Campaign Structure

This section outlines the specific contracts and code repositories that were the focus of our testing efforts, the invariants and reward functions that guided our AI agents' exploration of the contract's state space, and the timeline and duration of the campaign. By presenting this information in a clear and structured format, we aim to provide the Legion team with a comprehensive understanding of the scope, methodology, and results of our security assessment.

Structure

Table 2: Overview of Predator structure for current campaign against Legion

Target Platform	Legion: A token sale protocol whose setup is defined in the evm-contracts repo in test/LegionFixedPriceSale.t.sol, test/LegionPreLiquidSale.t.sol, and test/LegionSealedBidAuction.t.sol The arbitrum sepolia network was forked at blockheight 88712500 and utilized the LegionSaleFactory instance deployed at https://sepolia.arbiscan.io/address/0x9c68A0F9AE106A49B27e2516b6Aa7c1755B66f10#code As well as a 6 decimal bid token chosen by TestMachine https://sepolia.arbiscan.io/address/0xd11406c415436644678dA07cDe450A03Bd7410e2#code And an 18 decimal ask token chosen by TestMachine https://sepolia.arbiscan.io/address/0x20D2A6D53eE43b6758AB16D3F4949A5110876B70#code
Timeline	10.21.2024 – 11.21.2024
Languages	Python; Rust; Solidity
Methods	Reinforcement Learning (RL)
Invariants	Blacklist, Minting, Management
Contracts	**LegionFixedPriceSale;LegionPreLiquidSale;LegionSealedBidAuction;MockBidToken;AskToken**

Engagement

4. / Findings

Predator can identify custody-impacting risks in token contracts. In the campaigns run, a bid and ask token were selected by TestMachine and used to engage with the sale instances. Campaigns were run on these tokens in isolation to highlight and validate the risks associated with integrating with external projects. We believe that Predator adds a necessary layer of security when determining which tokens to bring on to the Legion protocol, which is validated by our current findings.

Engagement with the sale instances identified multiple Solidity panic conditions related to integer overflows. After Solidity 0.8, overflows that occur in solidity (except within "unchecked" blocks of code) return an "0x11" code and revert (see https://docs.soliditylang.org/en/latest/control-structures.html#panic-via-assert-and-error-via-require).

Both of these findings are considered informational, although Predator has experience vetting tokens and can quickly identify custody impacting risks.

4.1. Token Custody Risks

Predator ran 2 separate campaigns specifically targeting custody risks. One targeting risks on the ask token (controlled by the project admin actor), and another targeting risks on the bid token (controlled by an external actor).

The ask token used in all campaigns (https://sepolia.arbiscan.io/token/0x20D2A6D53eE43b6758AB16D3F4949A5110876B70#code), flagged Predator's Blacklist, Management, and Minting risks.

Thistoken requires the use of a privileged "TRANSFER_ROLE" necessary for a sender to transfer tokens. This transfer role implements a whitelist: the AskToken's admin must grant the role to any address before it can transfer tokens. This role can be granted or revoked arbitrarily by the ask token project's admin. This functionality was flagged by Predator's Blacklist invariant in Figure 1.

Access Control Lists (ACLs) in an AskToken present a serious risk for Legion, as the token admin could easily change Legion's access to the token, possibly during a sale. Careful measures must be taken to guarantee that Legion's access to any given token is not compromised throughout the lifecycle of a Legion sale.

A Legion sale could be further invalidated or attacked through the Management and Mint functionalities identified by Predator. Admins are able to flexibly assign and revoke roles to themselves and others. Strict guarantees must be required throughout the lifecycle of a sale to manage the risk inherent in these capabilities. Minting could invalidate a Legion sale by sending tokens directly to an attacking address. Users could be locked out of vesting funds with no recourse by a malicious admin blacklisting a vesting contract and revoking the admin role from themselves.

In total, these findings highlight Legion's vulnerability to attacks from the external token contracts, and must take careful steps to guarantee that a token contract will not behave maliciously during the lifecycle of a sale. These risks can be effectively controlled through visibility into the token's behavior, which Predator is adept at providing.

File 1 of 45 : TokenERC20.sol

```
135     /// @dev Runs on every transfer.
136     function _beforeTokenTransfer(address from, address to, uint256 amount) internal override {
137         super._beforeTokenTransfer(from, to, amount);
138
139         if (!hasRole(TRANSFER_ROLE, address(0)) && from != address(0) && to != address(0)) {
140             require(hasRole(TRANSFER_ROLE, from) || hasRole(TRANSFER_ROLE, to), "transfers restricted.");
141         }
142     }
```

AskToken	transfer	transfers restricted.	12
AskToken	transferFrom	ERC20: insufficient allowance	3613
AskToken	transferFrom	transfers restricted.	85


```
{
    "tx_to": "0x20d2a6d53ee43b6758ab16d3f4949a5110876b70",
    "step_id": 5,
    "tx_data": "0xd547741f8502233096d909befbda0999bb8ea2f3a6be3c138b9fbf003752a4c8bce86f6c0000000000000000000000001111111111111111111111111111111111111111",
    "tx_from": "0xd625fc559d40f3334872c1cdd44b3cc4a690b409",
    "tx_halt": false,
    "contract": "AskToken",
    "function": "revokeRole",
    "tx_value": "0",
    "arguments": [
        "0x8502233096d909befbda0999bb8ea2f3a6be3c138b9fbf003752a4c8bce86f6c",
        "0x1111111111111111111111111111111111111111"
    ],
    "tx_result": "0x",
    "tx_revert": false,
    "parameters": [
        "bytes32",
        "address"
    ]
}
```


Figure 1: The whitelisting logic required for successful transfers of AskTokens in the logic, output from Predator that triggered the Blacklist invariant after an agent's address had its TRANSFER_ROLE revoked

The Bid token used in all campaigns (https://sepolia.arbiscan.io/address/0xd11406c415436644678dA07cDe450A03Bd7410e2#code), was found to only be able to have its ownership changed, with no further implications. Despite functionality existing that can levy transfer fees and transfer limit amounts existing within the source code, this functionality is unreachable by a superuser and is not a risk.

MockBidToken	setMaxTokenAmountPerAddress	Ownable: caller is not the owner	7596
MockBidToken	setMaxTokenAmountPerAddress	MaxTokenAmountNotAllowed()	4474
MockBidToken	setTaxConfig	Ownable: caller is not the owner	5270
MockBidToken	setTaxConfig	TokenIsNotTaxable()	3481

Figure 2: Output from Predator that shows failures from attempting to levy a transaction fee or token amount limit on the bid token

4.2. Overflow Panics

Overflow panics are thrown and cause a transaction to revert when inputs go outside the bounds of a type. After Solidity version 0.8, these cases simply cause the transaction to fail. While these findings are informational, validating inputs and throwing a custom error is typically preferable.

Predator encountered overflow panics in LegionFixedPriceSale's publishSaleResults function from calculating totalCapitalRaised (Figure 3, line 160) and LegionPreLiquidSale's withdrawExcessCapital function (Figure 4,lines 535 and 538).

```
137         /**
138          * @notice See {ILegionFixedPriceSale-publishSaleResults}.
139          */
140         function publishSaleResults(bytes32 merkleRoot, uint256 tokensAllocated, uint8 askTokenDecimals)
141             external
142             onlyLegion
143         {
144             /// Verify that the sale is not canceled
145             _verifySaleNotCanceled();
146
147             /// Verify that the refund period is over
148             _verifyRefundPeriodIsOver();
149
150             /// Verify that sale results are not already published
151             _verifyCanPublishSaleResults();
152
153             /// Set the merkle root for claiming tokens
154             claimTokensMerkleRoot = merkleRoot;
155
156             /// Set the total tokens to be allocated by the Project team
157             totalTokensAllocated = tokensAllocated;
158
159             /// Set the total capital raised to be withdrawn by the project
160             totalCapitalRaised = (tokensAllocated * tokenPrice) / (10 ** askTokenDecimals);
161
162             /// Emit successfully SaleResultsPublished
163             emit SaleResultsPublished(merkleRoot, tokensAllocated);
164         }
```

Figure 3: LegionFixedPriceSale's publishSaleResults function where Predator would sometimes encounter overflow panics

```
519          * @notice See {ILegionPreLiquidSale-withdrawExcessCapital}.
520          */
521         function withdrawExcessCapital(
522             uint256 amount,
523             uint256 saftInvestAmount,
524             uint256 tokenAllocationBps,
525             bytes32 saftHash,
526             bytes32[] calldata proof
527         ) external {
528             /// Verify that the sale has not been canceled
529             _verifySaleNotCanceled();
530 
531             /// Load the investor position
532             InvestorPosition storage position = investorPositions[msg.sender];
533 
534             /// Decrement total capital invested from investors
535             totalCapitalInvested -= amount;
536 
537             /// Decrement total investor capital for the investor
538             position.investedCapital -= amount;
539 
540             /// Cache the maximum amount the investor is allowed to invest
541             if (position.cachedSAFTInvestAmount != saftInvestAmount) {
542                 position.cachedSAFTInvestAmount = saftInvestAmount;
543             }
544 
545             /// Cache the token allocation in BPS
546             if (position.cachedTokenAllocationBps != tokenAllocationBps) {
547                 position.cachedTokenAllocationBps = tokenAllocationBps;
548             }
549 
550             /// Cache the hash of the SAFT signed by the investor
551             if (position.cachedSAFTHash != saftHash) {
552                 position.cachedSAFTHash = saftHash;
553             }
554 
555             /// Verify that the investor position is valid
556             _verifyValidPosition(msg.sender, proof);
557 
558             /// Emit successfully ExcessCapitalWithdrawn
559             emit ExcessCapitalWithdrawn(amount, msg.sender, tokenAllocationBps, saftHash, block.timestam
560 
561             /// Transfer the excess capital to the investor
562             IERC20(bidToken).safeTransfer(msg.sender, amount);
563         }
```

Figure 4: LegionPreLiquidSale's withdrawExcessCapital function where Predator would sometimes encounter overflow panics

5. / Engagement Metrics

When we say that TestMachine "engages" with a smart contract, we refer to the process of our AI agents actively interacting with the contract's code in a simulated environment. This engagement is a critical aspect of our vulnerability discovery methodology, as it allows our agents to explore the

contract's behavior and uncover potential security issues that might not be apparent from static analysis alone.

In the context of our Predator platform, engagement involves our reinforcement learning (RL) agents executing contract transactions, observing the resulting changes in the contract's state, and adapting their strategies based on the feedback they receive. This iterative process allows our agents to build a comprehensive understanding of the contract's functionality and identify any unintended or malicious behaviors.

The importance of engagement in identifying vulnerabilities lies in the fact that many security issues in smart contracts arise from the complex interactions between different functions, state variables, and external factors such as user inputs or timing constraints. By actively engaging with the contract, our agents can simulate these interactions and uncover edge cases or exploitation scenarios that would be difficult to predict or detect through manual code analysis.

5.1. Function Calls

The histogram of function calls, which displays the number of calls made by Predator to each function within the sale contracts and whether those calls result in a revert or successful execution, is a valuable tool for assessing the effectiveness and thoroughness of Predator's engagement with the protocol. The histogram is depicted in Figure 5, below.

This plot provides several key insights into Predator's testing process and the potential vulnerabilities or areas of concern within the sale contracts:

Function coverage: By visualizing the distribution of function calls, the histogram allows us to quickly identify which functions are being heavily targeted by Predator and which ones may be receiving less attention. This information is crucial for ensuring that the testing process is comprehensive and that all relevant functions are being adequately explored for potential vulnerabilities.
Revert rate: The histogram also highlights the proportion of function calls that result in a revert, which can be an indicator of potential issues or unexpected behaviors within the contract. A high revert rate for a particular function may suggest the presence of input validation errors, edge cases, or other vulnerabilities that are causing the function to fail under certain conditions.
Adaptive learning: As Predator continues to interact with the sale contracts, we would expect to see changes in the distribution of function calls over time. If the histogram shows a shift towards more successful calls and a lower revert rate, this may indicate that Predator is learning to generate more targeted and effective test cases, adapting its strategy based on the feedback it receives from the contract.

Function Histogram

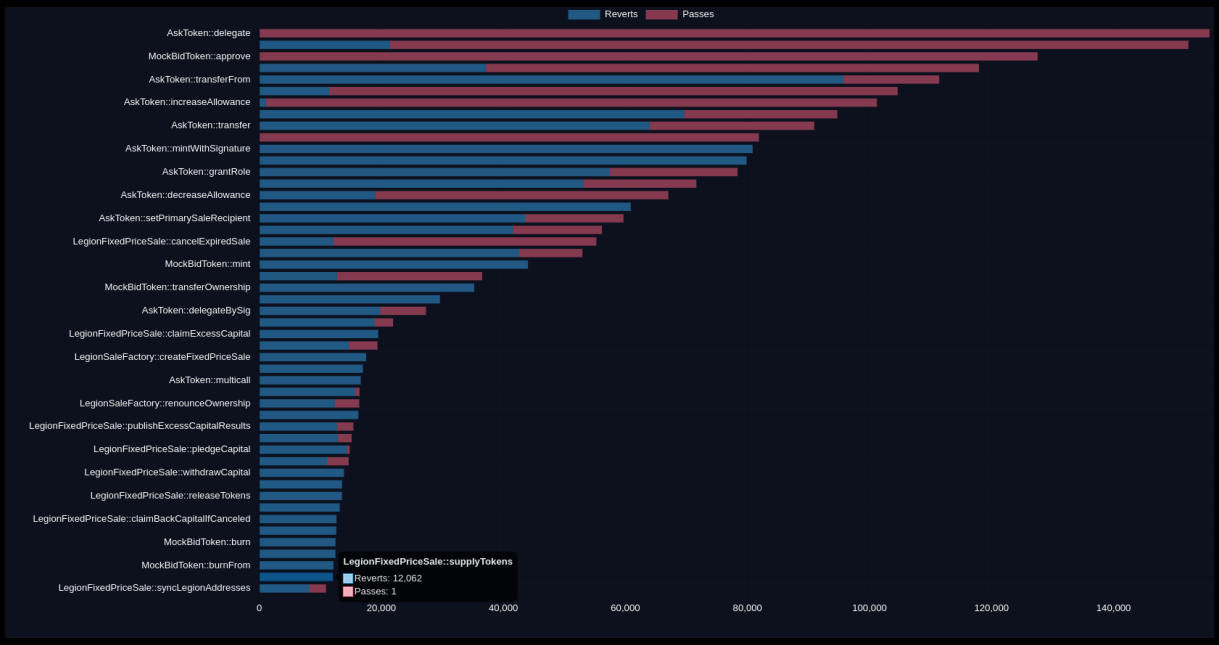


Figure 5: Function call histogram for a campaign with the Legion protocol (LegionFixedPriceSale). Red bars indicate function calls that passed, while blue bars indicate function calls that caused a contract revert. Over time, this chart evolves toward more red bars, indicating that Predator is learning how to better engage with the target contracts.

Figure 5 details all transactions made by the Predator system on a campaign that targeted a deployed instance of the LegionFixedPrice sale. Reverts are in blue, and passes are in red. In this campaign, the agent (controlling the ask token's project admin, legion admin, a whitelisted investor, and an unprivileged account) was able to successfully call pledgeCapital, publish sale results, and even succeeded at supplying tokens to the vested contract, although it never succeeded at fully vesting or claiming its tokens.

Initially, Predator had limited success at getting pledgeCapital functionality to pass, this is because at the beginning of the campaign Predator randomly chose actions with little indication of what would be successful, it also needed to utilize the Legion signer. Over the course of the campaign however, the agent learned that more functionality of the contract was reachable when certain actions were performed during a campaign. The agent was able to formulate thousands of transaction sequences that allowed passing on several of the LegionFixedPriceSale contracts state-mutable functions.

The number of passes on the LegionFixedPriceSale contract's investor functions remain small relative to the reverts due to the complexity of pledging capital, publishing sale results, supplying AskTokens and claiming them, however it is believed that the amount of passes would increase relative to reverts if the agent had even more time to engage with the contracts due to the reward function incentivizing actions that lead to successful passes. Improving engagement on these functions will be a primary focus of future work.

Transactions

5.2. Transaction Counts

Predator executed 10,536,08 transactions across the campaigns we ran. The total number of transactions executed during a TestMachine campaign is a critical metric for assessing the thoroughness and effectiveness of our vulnerability discovery process. In the case of the Legion campaign with all deployed sale instances, we are reporting this metric to provide the Legion team with a clear understanding of the scale and depth of our testing efforts. By tracking the total number of transactions, we can gauge the extent to which Predator has explored the various functionalities and pathways within each Legion contract.

In the Legion campaign, we made the deliberate choice to limit the number of transactions per sequence to ten before restarting the simulation. This decision was based on several key considerations:

Efficiency: By setting a limit on the number of transactions per episode, we ensure that Predator is not wasting computational resources on excessively long or unproductive sequences of interactions. This allows us to cover a broader range of scenarios and test cases within a given timeframe.
Realistic scenarios: Limiting the transaction count per episode helps to mimic realistic user behavior and interaction patterns. In most real-world scenarios, users are unlikely to engage in extremely long chains of uninterrupted transactions with a single smart contract.
Encouraging exploration: By restarting the simulation after a fixed number of transactions, we encourage Predator to explore a diverse set of starting conditions and initial states. This helps to uncover vulnerabilities that may only be reachable from specific entry points or contract configurations.
Focusing on meaningful interactions: With a transaction limit in place, Predator is incentivized to prioritize the most promising and impactful interactions within each episode. This helps to ensure that our testing efforts are focused on the areas of the contract that are most likely to yield valuable insights and potential vulnerabilities.

Figures 6,7, and 8 show snapshots from the Predator dashboard of revert messages the agent received on deployed instances of the LegionFixedPriceSale, LegionPreLiquidSale, and LegionSealedBidAuction. These messages are helpful in understanding what failure modes Predator is encountering and how frequently to understand why the agent may be getting stuck in some instances.

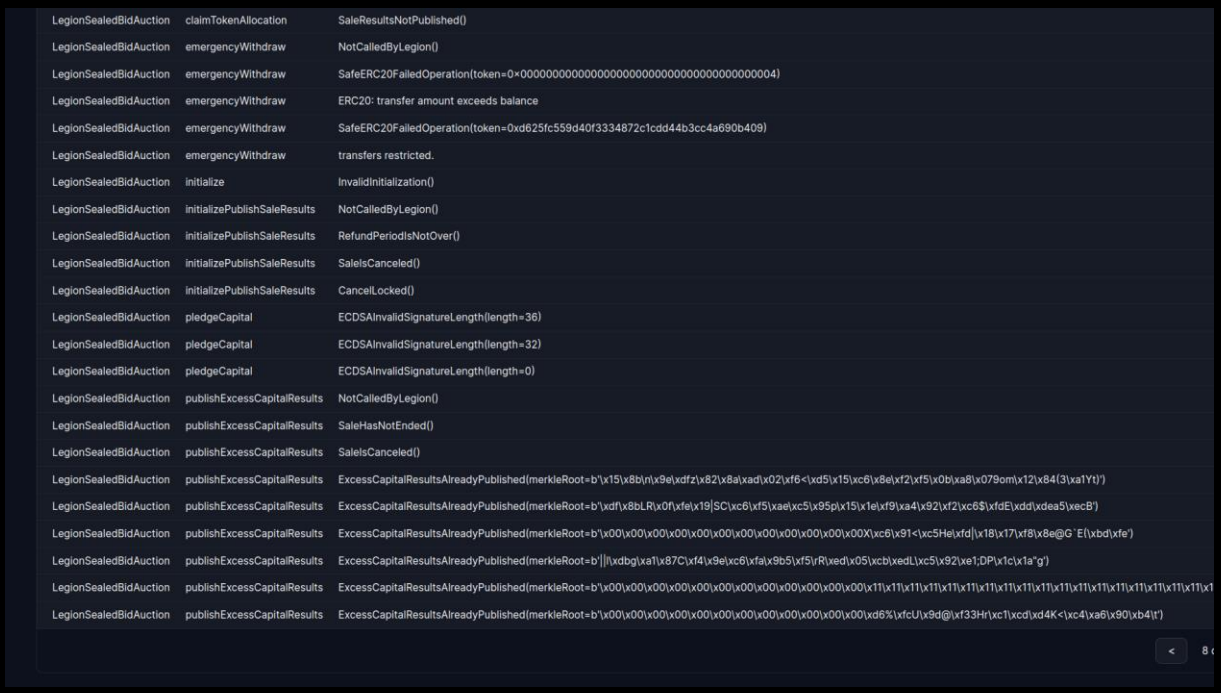
LegionFixedPriceSale	publishExcessCapitalResults	ExcessCapitalResultsAlreadyPublished(merkleRoot=b'BB"\xbb\x05\n\x1f\x7f\x14\x01r2\xa5g\x1f&\x80\xa2\xd3B\x0fPK\xd5e\xcf\x03\x03\\S\x19\x8a')	1
LegionFixedPriceSale	publishSaleResults	NotCalledByLegion()	13794
LegionFixedPriceSale	publishSaleResults	RefundPeriodIsNotOver()	1440
LegionFixedPriceSale	publishSaleResults	17	364
LegionFixedPriceSale	publishSaleResults	SaleIsCanceled()	300
LegionFixedPriceSale	publishSaleResults	TokensAlreadyAllocated(totalTokensAllocated=5)	2
LegionFixedPriceSale	publishSaleResults	TokensAlreadyAllocated(totalTokensAllocated=1)	1
LegionFixedPriceSale	publishSaleResults	TokensAlreadyAllocated(totalTokensAllocated=1000)	1
LegionFixedPriceSale	publishSaleResults	TokensAlreadyAllocated(totalTokensAllocated=100000000)	1
LegionFixedPriceSale	releaseTokens	ZeroAddressProvided()	13455
LegionFixedPriceSale	requestRefund	RefundPeriodIsOver()	8291
LegionFixedPriceSale	requestRefund	SaleHasNotEnded()	3039
LegionFixedPriceSale	requestRefund	InvalidRefundAmount()	1351
LegionFixedPriceSale	requestRefund	SaleIsCanceled()	13
LegionFixedPriceSale	supplyTokens	NotCalledByProject()	11261
LegionFixedPriceSale	supplyTokens	TokensNotAllocated()	3920
LegionFixedPriceSale	supplyTokens	InvalidTokenAmountSupplied(amount=1000)	2
LegionFixedPriceSale	supplyTokens	InvalidTokenAmountSupplied(amount=10)	2
LegionFixedPriceSale	supplyTokens	InvalidTokenAmountSupplied(amount=1000000000000000)	1
LegionFixedPriceSale	syncLegionAddresses	NotCalledByLegion()	9842
LegionFixedPriceSale	withdrawCapital	NotCalledByProject()	10531
LegionFixedPriceSale	withdrawCapital	SaleResultsNotPublished()	2207
LegionFixedPriceSale	withdrawCapital	RefundPeriodIsNotOver()	1106
LegionFixedPriceSale	withdrawCapital	SaleIsCanceled()	217

Figure 6: The Predator dashboard shows us a detailed view of which functions reverted, why they reverted, and how many times it has happened over the campaign on the LegionFixedPriceSale.

LegionPreLiquidSale	updateVestingTerms	NotCalledByProject()	14297
LegionPreLiquidSale	updateVestingTerms	SaleIsCanceled()	32
LegionPreLiquidSale	updateVestingTerms	TokensAlreadyAllocated()	2
LegionPreLiquidSale	withdrawCapitalIfSaleIsCanceled	SaleIsNotCanceled()	14651
LegionPreLiquidSale	withdrawCapitalIfSaleIsCanceled	InvalidClaimAmount()	119
LegionPreLiquidSale	withdrawExcessCapital	17	10617
LegionPreLiquidSale	withdrawExcessCapital	InvalidPositionAmount(investor=0×65b96c229e1a3529e04e9fc1360b9a901cd91e6d)	398
LegionPreLiquidSale	withdrawExcessCapital	InvalidPositionAmount(investor=0xd625fc559d40f3334872c1cdd44b3cc4a690b409)	371
LegionPreLiquidSale	withdrawExcessCapital	InvalidPositionAmount(investor=0×1111111111111111111111111111111111111111)	367
LegionPreLiquidSale	withdrawExcessCapital	InvalidPositionAmount(investor=0×0000000000000000000000000000000000000004)	354
LegionPreLiquidSale	withdrawExcessCapital	SaleIsCanceled()	109
LegionPreLiquidSale	withdrawExcessCapital	InvalidProof(investor=0×65b96c229e1a3529e04e9fc1360b9a901cd91e6d)	28
LegionPreLiquidSale	withdrawExcessCapital	InvalidProof(investor=0×1111111111111111111111111111111111111111)	28
LegionPreLiquidSale	withdrawExcessCapital	InvalidProof(investor=0×0000000000000000000000000000000000000004)	25
LegionPreLiquidSale	withdrawExcessCapital	InvalidProof(investor=0xd625fc559d40f3334872c1cdd44b3cc4a690b409)	19
LegionPreLiquidSale	withdrawRaisedCapital	NotCalledByProject()	12828
LegionPreLiquidSale	withdrawRaisedCapital	CapitalNotInvested(investor=0×58c6913cc54865fd7c1817f88e4047604528bdfe)	465
LegionPreLiquidSale	withdrawRaisedCapital	CapitalNotInvested(investor=0xd11406c415436644678da07cde450a03bd7410e2)	452
LegionPreLiquidSale	withdrawRaisedCapital	CapitalNotInvested(investor=0×20d2a6d53ee43b6758ab16d3f4949a5110876b70)	440
LegionPreLiquidSale	withdrawRaisedCapital	CapitalNotInvested(investor=0×0000000000000000000000000000000000000004)	435
LegionPreLiquidSale	withdrawRaisedCapital	CapitalNotInvested(investor=0×1111111111111111111111111111111111111111)	431
LegionPreLiquidSale	withdrawRaisedCapital	CapitalNotInvested(investor=0xb467c59be9f594abde76897b13e49db8d64b6a14)	429
LegionPreLiquidSale	withdrawRaisedCapital	CapitalNotInvested(investor=0×807116c3ea6a291e3af711b8dcc4fffab2ac7712)	427

< 6 of 10 >

Figure 7: The Predator dashboard shows us a detailed view of which functions reverted, why they reverted, and how many times it has happened over the campaign on the LegionPreLiquidSale.



LegionSealedBidAuction	claimTokenAllocation	SaleResultsNotPublished()
LegionSealedBidAuction	emergencyWithdraw	NotCalledByLegion()
LegionSealedBidAuction	emergencyWithdraw	SafeERC20FailedOperation(token=0×0000000000000000000000000000000000000004)
LegionSealedBidAuction	emergencyWithdraw	ERC20: transfer amount exceeds balance
LegionSealedBidAuction	emergencyWithdraw	SafeERC20FailedOperation(token=0xd625fc559d40f3334872c1cdd44b3cc4a690b409)
LegionSealedBidAuction	emergencyWithdraw	transfers restricted.
LegionSealedBidAuction	initialize	InvalidInitialization()
LegionSealedBidAuction	initializePublishSaleResults	NotCalledByLegion()
LegionSealedBidAuction	initializePublishSaleResults	RefundPeriodIsNotOver()
LegionSealedBidAuction	initializePublishSaleResults	SaleIsCanceled()
LegionSealedBidAuction	initializePublishSaleResults	CancelLocked()
LegionSealedBidAuction	pledgeCapital	ECDSAInvalidSignatureLength(length=36)
LegionSealedBidAuction	pledgeCapital	ECDSAInvalidSignatureLength(length=32)
LegionSealedBidAuction	pledgeCapital	ECDSAInvalidSignatureLength(length=0)
LegionSealedBidAuction	publishExcessCapitalResults	NotCalledByLegion()
LegionSealedBidAuction	publishExcessCapitalResults	SaleHasNotEnded()
LegionSealedBidAuction	publishExcessCapitalResults	SaleIsCanceled()
LegionSealedBidAuction	publishExcessCapitalResults	ExcessCapitalResultsAlreadyPublished(merkleRoot=b'\x15\x8b\n\x9e\xdfz\x82\x8a\xad\x02\xf6<\xd5\x15\xc6\x8e\xf2\xf5\x0b\xa8\x079om\x12\x84(3\xa1Yt)')
LegionSealedBidAuction	publishExcessCapitalResults	ExcessCapitalResultsAlreadyPublished(merkleRoot=b'\xdf\x8bLR\x0f\xfe\x19|SC\xc6\xf5\xae\xc5\x95p\x15\x1e\xf9\xa4\x92\xf2\xc6$\xfdE\xdd\xdea5\xecB')
LegionSealedBidAuction	publishExcessCapitalResults	ExcessCapitalResultsAlreadyPublished(merkleRoot=b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00X\xc6\x91<\xc5He\xfd|\x18)\x17\xf8\x8e@G`E(\xbd\xfe')
LegionSealedBidAuction	publishExcessCapitalResults	ExcessCapitalResultsAlreadyPublished(merkleRoot=b'||\xdbg\xa1\x87C\xf4\x9e\xc6\xfa\x9b5\xf5\rR\xed\x05\xcb\xedL\xc5\x92\xe1;DP\x1c\x1a"g')
LegionSealedBidAuction	publishExcessCapitalResults	ExcessCapitalResultsAlreadyPublished(merkleRoot=b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x1
LegionSealedBidAuction	publishExcessCapitalResults	ExcessCapitalResultsAlreadyPublished(merkleRoot=b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xd6%\xfcU\x9d@\xf33Hr\xc1\xcd\xd4K<\xc4\xa6\x90\xb4\t')

Figure 8: The Predator dashboard shows us a detailed view of which functions reverted, why they reverted, and how many times it has happened over the campaign on the LegionSealedBidAuction.

Reward Function

5.3. Reward Function

In the context of breaking invariants, a sudden spike or sustained increase in the reward function value can indicate that Predator has discovered a sequence of actions that brings the contract closer to violating one or more of the specified invariants. This is a strong signal that a potential vulnerability has been identified and warrants further investigation by our security experts.

In addition to the mean reward function, monitoring the maximum reward value achieved during each training run provides a sense of the upper bound on Predator's performance. A high maximum reward value suggests that Predator has identified a particularly promising or impactful vulnerability, even if it may not be consistently achievable across all episodes.

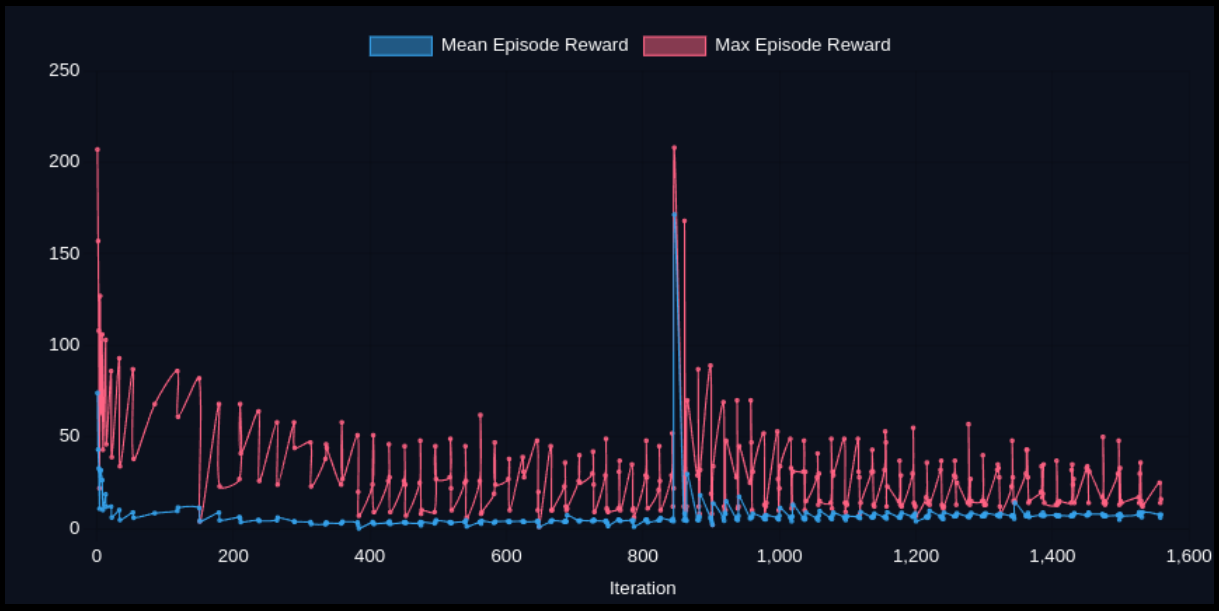


Figure 9: Mean and max episodic rewards for the Legion campaign. Initial rewards are the highest, because Predator is getting lots of reward for quickly exploring novel states.

5.4. Observation Space

The observation space is the system's window into what is going on in the environment. This includes things like the state of the EVM and its callstack, the balances of accounts, and the previous actions in the current iteration. Predator's policy will use the observation space to create a probability distribution that it samples from over the available actions. Predator also leverages Natural Language Processing (NLP) techniques to extract meaning from event and revert messages. Not only do these messages enrich the action space, the encountering of new events and error messages gives Predator a better idea of its code coverage. A more informative observation space means that Predator is better suited to identify potential vulnerabilities and learn to sample actions that can effectively exploit these vulnerabilities.

Storage Key Mutations

5.5. Unique Storage Mutations

The number of unique storage key mutations observed over time is a crucial metric for assessing the effectiveness and efficiency of Predator's search process during the Legion campaign. In the context of smart contract testing, a storage key refers to a specific location in the contract's state storage where data is read from or written to. Each unique combination of storage key values represents a distinct contract state that Predator can explore and potentially exploit.

By tracking the number of unique storage key mutations over the course of the campaign, we gain valuable insights into how well Predator is navigating the contract's state space and discovering novel configurations. A high number of unique mutations indicates that Predator is effectively exploring a diverse range of contract states, increasing the likelihood of uncovering vulnerabilities that may only manifest under specific conditions.

The graph of unique storage key mutations over time typically exhibits a characteristic shape, with an initial steep increase followed by a gradual leveling off. This pattern reflects the natural progression of Predator's search process as it explores the contract's state space.

During the early stages of the campaign, Predator rapidly discovers many novel states as it learns to interact with the contract's core functionality and test various input combinations. This results in a steep increase in the number of unique storage key mutations, as each new state represents a previously unseen configuration of the contract's storage.

However, as the campaign progresses and Predator exhausts the most easily accessible states, the rate of discovering novel configurations begins to slow down. This is reflected in the graph as a gradual leveling off of the unique storage key mutation count. At this stage, Predator must work harder to find new states, often by exploring more complex interaction sequences or targeting specific edge cases.

The asymptotic behavior of the graph is a natural consequence of the finite nature of the contract's state space. While the total number of possible states may be vast, it is ultimately limited by the contract's code and the range of valid input values. As Predator discovers an increasing proportion of these states, the remaining undiscovered configurations become increasingly rare and difficult to reach.

By presenting the unique storage key mutation graph in our report, we aim to provide a clear and intuitive visualization of Predator's search progress throughout Legion campaigns. This metric, along with others such as the reward function and transaction count, helps to paint a comprehensive picture of Predator's effectiveness in exploring the contract's state space and identifying potential vulnerabilities.

Moreover, by comparing the unique storage key mutation graphs across different campaigns or contract versions, we can gain valuable comparative insights into the relative complexity and testability of different smart contract systems. A contract with a more rapidly plateauing graph may indicate a

simpler or more well-defined state space, while a contract with a more prolonged or irregular growth pattern may suggest a more complex or open-ended set of possible configurations.

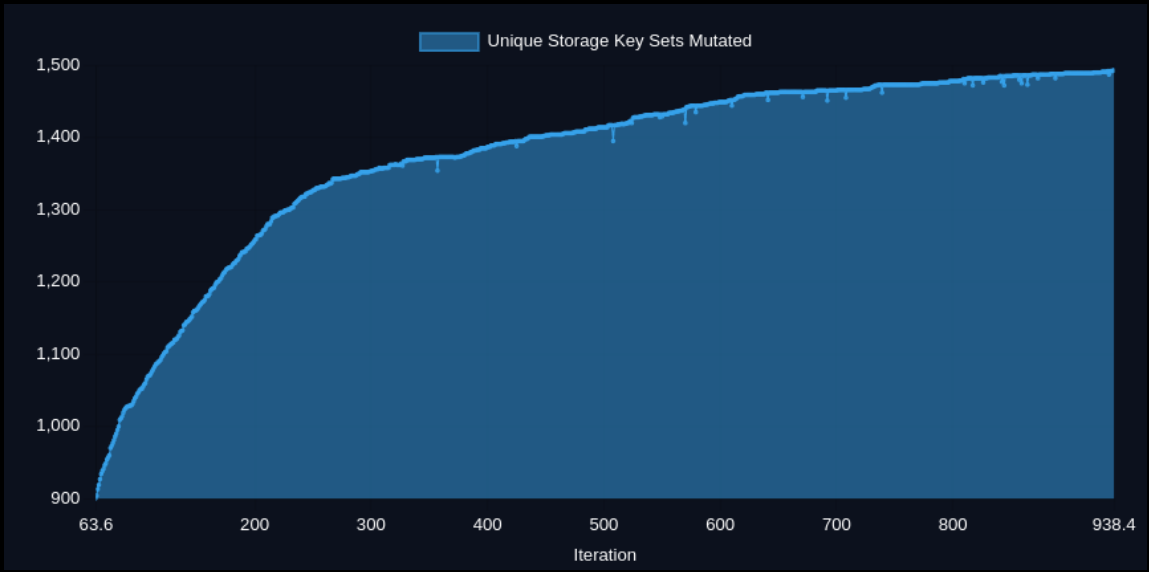


Figure 10: Unique storage key mutations discovered by Predator over the course of the LegionFixedPriceSale campaign. The graph shows a steep initial increase as Predator rapidly explores novel contract states, followed by a gradual leveling off as the remaining undiscovered configurations become increasingly rare and difficult to reach. This visualization provides insight into the effectiveness and efficiency of Predator's search process in navigating the contract's state space and identifying potential vulnerabilities.

Figure 10 highlights a graph of unique storage key sets over a snapshot of campaign iterations. In the early stages, there is a large spike in unique storage keys found as Predator quickly explores novel states. Later, the graph levels off as Predator takes a closer look at states it has already explored, but the graph still trickles upwards as it encounters new states.

In summary, tracking the number of unique storage key mutations over time is a powerful tool for assessing the effectiveness and efficiency of Predator's search process during smart contract testing campaigns.

6. / Future Efforts

Future Efforts

TestMachine is committed to leveraging our expertise and resources to deliver even more comprehensive and effective vulnerability discovery. Our R&D team is hard at work improving Predator to dominate the crypto security ecosystem. We'd like to share with you a few of our goals for the system that are sure to create value for the Legion team and the entire crypto community.

Predator can also be used to attempt to break other invariants. For example, token drain or a "value" drain where after each transaction Predator checks whether it made a simulated profit or drained tokens from a vesting contract. Future custom implementations and tweaks to RBAC invariants can also be covered to examine other specific nuances or ensure some key characteristic of the protocol remains functioning as intended.

We plan on applying this approach to Legion's sale instance contracts to probe the possibility of token/value drain and other critical vulnerabilities.

6.1. Deeper Engagement with Stages of Sale Instances

Legion's protocol and sale instances have several time and condition-dependent stages, (lockup periods, refund periods, vesting periods etc). Predator's ability to advance the simulation allow it to move through and engage the protocol at all these stages, however more work is required to interrogate the entire lifecycle of the protocol, especially in cases that engage with long byte arguments (e.g. the MerkleProof in LegionFixedPrice.sol). Campaigns that run longer Predator should have the ability to submit valid and invalid examples of these types of arguments to better engage and interrogate the protocol. We would like to show that the Legion contracts are robust to state changes, and only expose the intended functionality during each stage.

6.2. Continue Evaluation of Token Custody Risks

TestMachine can evaluate several Bid/Ask token candidates that may be used in Legion's sales. Predator's specific token invariants allow for identification and characterization of the trust levels required out of project admins and bid tokens by providing transparent and testable rules specific to custody risks involved. These campaigns can be performed in isolation of Legion's sale instances and provide significant information on the levels of permissionless that can obtained from integrating specific tokens.

6.3. Custom Value Drain Invariants

TestMachine can use assumed prices for bid/ask tokens to look for potential arbitrage or token drain vulnerabilities. Future work will involve writing an invariant that checks the value agent accounts hold and attempt to extract value out of the protocol maliciously or bypasses a condition check.

7. / Legal

Disclaimer

This intermediate report is not, nor should be considered, an “endorsement” or “disapproval” of any project or team. This report is not, nor should be considered, an indication of the economics or value of any “product" created by any team that contracts TestMachine. This report is based on the scope of materials and documentation provided to TestMachine. Results may not be complete nor inclusive of all vulnerabilities. This report does not provide any warranty or guarantee regarding the absolute product, software or services or that such will be bug-free, without risk or subject to vulnerabilities nature of the technology analyzed. This report should not be used in any way to make decisions around investment or involvement with any project. This report in no way provides financial or investment advice, nor should be leveraged as financial or investment advice of any sort. TestMachine's position is that each company and individual are responsible for their own due diligence and continuous security. TestMachine's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies.