



SMART CONTRACT SECURITY AUDIT

J-memepad

Bonding and Graduation Contracts

CLIENT

J-memepad

LANGUAGE

Solidity

REVIEW

October 2025

DOCUMENT

Document information

Project	J-memepad, Bonding and Graduation Contracts
Client	J-memepad
Platform	Not recorded in the original report
Language	Solidity
Repository	Not recorded in the original report
Commit	Not recorded in the original report
Methods	Not recorded in the original report
Assessed	October 2025

Document history

VERSION	DATE	CHANGE
1.0	October 2025	Original report issued View original report
1.1	October 2026	Reformatted into the Safe Edges standard template. Findings unchanged

This report was reformatted into the Safe Edges standard template in October 2026. Findings, severities, recommendations and code references are unchanged from the original report, which is preserved unchanged in the [Safe Edges Audit Portfolio on GitHub](#).

CONTENTS

Contents

01	Executive summary	3
02	Scope	4
03	Methodology	5
04	Severity and status	6
05	Findings summary	7
06	Detailed findings	8
	C-01 UniswapV3 pool initialization front-run can potentially cause and user loss	8
	H-01 Incomplete Blacklist Enforcement in Bond.sol	9
	M-01 Incorrect use of deadline parameter in GraduationLib.executeGraduation	10
	L-01 IpReserves Not Updated in LPTokenVault.emergencyWithdraw	12
	L-02 Missing Functionalities for Key Royalty Functions in Royalty.sol	12
	I-01 Weak Reserve Token Validation in _checkMethodExists	13
	I-02 Missing address validation	13
	I-03 Missing event emission for important state change	14
	G-01 Mark public functions as external to reduce gas costs	14
	G-02 Remove unused error to save some gas	14
	Disclaimer	16

Executive summary

About J-memepad

J-memepad is a token launchpad. Tokens are minted and burned along a bonding curve, and when a token reaches its maximum supply it graduates, with its liquidity deployed into a Uniswap V3 pool.

Summary of the review

Safe Edges reviewed J-memepad's bonding, graduation, LP vault and royalty contracts in October 2025. Ten issues were identified: one Critical, one High, one Medium, two Low and five Informational, two of which are gas optimisations. The Critical issue let an attacker front-run graduation and initialise the Uniswap V3 pool at a manipulated price. The original marks eight findings Fixed and records no status for two Informational items.

Findings by severity



Findings by status

STATUS	ISSUES
● Resolved	8
● Not recorded	2

Scope

Repository	Not recorded in the original report
Commit	Not recorded in the original report
Platform	Not recorded in the original report
Language	Solidity

Only the code at the commit above was reviewed. Changes made after this commit are not covered by this report.

Methodology

Methods recorded for this engagement: Not recorded in the original report.

Manual review

Auditors read the in-scope code line by line, tracing control flow, state changes and trust boundaries against the intended behaviour described by the client.

Threat modelling

We map privileged roles, external dependencies and value flows, then test the assumptions the code makes about each of them.

Functional and automated testing

Where tooling exists for the language and chain, we run static analysis and targeted tests to confirm suspected issues and check edge cases.

Fix review

After the client addresses the findings, we review the updated code and record the final status of each issue.

Severity and status

Severity levels

SEVERITY	DEFINITION
Critical	Direct loss or permanent lock of funds, or full compromise of the protocol, with few or no preconditions. Must be fixed before deployment.
High	Significant loss of funds, broken core functionality or manipulation of critical state under realistic conditions. Should be fixed before deployment.
Medium	Limited loss or degraded functionality, or issues that need specific conditions or privileged access to exploit. Should be fixed.
Low	Minor impact, low likelihood, or deviations from best practice. Fix when practical.
Informational	Code quality, documentation, gas and style observations with no direct security impact.

Status definitions

STATUS	DEFINITION
Resolved	The issue was fixed and the fix was verified.
Partially resolved	The risk was reduced but not fully removed.
Acknowledged	The client accepted the issue without changing the code.
Open	The issue was not addressed at the time of the report.
Not recorded	The original report did not record a resolution status for this issue.

Findings summary

ID	TITLE	SEVERITY	STATUS
C-01	UniswapV3 pool initialization front-run can potentially cause and user loss	Critical	● Resolved
H-01	Incomplete Blacklist Enforcement in Bond.sol	High	● Resolved
M-01	Incorrect use of deadline parameter in GraduationLib.executeGraduation	Medium	● Resolved
L-01	IpReserves Not Updated in LPTokenVault.emergencyWithdraw	Low	● Resolved
L-02	Missing Functionalities for Key Royalty Functions in Royalty.sol	Low	● Resolved
I-01	Weak Reserve Token Validation in _checkMethodExists	Informational	● Resolved
I-02	Missing address validation	Informational	● Not recorded
I-03	Missing event emission for important state change	Informational	● Resolved
G-01	Mark public functions as external to reduce gas costs	Informational	● Not recorded
G-02	Remove unused error to save some gas	Informational	● Resolved

Detailed findings

C-01 UniswapV3 pool initialization front-run can potentially cause and user loss

Severity **Critical** Status ● Resolved

DESCRIPTION

GraduationLib.executeGraduation calls createAndInitializePoolIfNecessary on Uniswaps NonfungiblePositionManager. That helper will create and, if the pool is new, initialize it to the sqrtPriceX96 supplied by the caller.

An attacker can front-run executeGraduation and create/initialize the pool first with an arbitrary, malicious initial price (e.g., extremely high or low).

Because executeGraduation uses caller-supplied token amounts to derive the initialization price and does not check the pool price against a trusted source, the protocol may mint liquidity at a manipulated price causing the callers intended asset allocation to be mis-specified and funds to be lost or misallocated.

```
function executeGraduation(GraduationParams memory params, address positionManager)
    internal
    returns (bytes memory graduationData, uint256 tokenId)
{
    //existing code
    // Create and initialize pool if it doesn't exist
    @>> address pool =
    INonfungiblePositionManager(positionManager).createAndInitializePoolIfNecessary(
        token0, token1, DEFAULT_FEE, encodePriceSqrt(amount1, amount0)
    );
}
```

ATTACK PATH

1. Alice triggers the graduation flow: (totalSupply >= maxSupply) calls to graduate LP tokens (invoke GraduationLib.executeGraduation).
2. executeGraduation will derive an initialization price and call createAndInitializePoolIfNecessary(tokenLP, reserveToken, DEFAULT_FEE, correctPrice).
3. An attacker monitors the mempool and immediately submits a competing transaction that calls createAndInitializePoolIfNecessary(tokenLP, reserveToken, DEFAULT_FEE, attackerPrice) (for example: 1 LPToken = 1000 USDC).
4. The attackers transaction is mined first and initializes the pool at the attacker-chosen price.
5. Alices original transaction executes afterward.
6. Because the pool already exists, the initialization step is skipped and Alices mint uses the attacker-initialized pool state.

7. As a result Alices deposit is placed into a badly mispriced pool (they spends many more (reserveToken) than expected for the intended LP allocation).
8. The attacker immediately arbitrages the mispriced pool extracting value, leaving Alice with a net loss.

IMPACT

Attacker can front-run and initialize the pool to their advantage and extract value. Leading to user's funds loss due to incorrect pool price.

RECOMMENDED MITIGATION

1. Atomic Initialization: Instead of initializing the pool at graduation time (where it can be front-run), initialize the Uniswap V3 pool atomically during token creation with a fair reference price. The initial price can be derived from the bonding curve parameters (e.g., using steps.price and the maxSupply). While this may not be exact, it ensures a defensible starting point rather than leaving initialization open to arbitrary front-runners.

2. Slippage protection: Always set non-zero amount0Min / amount1Min parameters when calling Uniswaps mint. Even with early initialization, keep slippage protection in place at graduation to defend against sudden pool price manipulations. Optionally, validate pool price against a TWAP or trusted oracle before minting liquidity.

H-01 Incomplete Blacklist Enforcement in Bond.sol

Severity High Status ● Resolved

DESCRIPTION

The contract implements a blacklistedUsers mapping and a notBlacklistedUser modifier, but enforcement is inconsistent and incomplete. While some functions (mint, burn, burnWithNative) verify that the caller is not blacklisted but fails to check whether receiver is blacklisted or not, many other user-facing functions omit this check. As a result, blacklisted users can still receive minted tokens or redemption proceeds indirectly through a non-blacklisted intermediary.

For example:

- createToken – allows a blacklisted user to deploy new tokens.
- updateBondCreator – allows a blacklisted user to assign a new creator address.
- adminMint – allows a blacklisted user to mint supply directly if they control a restricted role.

Finally, blacklist enforcement exists only at the Bond contract level; the ERC20 token contracts deployed via clones do not enforce blacklist restrictions on transfer. A blacklisted account can therefore receive tokens via normal ERC20 transfers even if direct interaction with Bond is partially blocked.

```
function updateBondCreator(address token, address creator) external {
    BondData storage bond = tokenBond[token];
    @>> if (bond.creator != _msgSender()) revert Bond__PermissionDenied();
    // null address is not allowed, use dEaD address instead
    if (creator == address(0)) revert Bond__InvalidCreatorAddress();
    bond.creator = creator;
    emit BondCreatorUpdated(token, creator);
}

function mint(address token, uint256 tokensToMint, uint256 maxReserveAmount, address receiver)
    external
    nonReentrant
    whenNotPaused
    whenTokenNotPaused(token)
    @>> notBlacklistedUser
    _notGraduated(token)
    returns (uint256)
{
    // existing code
    @>> ICommonToken(token).mintByBond(receiver, tokensToMint);
}
```

IMPACT

- Blacklisted users can still perform sensitive operations such as token creation or creator reassignment.
- Blacklisted users can indirectly receive tokens or reserve refunds, bypassing intended restrictions

RECOMMENDED MITIGATION

Add blacklist checks for both the caller and the receiver to all externally accessible user functions. Also, if blacklist restrictions are intended to apply at the token level, extend enforcement to the cloned ERC20 token contracts by overriding functions.

M-01 Incorrect use of deadline parameter in GraduationLib.executeGraduation

Severity **Medium** Status ● Resolved

DESCRIPTION

The protocol is using block.timestamp as the deadline argument when interacting with the Uniswap's NonfungiblePositionManager. The deadline parameter is intended to limit the validity of a transaction, preventing it from being executed after a certain time. Functions that modify pool liquidity check this parameter against the current block timestamp to reject expired actions.

Marking it to block.timestamp effectively means there is no deadline. Any pending transaction that gets delayed due to network congestion, miner selection, or front-running is not protected, and can be executed arbitrarily far into the future, when market conditions may have changed such that the action is now largely unprofitable. This potentially allow the protocol to be exploited everytime it interacts with Uniswap V3, putting users funds at risk.

FROM UNISWAP DOCS

Transaction `deadline` is used to set a time after which a transaction can no longer be executed. This limits the "free option" problem, where Ethereum miners can hold signed transactions and execute them based off market movements.

```
INonfungiblePositionManager.MintParams memory mintParams =
INonfungiblePositionManager.MintParams({
    token0: token0,
    token1: token1,
    fee: DEFAULT_FEE,
    tickLower: MIN_TICK,
    tickUpper: MAX_TICK,
    amount0Desired: amount0,
    amount1Desired: amount1,
    amount0Min: 0,
    amount1Min: 0,
    recipient: address(this),
    @>> deadline: block.timestamp
});
```

IMPACT

- Delayed transactions could be exploited by MEV bots to gain an advantage.
- The deadline parameter gives users a false sense of protection as it provides no actual safeguard.

RECOMMENDED MITIGATION

Set a reasonable buffer for the deadline like this:

```
INonfungiblePositionManager.MintParams memory mintParams =
INonfungiblePositionManager.MintParams({
    token0: token0,
    token1: token1,
    fee: DEFAULT_FEE,
    tickLower: MIN_TICK,
    tickUpper: MAX_TICK,
    amount0Desired: amount0,
    amount1Desired: amount1,
    amount0Min: 0,
    amount1Min: 0,
    recipient: address(this),
    - deadline: block.timestamp
    + deadline: block.timestamp + 1 hours
});
```

L-01 lpReserves Not Updated in LPTokenVault.emergencyWithdraw

Severity Low Status Resolved

DESCRIPTION

The emergencyWithdraw function allows restricted callers to withdraw tokens from the vault without updating the lpReserves mapping. As a result, the vaults internal accounting can become inconsistent with its actual token holdings.

If an authorized caller withdraws LP tokens via emergencyWithdraw, a subsequent call to executeGraduation may attempt to open a position using the `lpReserves[token]` value, which now overstates the actual LP tokens available.

This discrepancy could lead to mispriced LP positions, where the position assumes more tokens than are actually held.

```
function emergencyWithdraw(address token, address to, uint256 amount) external restricted
{
  @>> IERC20(token).safeTransfer(to, amount);
}
```

IMPACT

lpReserves could report incorrect vault balances. Additionally, LP positions could be created with incorrect token amounts, potentially mispricing the positions.

RECOMMENDED MITIGATION

Update the lpReserves mapping whenever emergencyWithdraw is called:

```
function emergencyWithdraw(address token, address to, uint256 amount) external restricted
{
  + lpReserves[token] -= amount;
  IERC20(token).safeTransfer(to, amount);
  IERC20(token).safeTransfer(to, amount);
}
```

L-02 Missing Functionalities for Key Royalty Functions in Royalty.sol

Severity Low Status Resolved

DESCRIPTION

The claimRoyalties function is intended to allow a user to claim royalties for a specific token they have earned. However, in the current implementation, the only function that adds royalties is `_addRoyalty`, which exclusively updates the protocolBeneficiary's royalty balance. As a result, for any address other than protocolBeneficiary, userTokenRoyaltyBalance will always be zero, making claimRoyalties effectively unusable for other users.

Similarly, the `burnRoyalties` function allows anyone to burn royalties assigned to the `BURN_ADDRESS`. However, the contract provides no mechanism that ever credits royalties to `BURN_ADDRESS`, meaning this function also cannot be effectively used under normal operation.

In practice, these two functions are essentially no-ops unless the caller is `protocolBeneficiary` (for `claimRoyalties`).

IMPACT

Users other than the `protocolBeneficiary` are unable to accrue or claim royalties,

RECOMMENDED MITIGATION

Implement a mechanism to credit royalties to arbitrary user addresses if the contract is intended to support user-specific royalty accrual. Otherwise, consider removing or restricting `claimRoyalties` and `burnRoyalties` to avoid dead code and prevent confusion.

I-01 Weak Reserve Token Validation in `_checkMethodExists`

Severity Informational Status ● Resolved

DESCRIPTION

The contract attempts to validate reserve tokens during bond creation by checking whether the token contract implements `decimals()`, `name()`, and `symbol()`. This is done using the private `_checkMethodExists` helper, which performs a `staticcall` to the given method and only verifies that: The call did not revert. And the `return data.length` is greater than a minimum threshold.

However, the function does not decode the return data to confirm that it matches the expected type (e.g., `uint8` for `decimals()`, `string` for `name()/symbol()`). A malicious contract can return arbitrary data of sufficient length to bypass these checks without actually being ERC20-compliant.

However, if the protocol only ever uses some whitelisted reserve tokens, then this weak validation (`data.length` only) is harmless.

I-02 Missing address validation

Severity Informational Status ● Not recorded

DESCRIPTION

The contract does not verify that the provided address is non-zero when assigning it to critical state variables.

```
constructor(address protocolBeneficiary_, uint256 creationFee_, address accessManager)
    AccessManaged(accessManager)
{
    + if (protocolBeneficiary == address(0)) revert InvalidAddress();
    protocolBeneficiary = protocolBeneficiary_;
    creationFee = creationFee_;
}
```

I-03 Missing event emission for important state change

Severity **Informational** Status ● Resolved

DESCRIPTION

Several functions in the contract do not emit events. For example, the `Bond._clone` function does not emit any event, and OpenZeppelins Clone implementation also does not emit an event upon clone deployment. Similarly, there are other instances throughout the contract where events should be emitted to improve transparency and traceability of critical actions.

```
function _clone(address implementation, string calldata symbol) private returns (address)
{
    function adminMint(address token, address recipient, uint256 amount) external restricted
    nonReentrant {
    function emergencyWithdraw(address token, address to, uint256 amount) external restricted
    {
```

G-01 Mark public functions as external to reduce gas costs

Severity **Informational** Status ● Not recorded Category Gas

DESCRIPTION

Functions declared as public can be accessed both internally and externally, which incurs slightly higher gas costs when called externally. If a function is not invoked internally, it is more efficient to declare it as external to optimize gas usage.

```
- function withdraw(uint256 wad) public {
+ function withdraw(uint256 wad) external {
- function totalSupply() public view returns (uint256) {
+ function totalSupply() external view returns (uint256) {
- function approve(address guy, uint256 wad) public returns (bool) {
+ function approve(address guy, uint256 wad) external returns (bool) {
- function transfer(address dst, uint256 wad) public returns (bool) {
+ function transfer(address dst, uint256 wad) external returns (bool) {
```

G-02 Remove unused error to save some gas

Severity **Informational** Status ● Resolved Category Gas

DESCRIPTION

Unused error definitions increase contract size and consume unnecessary gas. Consider removing any errors that are never referenced.

```
- error Bond__TokenSymbolAlreadyExists();  
- error Bond__InvalidLPTokenAmount();  
- error LPTokenVault__OnlyBond();
```

Disclaimer

This report is based on the code at the commit listed in the Scope section and on information the client provided during the review. It reflects Safe Edges' assessment at the time of the review and is not a guarantee that the code is free of vulnerabilities.

Changes made after the reviewed commit, deployment configuration, off-chain components and third-party dependencies are outside the scope of this report unless stated otherwise.

Security is ongoing. We recommend continued testing, monitoring and bug bounty coverage alongside any audit.

This report is not investment advice and is not an endorsement of the project, its team or any token.



Safe Edges is a Web3 security firm and official security partner for the Fuel, ENI and XI EcoChain ecosystems. We work across smart contract audits, Rust and WASM reviews, ZK circuit review, AI and MCP security, and penetration testing.

200+

PROJECTS AUDITED

\$4B+

TVL SECURED

0

POST-AUDIT HACKS