The Steps for a Full Smart Contract Audit

Vulnerabilities enabling the above-known attacks, as well as other bugs and security concerns, would be found with the following auditing process, which takes inspiration from [ConsenSys Best Practices](), the [HashEx audit framework](), and public audits to create a more encompassing structure:

0. Ensure the Audit Will be Completed on a Deployed Smart Contract2: Your audit should be performed on a release candidate (RC), or the final Smart Contract stage before public release, as this is what is closest to the end-user product.

1. Provide A Legal Disclaimer: Note that the purpose of the audit is to foster discussion grounded in security principles, rather than to provide any guarantees.
2.

For example: "The information appearing in this audit is for general discussion purposes only and is not intended to provide legal security guarantees to any individual or entity."

3. **Explain Who You Are**: Explain your authority in the space, or why you can be trusted to conduct a rigorous analysis, and then back it up with a strong audit.

4. **Explain Your Audit Process**: Outline the Smart Contract(s) you are auditing and the process you will use, from a security perspective.

5. **Conduct Attack Vulnerability Tests**: Analyze whether any of the relevant attacks documented above could be successfully launched against the contract.

6. **Detail Vulnerabilities Found and Concerns**: In this step, discuss critical medium, and low severity vulnerabilities, along with suggestions for fixes. There may be areas that are not immediately vulnerable, but a potential point of concern – make note of these as well.

7. **Analyze Contract Complexity**: Complexity increases the likelihood of errors, so make note of complex contract logic, non-modularized code, proprietary tools and code, and performance over clarity. None of these is necessarily a red flag but should be avoided wherever possible.

8. **Analyze Failure Preparation**: How would the contract respond in the event of failures, such as a bug or vulnerability? Check that the contract would pause and that money at risk would be managed.

9. **Analyze Code Currency**: Are all libraries and tools updated to their latest versions? Latest tool versions could come with vulnerability patches, so using older versions is an unnecessary and easily preventable risk.

10. **Analysis of Re-used Versus Duplicated Code**: Duplicated code from previously-deployed, security-proven contracts does not require rigorous analysis. However, re-used code that has not been previously audited must be heavily scrutinized and should not be used if a well-tested and previously deployed version is available.

11. **Analyze External Calls**
    1. **Are State Changes After External Calls Avoided**? External calls may manipulate control flow, so be sure to complete all internal calls first.
    2. **Are Untrusted Contracts Marked**? External contracts should be clearly marked to convey that code interactions are potentially unsafe. This includes naming conventions, such as UntrustedSender as opposed to Sender.
    3. **Are errors in external calls handled correctly**? Contract calls will automatically propagate a throw if an exception is encountered, and without handling this possibility (by checking the return value), the contract will fail.

12. **Do external calls favor push over pull**? Make sure external calls are isolated into their own transactions to minimize the consequences of external call failure.
    11. **Initial Balance Analysis**: Does the code make any assumptions that the contract will begin with zero balance? A contract address may receive wei

before the contract is created, so there should not be an initial balance assumption.

12. **Analyze security of on-chain data**: Ensure that the time at which certain on-chain data appears is not crucial to the contract functionality, as this data is public and the wrong order could favor one party over another (such as in a rock-paper-scissors game).

13. Analyze N-party Contracts: Is it OK if participants drop and do not return? This possibility must be taken into account.

14. Solidity Specific

1. **Are Invariants Enforced**? A failed assertion triggers an assert guard. An assert() should be used when dealing with invariants, such as assert(this.balance >= totalSupply);
2. **Is Integer Division Conducted**? Simply, all integer division rounds down to the nearest integer in Solidity. If this is a problem, use a multiplier instead.
3. **What happens if Ether is forcibly sent**? Since ETH can be forcibly sent to an address, take note of any invariant coding that checks the balance of a contract, and how forced ETH behavior may impact the code.
4. **Is tx.origin used**? tx.origin should never be used for authorization, as it contains your address, so another contract can call your contract and be authorized (if tx.origin is used, recommend using msg.sender() instead).
5. **Timestamp dependence**: As discussed in the Known Attacks section, The Ethereum Timestamp is disconnected from a synchronized global clock, and this discrepancy can be taken advantage of by miners, so timestamp dependence should be minimized.

15. Offer Next Steps: Suggest fixes to the vulnerabilities found and steps moving forward. If these were fixed, would the contract be safe for mainnet usage?