Crypto Valley

Publication | 03.03.2022

Smart Contract Secure Development Lifecycle

Published by CVA's Cybersecurity Working Group

www.cryptovalley.swiss

@thecryptovalley

Authors

Thilo Weghorn
Swisscom

Ognjen Maric
Dfinity

Sebastian Banescu
Quantstamp

Petar Tsankov
ETH Zurich

Duncan Townsend
Immunefi

Table of Contents

1. Introduction

Blockchains and similar technologies (e.g., those based on DAG consensus) provide mutually distrusting parties with shared, distributed databases, often called distributed ledger technologies (DLT). Most such systems allow the users to create programs, called smart contracts, which govern how the shared data may be changed. This shared data typically includes balances in a decentralized currency (e.g., Ethereum), balances of issuer-controlled tokens (e.g., Tether), and digital representations of other valuable assets (e.g., ownership titles). Security issues with smart contracts and distributed ledger platforms can thus inflict direct financial damage to the ledgers' users, as they can result in valuable assets being stolen, destroyed, or rendered unusable. As the prices of digital assets held on ledgers grow, so does the incentive for attackers. The attacks are by now estimated to have caused 21 billion USD (at the time of writing) in damages across different public distributed ledgers[1].

While smart contracts are just programs, securing them is exceptionally challenging compared to most other programming tasks, as their execution environment -- distributed ledgers -- can be extremely hostile. Ledgers usually allow the attacker to directly interact with the program code, instead of a restricted interface as common in most other platforms. The DLT platforms themselves often have properties that are complicated and poorly understood. All this calls for extreme caution while developing smart contracts. However, while the programming community around smart contracts is fast-moving and innovative, it is also immature and a proper software development life cycle is rarely followed. In fact, even recommendations for such a lifecycle are missing. This whitepaper aims to fill this gap, by providing a high-level overview of the smart contract secure development lifecycle.

Our aim is to make this paper useful both to technical management people (such as CSOs and project managers), as well as developers of smart contracts. The explanations are high-level, and don't assume any background knowledge about smart contracts, but we still try to keep them actionable. The paper is organized as follows:

[1] SlowMist Hacked: https://hacked.slowmist.io/en/

- Section 2 - A laundry list of questions necessary to understand the distributed ledger where the respective contract will be developed and deployed
- Section 3 - Defining the requirements for the smart contract
- Section 4 - Common implementation pitfalls and best practices
- Section 5 - Quality assurance (QA) for the implementation
- Section 6 - The post-deployment phase, including monitoring, responding to incidents, and upgrading

2. Setting and framework

The term "smart contract" denotes a program executing on a distributed ledger. These execution platforms have different architectures, which give rise to different environments and execution models for smart contracts, though many share the same characteristics. For example, the programming platform at the core of Ethereum, the Ethereum Virtual Machine (EVM), is also used in other well known distributed ledgers, such as Binance Smart Chain (BSC), RSK, Tron, Fantom, and Cardano, as well as enterprise distributed ledger software such as Quorum and Hyperledger Burrow.

Secure development of contract code critically relies on a thorough understanding of the environment and its execution model. We don't aim to review the characteristics of the individual platforms, as they are too numerous to list. Instead, in this section, we propose a framework for evaluating the critical platform characteristics that affect smart contract security, and that are important to understand before starting to design your solution. The framework's structure is inspired by well-known frameworks such as the NIST cybersecurity framework[2] and can be used to:

1. Decide which DLT is the best fit for the problem you want to solve, and
2. Discover the characteristics and limitations of a particular platform that you have already decided on. This might help you to adjust your design requirements appropriately for the platform to work around any relevant limitations

[2] Nist Cybersecurity Framework: https://www.nist.gov/cyberframework

Category	Sub-category	Questions
Data Model	Basic Characteristics	What kind of data can the platform store, and how is it structured?
		Is data mutable or immutable?
	Data confidentiality and privacy	What mechanisms (if any) does the platform offer to protect the confidentiality of contract data? E.g., who sees the data by default?
		How well do these mechanisms align to your business use case?
		Do the confidentiality mechanisms limit the interaction between the different contracts?
	Transparency of data & execution	Are contract execution and data guaranteed to be visible to all parties who need to see this data?
		Is there a trade-off between transparency and confidentiality?
Execution Model	Cost model	Do you need to pay for executing contract code?
		How is the execution cost determined, and who bears it?
		What mechanisms do you have for controlling the costs? For example, if end-users can interact with your program, does the platform always ensure that they take the costs? If not, can you pass the cost onto them?
		What happens if an execution requires more funds than available? In particular, can your contract get stuck in unexpected ways?
	Execution bounds	Are there bounds on the resources (computation, memory, etc) that your contract may consume during execution?
		Can the bounds be changed, and how?
	Contract interactions	Can different smart contracts interact at all?
		How can you access other smart contracts' data?
		How can you call other contracts' code and how can other contracts call your code?
		Can you know or limit in advance which contracts your contracts may end up calling?
		Can your contract end up calling other contracts that are under attacker control?
		How do the interactions affect the cost model? In particular, who pays for the execution in a setting where multiple contracts end up calling each other?
		What kind of transactionality guarantees does the platform offer? In particular, are atomic changes to the data of the different contracts possible?
	Error handling	How are errors signalled?
		How do errors affect the interaction of the different contracts?
		How do errors interact with the cost model?
	Time model	Does the platform provide a built-in notion of time? If not, what are the sources of time you can use?
		Who can influence the provided notions of time?
		Is it guaranteed that the provided time is monotonic, i.e., that it will never "go back"?
		Are the time guarantees different for workflows accessing a single contract and those accessing multiple contracts?
	Trust assumptions	What are the trust assumptions of the platform?
		Which properties suffer if the trust assumptions are violated?
	Concurrency and consistency	Can contracts be executed in parallel?
		Is the platform sharded?
		If the platform is sharded, can contracts from different shards interact?
		If the platform is sharded, can you control which contracts from which other shards your contract interacts with?
		If the platform is sharded, do consistency guarantees differ when accessing a single shard vs different shards?
Platform Interaction	Identity management	How are the identities defined on the platform?
		How are identities managed?
		If you need to map platform identities to real-world identities, what support/integrations does the platform provide?
	Authentication and authorization	How do the end-users identify themselves on the platform?
		Is the smart contract fully responsible for handling authorization, or does the platform provide authorization mechanisms or policies?
	Platform access permissions	Is the platform fully public, or is the access restricted to a known set of parties?
		Who may trigger the execution of your code in particular, and can you trust these parties?
		How much do the platform participants trust each other?
	Execution requests	How is contract execution triggered by external parties?
		If there are concurrent execution requests, how does the platform decide to order them? Does the platform provide fairness guarantees?
		Can an attacker overtake a legitimate request?
		Do failed requests remain secret?
	Performance	What throughput does the platform offer?
		What is the latency of contract execution on the platform?
		Can heavy load or denial-of-service attacks affect the performance?
	Integration with external systems	If an execution of a smart contract call succeeds, can it still be rolled back in some way (e.g., by a competing chain in a blockchain that eventually turns out to be longer)?
		Are rejections of a command definite, or can they be made definite?
		Does the platform guarantee replay protection?
		Which (if any) de-duplication guarantees does the platform offer?
	Evidence and non-repudiation	Does the platform provide any kind of cryptographic or other evidence of contract execution?
		Does the provided evidence suffice for your needs? In particular, in terms of non-repudiation, or an audit log?
		How can the evidence (if any) be verified by a third party?

3. Design Guidelines

3.1 Requirement engineering

According to a survey of scientific literature[3] we can segment the requirements or properties of smart contracts into four categories: *Security*, i.e. the absence of vulnerabilities, *Privacy*, i.e. respect for privacy of the participants in a contract, *Finance*, i.e. reasonable resource consumption, and *Fairness in Social Games*, i.e. conformance to business-level rules and the fairness to users. The survey elaborates these properties as follows:

Security. Detection of security flaws in smart contracts is one of the main concerns nowadays and, therefore, following security properties are currently studied by the research community:

- Liquidity: A non-zero contract balance is always eventually transferred to some participants.
- Atomicity: If one part of the transaction fails, then the entire transaction fails and the state is left unchanged.
- Single-entrance: The contract cannot perform any more calls once it has been reentered.
- Independence: The execution of the contract is independent of the mutable state and miner-controlled parameters.

Security guarantees of a smart contract also include proper ***access control*** for safety-critical operations (see below), ***arithmetic correctness***, and ***reasonable resource consumption***. Since smart contracts are very similar to concurrent programs, well-established properties like ***linearizability*** and ***serializability*** of its executions must hold.

Privacy. Smart contracts that implement auctions and lotteries select a winner among the users who submitted their bids. Since transactions are publicly visible on distributed ledgers, malicious actors can decide on their moves based on the actions of other contract participants. For example, in a name registration smart contract, attackers can intercept the names chosen by other users from an unconfirmed transaction and register their addresses under their names first.

[3] A Survey of Smart Contract Formal Specification and Verification: https://arxiv.org/abs/2008.02712

Furthermore, in privacy-sensitive domains, such as healthcare and voting, the transparency of the transactions and ledger data may discourage distributed ledger, and thus the usage of smart contracts. Potential solutions for privacy issues in smart contracts include trusted execution environments (TEE) and zero-knowledge proof protocols (ZKP). Two examples that prevent data leaks by using privacy annotations for variables, which specify who is allowed to read the variables' values, are the smart contract language *zkay* that supports variable annotations, and the *Raziel* framework.

Finance. One of the key applications for smart contracts is to process digital assets, specifically those known as cryptocurrency. Smart contracts automate the execution of financial processes that store and transfer funds between users and contracts. Common examples are cryptocurrency wallets, banking services, escrows, cryptocurrency exchanges, and crowdfunding campaigns.
A standard method to ensure the correctness of financial operations in a smart contract is verifying invariants over the contract's balance and aggregations of involved user balances. A common invariant ensures that the sum of all involved user balances equals the token supply that aggregates the total number of tokens. Another common invariant ensures the consistency between sums of sender and receiver balances before and after the smart contract's execution.

Fairness in Social Games. We consider *social games* as a category of smart contracts that describe how their users, i.e. players, participate and interact according to some predefined rules. From this a fairness notion can be derived that these smart contracts intend to provide. Given utility functions of the participants, game-theoretical techniques, e.g. provided in the theory of mechanism design are used to verify if the intended fairness notions of such a smart contract hold or if there are possible loopholes.
Examples of such social games include auctions, voting schemes, lotteries, classical games like rock-paper-scissors, and Ponzi-like games. The payout to each player of the game is automatically determined by the smart contract and, therefore, fairness is the main property to be verified. However, providing fairness properties can lead to privacy issues, e.g., when participants must be able to track submissions of other players on the underlying distributed ledger.

3.2. Access control model

Access control defines *what each user is allowed to do* in a given system. In the world of smart contracts, transparency of the whole system state is often an essential feature and distributed ledgers are generally open platforms where anyone can submit transactions. Therefore, access control of your contracts may govern who can mint tokens, vote on proposals, freeze transfers, and other sensitive actions. It is therefore critical to understand how to implement proper access control mechanisms, since many exploits of smart contracts are based on design flaws in their access control. One example of such an exploit is the Parity wallet hack which allowed anyone to kill the contract and freeze the funds[4].

The first crucial design decisions in defining the access control of a smart contract must include:

1. Define privileged users: can be user accounts or other contracts
2. Define sensitive actions: ether transfers, token transfer, changing ownership, etc.
3. Decide for an ***access control model*** that determines the relationship between which user can execute which sensitive action.

The most commonly used access control models are the following three:

1. The most common form for access control of smart contracts is ***ownership***. By default any party can call a contract method, but it must be ensured that sensitive contract methods can only be executed by the *owner* of a contract. The owner of a smart contract is a user account who can execute administrative tasks on the contract, which are sensitive and crucial for the correct operation of the contract. Usually, this makes mainly sense if a contract provides just a single administrative user. The ***ownership pattern*** is an authorization pattern used in such cases. In this pattern the contract creator's address is stored as owner of this contract and restrict method execution dependent on the caller's address.

2. However, if more different levels of authorization are needed, e.g. an account who is able to ban users, but is not able to create new tokens, then ***Role-Based Access Control (RBAC)***

[4] Parity Wallet Security Alert — Vulnerability in the Parity Wallet library contract: https://medium.com/crypt-bytes-tech/parity-wallet-security-alert-vulnerability-in-the-parity-wallet-service-contract-1506486c4160

provides more flexibility. In this access control framework multiple *roles* can be defined that are able to perform different sets of actions. Furthermore, rules can be defined that assign these roles to users, transfer and delegate them, etc. Best practices often suggest a role for the majority of regular users that consume the service of a contract, a role for supervisors or managers, and a role for few users that have administrative privileges.

3.Attribute-based access control (ABAC) is a model which evolved from RBAC to include attributes of users, resources, actions and context. Examples for such attributes are:

- the user e.g. citizenship, age,
- the resource e.g. classification, department, owner,
- the action, e.g. criticality, affected parties, and
- the context e.g. time, location, IP.

By default a contract method is executed without any preconditions being checked, but it is desired that the execution is only allowed if certain requirements are met. Since there is no built in mechanism to control execution privileges, a common pattern is to restrict function execution, called the *access restriction pattern*. This pattern first defines generally applicable modifiers that check the desired requirements and then apply these modifiers in the critical function's definition. These can refer to different categories, such as temporal conditions, caller and transaction info, or other requirements that need to be checked prior to a function execution.

The advantage of ABAC compared to RBAC is that it is policy-based, i.e. in ABAC we can provide policies that depend on attribute values rather than static permissions to define what is allowed or what is not allowed. This can be very nicely implemented in smart contracts by checking various attributes of an account/address. For example, if a user has at least 1000 ERC20 tokens of some kind they can use the premium features of some smart contract or web service.

Nowadays, RBAC is mainly used for smart contracts due to its simplicity. However role hierarchies are hard to implement correctly since they represent a tree structure that can lead to unbounded gas usage.

3.3. Threat Modelling

Threat modeling is the process to identify potential threats, such as vulnerabilities or the absence of safeguards, to a valuable asset. Valuable assets in a smart contract ecosystem are commonly sensitive functions and data that need to be protected. After identification of these threats the next steps include prioritizing the threats and providing defenses to mitigate them.

Attack Tree: This is a common concept using decision trees with its root node providing the attackers' goal and its paths from the leaf nodes providing the different steps to achieve this goal. An attack tree can include special knowledge, equipment, time, or operational and development resources needed to execute each step of the attack.

Threat Actor: The threat actors are usually considered to be users or groups of users that perform malicious acts to break one of the properties defined above. As already mentioned, public smart contracts are accessible by almost anyone and therefore the number of threat actors is higher than for standard software products.
The most powerful threat actors are, firstly, *nation state actors* with almost unlimited resources and, secondly, *organized crime actors* aiming for financial gains. *Insider actors* have extensive knowledge of the system with personal goals like revenge, and *hacktivist actors* are ideologically driven. Finally, *script kiddies* at the lower end have the most limited knowledge and very few resources and often aim for recognition.
Furthermore, threat actors are often categorized as either unintentional or intentional and either external or internal. The likelihood and impact of a threat is closely related to the attacker's motive. For example, a nation state actor has a high level of motivation and corresponding resources in contrast to script kiddies which provide the lower end of motivation and therefore likelihood. In the context of smart contracts, threat actors can be further divided into two types, malicious smart contracts and humans interacting with smart contracts.

4. Implementation

There is nothing inherently different between regular secure software development and smart contract development. Frequently, however, the stakes are higher due to the open, permissionless nature of most distributed ledgers and due to the unique challenges of DLT programming. It is beyond the scope of this section to discuss specific pitfalls, but we endeavor to describe best practices and common misconceptions.

4.1. Common pitfalls

DLTs, and in particular EVM-based chains, provide many useful features that *appear* to make development easier. However, caution must be exercised when integrating with these features in order to avoid introducing vulnerabilities.

On-chain pricing oracles are a common source of vulnerabilities, as they are frequently subject to manipulation by attacks like flash loans and sandwiches. Correct integration with an on-chain pricing oracle is frequently computationally expensive and difficult to correctly incentivize. Using a time-weighted average price (TWAP) oracle is best practice here.

Block timestamps are an important source of information for smart contracts, but they are also manipulable by a miner, creating opportunities for miner-extractable value. If a deviation in timestamp on the same order as the finality duration would significantly affect the integrity of the application, it is inappropriate to use the block timestamp. Use an off-chain timing oracle instead. Never pass the block timestamp into a hash function as a source of entropy.

Similarly, block hashes should not be used as a source of entropy if miner-extractable value is a possible attack vector. Use an off-chain source of entropy or use a verifiable delay function with a commit/reveal scheme that takes the block hash as its input.

Particularly in EVM-based blockchains, passing function selectors or entire calldata into low-level operators like `call` or `delegatecall` introduces insufficient validation vulnerabilities. Either the source of these selectors/calldata must be validated, such as whitelisting callers, or the

selector/calldata itself must be validated which is frequently error-prone and computationally expensive.

Smart contracts that rely on transaction ordering to avoid races or market manipulation can also be attacked. Until a transaction is included in a block, it is contained in the mempool, where the relative ordering of the transaction has not been fixed yet. There are systems in place to programmatically reorder transactions in the mempool to extract value by miners and by attackers.

There is significant platform risk when integrating with systems that have centralized control by third parties. If the contract you interact with is behind an upgradeable proxy, consider that whoever controls that proxy has complete control over the function of the contract. Sophisticated rug-pull attacks as well as attacks that specifically penalize individual integrators are possible.

DLTs usually allow smart contracts to call other smart contracts, whose code might not be known in advance. If the contracts are mutable, after the control returns to the caller contract, the state of the caller contract and its environment might be arbitrarily different from the state before the call. This makes smart contracts similar to concurrent programs, and common classes of bugs in concurrent programs (most notable, re-entrancy bugs) may also appear in smart contracts.

Finally, arithmetic issues are another common source of bugs. The most common sources include under- and overflow issues, as well as issues with precision when using floating point or decimal arithmetic.

4.2. Reproducible builds

In most applications, such as DeFi, smart contracts are used by different parties that don't fully trust each other, or do not even know each other. Thus, the parties should be able to independently verify which code the contract is running. This verification is complicated by the fact that, on most DLT platforms, smart contracts are written in a high-level language, and then compiled to a lower-level language for execution. For example, Ethereum contracts are usually written in Solidity, and then compiled down to EVM bytecode. It is crucial that all parties are able to independently verify the link between these two levels of code, that is, the low-level code is a faithful translation of the source

code. This is especially true since audits and other forms of quality assurance are generally performed on the source code level.

To enable third parties to verify this link, the authors of the smart contract must ensure that their process of building the low-level code is reproducible by these third-party verifiers. The following guidelines help to achieve that:

1. When producing the final version of your container code, use Docker or a similar technology to conveniently set up the operating system and the build tools, and fix their versions for the user. If the build tools you are using do not guarantee fully reproducible builds, Docker can also help by minimizing the differences in paths, environment variables etc.
2. Your basic infrastructure, such the build tools and the base Docker image should be sourced from a trusted origin.
3. Minimize the use of dependencies to those that are absolutely necessary for your use case, as the dependencies will have to be audited by your users in addition to your own smart contract code. If your build tools support using dependencies from external sources, pin the versions, or even better, the checksums of these dependencies.
4. If you do use external dependencies (i.e., not bundled with your source code), the repositories hosting these dependencies may disappear if they are not under your control. Thus, ensure archiving them as long as you expect your contract to be in use.

4.3. Best practices resources

For common design patterns, it is best to use well-vetted libraries instead of re-implementing the pattern yourself. A well-known collection of open-source contracts for EVM is OpenZeppelin[5] which include contracts to implement access control, e.g. RBAC, and SafeMath for safe arithmetic operations. This is, of course, good practice for all software development, but it is especially important in smart contract development where there are unforeseen pitfalls.

5. Quality assurance

The goal of quality assurance is to systematically inspect the behavior of smart contracts to gain confidence in their correctness and identify defects that must be fixed. To proceed with quality assurance, it is therefore of utmost importance to have well-formulated and sufficiently complete

[5] OpenZeppelin: https://openzeppelin.com/

requirements, as discussed in Section 3. A key challenge in conducting a thorough assessment of the smart contracts is that they are often deployed in an open-ended environment, as discussed in Section 2, allowing potentially any user to craft any input (e.g., Ethereum transaction) and submit it to the contract. That is, the contract must anticipate *arbitrary inputs*, submitted in *arbitrary order*, from *arbitrary users* (including malicious ones). Further, the contracts often interact with other smart contracts on the distributed ledger, whose code may be unknown in advance. Thus, the contract must anticipate arbitrary behaviors of the smart contracts it interacts with and ensure that the elicited requirements are not violated.

In the following, we review four of the most common types of methods used to assess the quality of smart contracts.

5.1. Testing

A test consists of a concrete sequence of inputs that is executed against the smart contracts, along with an assertion that checks whether the contract's behavior is correct. The tests can be written manually, collectively defining a regression test suite that is used to assess the smart contracts upon any changes to their code. Since manually defining a comprehensive set of input sequences that thoroughly exercises the behaviors of the smart contract, techniques that automate the testing process are also popular. We highlight two types of automated testing techniques:

- Fuzzing: Fuzzing is a technique for automatically generating sequences of inputs and running these against the smart contract. To check whether the smart contract's behavior is correct, developers must still provide assertions, e.g. that define relevant state properties, and the fuzzing tool would then generate inputs to check whether the provided assertion holds, or not. A key limitation of fuzzing tools is that it can be challenging to generate inputs that thoroughly exercise the contract's behavior and find violations to assertions that do not hold.
- Symbolic execution: Symbolic execution can be used to explore a much larger number of behaviors by symbolically encoding many behaviors of the contract and checking their correctness using off-the-shelf constraint solvers. For example, they can check the correctness of the contract for certain transaction depths (typically 3-4 transactions) or along specific execution paths of the smart contract.

Regardless of the technique used, it is important to note that testing cannot prove the absence of faults in the smart contract's code. This is because, in practice, it is infeasible to enumerate all possible input sequences and certain bug-exposing inputs can be inevitably omitted in the test process.

5.2. Formal methods

In contrast to testing, formal methods such as program analysis and verification are able to *prove* the correctness of the smart contract's code, achieved by deriving a formal mathematical proof of correctness. Formal method tools are typically designed to reason about all possible smart contract behaviors. However, many times these tools need to make compromises in order to enhance their practicality and scale their performance to deal with realistic smart contracts.

When it comes to using formal method tools, it is important to distinguish *what* is verified, i.e. what is the formal property that is the target of verification, and *how* the target property is verified.

In terms of *what* is verified, we distinguish tools and techniques that are able to verify:

- Pre-defined requirements: These often cover generic security requirements, such as atomicity, single-entrancy, and liquidity, as described in Section 3, which are well-established and are expected to hold for *any* smart contract. This allows security experts to create tools that target the verification of these critical properties.
- Custom requirements: These typically cover properties that pertain to a specific smart contract, such as access control requirements which are often use-case specific. Since contracts are stateful and typically offer different functionality depending on the contract's state, such custom requirements are often temporal properties, defining correctness in terms of sequences of states (for example, refunds are not allowed upon successful crowdsale).

In terms of *how* these formal properties are verified, we distinguish primarily two types:

- Manual verification: Manual verification is typically carried out in a theorem prover where a verification engineer derives a correctness proof assisted by the prover. A key benefit to contracts verified in theorem provers is that one obtains proof that can be mechanically checked by the proof assistant. The downside is that the proofs may require significant expertise to derive relevant inductive invariants to derive the proof.

- Automated program analysis: There are also smart contract verifiers that aim to automatically prove the contract's correctness relative to its specification. Typically, these verifiers are based on different forms of abstract interpretation, which provide automation but sacrifice the ability to prove any requirement that is correctly implemented in the contract. That is, a failure to prove the target property does not necessarily mean that the smart contract violates the requirement. Because of this, spurious issues are often identified, which consumes development time.

Important to note is that verification, in general, does not guarantee that the smart contracts are 100% correct and secure. Any correctness guarantees are relative to the elicited requirements of the smart contracts. In practice, however, there are always implicit, unspecified requirements and critical bugs may slip through the verification process. Therefore, formal verification alone is insufficient to ensure a thorough assessment of the code and one must also consider complementary techniques. For example, penetration testing where a security expert deliberately attempts to attack the contract can nicely complement the verification process.

Finally, we note that while the importance of formal methods is still underappreciated in other software development environments, they are very appealing in the space of smart contracts since software defects in smart contracts can cripple entire crypto-economic systems. Further, as these tools evolve and become better understood by the public, we will be seeing that these are sometimes also employed by hackers (both good and bad) to scout for issues at scale. For example, a well-known insurance provider is starting to make use of these tools in conjunction with audits to lower their claim payouts.

5.3. Audit

The first decision necessary when thinking about getting an audit is the type of audit you would like to get. There are many flavors of audits which can be performed on a smart contract project. Some of the most well known types of audits are:

1. Timeboxed security reviews, which generally have a lower time budget and require that at least one security auditor reviews the project by looking at the source code to find vulnerabilities.
2. Full audits, which are not restricted from a time budget perspective and employ the 4-eyes principle, meaning that at least two security auditors look at the same part of the source code to find vulnerabilities.
3. Formal verification, as explained in the preceding section.

Once you have decided which type of audit to perform, these are the steps that you need to take into consideration:

1. Share the code to be audited with various firms such that they can scope the amount of work required and provide you a quote. Ideally the scoped code has a minimal difference to the audited code. This means that the code is at least 90% done when providing it for scoping. Note that at the moment when you want to perform an audit there might be a high demand for audits and you may need to wait for a long period (e.g. 3 months) before a firm starts auditing your project. After you receive the first audit report, you will also need to go through a period where the issues identified in the audit are fixed and then re-audited.
2. After receiving quotes from multiple auditing firms, decide on which company you want to choose based on the price, timeline and reputation of the firm.
3. Set-up a shared communication channel between your team and the auditors to facilitate information sharing.
4. Provide the auditors with the most succinct, but sufficient documentation such that they have a good understanding of how your project is supposed to work before they start auditing. This is key for obtaining high quality findings, which are for example vulnerabilities caused by deviations from the specifications.
5. Walk the auditors through your code just before they start the audit.
6. After the auditors have finished going through your code they will compile a report and ship the report to you.
7. Go through the findings and respond to all signaled issues i.e. either fix the issues in the code or write a formal response for the issues which you do not wish to address, providing the reason why such issues are not addressed in the code. This step should be performed soon after the audit report is received to minimize the amount of information that auditors will forget about your project and code.
8. After the auditors receive your code fixes and the formal response for issues which were not fixed, they will check the fixes and mark all issues in the report as fixed or acknowledged.
9. Publish the audit report yourself and also ask the audit firm to publish it as well in order to avoid any false claims about the integrity of the audit report document.

6. Deployment

This section discusses the most relevant security aspects that should be taken into consideration after deploying smart contracts onto a DLT.

6.1. Monitoring and Alerting

It is of utmost importance to employ a monitoring solution that can detect attacks on your smart contracts running in production. Often attacks in this space can cause a lot of damage in a short time frame, e.g. in a single transaction or a handful of transactions. Therefore, the monitoring solution you choose should be able to quickly detect and alert the right stakeholders, e.g. the engineers, who could perform a privileged operation to stop the attack.

The majority of smart contracts have different interfaces (i.e. state variable names, function signatures) and different security requirements. Until a push-button anomaly detection solution arrives on the market, the typical rule-based monitoring (in wide-use today) requires customization with specific rules to raise alerts. One example of a rule in a decentralized lending platform would be to trigger an alert if a healthy position would get liquidated. Such rules could be a byproduct of the white-box security audit.

6.2. Incident Response

Monitoring and alerting are worth little without designing a clear plan for incident response. In order to perform effective incident response, there need to be emergency mechanisms and procedures set in place for each possible alert type that the monitoring system can trigger. One typical example of such mechanisms are so-called circuit-breakers, which can disable specific smart contract functions. Such circuit-breakers are often found in systems which are not decentralized and where an admin account can trigger such mechanisms. Additionally, there should be an emergency shutdown mechanism, which can handle scenarios that were not envisioned. Such emergency shutdown mechanisms exist even in highly decentralized smart contracts such as those of Maker DAO[6].

[6] Emergency Shutdown: https://makerdao.world/en/learn/governance/emergency-shutdown/

It is important to note that incidents are not exclusively raised by the monitoring system. For example, a vulnerability could be discovered internally by the engineering team while extending the code. Depending on the impact and the likelihood that the vulnerability could be exploited, the team decides how to respond. Moreover, incidents could also stem from bug bounty programs, where ethical hackers responsibly disclose vulnerabilities found in the system without exploiting them. In such situations, the engineering team typically has approximately 90 days until the ethical hacker publishes their findings publicly.

To respond uniformly requires formalizing a business/decision process that is able to answer the following questions:

1. How many different types of alerts exist and how are they ordered? For example, alerts could be ordered in terms of severity: low, medium, and high.
2. Who needs to be involved/notified for each type of alert? For example:
    a. Low severity incidents must be handled (not resolved) by engineers within 5 working days from when they are received.
    b. Medium severity incidents must be handled (not resolved) by engineers by the end of the next working day from when they are received.
    c. High severity incidents must be handled within one hour on the day when they are received, by the engineer who is on-call.
3. What is the escalation process in case the person responsible does not handle the incident in the required amount of time?
4. Which emergency mechanisms can be used for which incident types and in which situations and in which order? For example, the engineer which is on-call when a high severity incident happens can directly perform an emergency-shutdown of the smart contract (blocking all end-user operations), if the monitoring system indicates a loss greater than 10% of the total value locked in the system prior to the incident.

7. Upgrading Smart Contracts

In the web 2.0 world post-deployment considerations typically include regular and sometimes quite frequent updates to the code. However, when it comes to smart contracts updates are generally way less frequent due to the immutability of smart contracts on some blockchains (e.g. Ethereum). However, even though some DLTs like Hyperledger Fabric or DFINITY's Internet Computer do facilitate updating contract code, such updates are typically done less frequently than web

application updates. Moreover, upgrading contract code may change the contract's behavior in arbitrary ways. In some contexts, such as DeFi, this is usually problematic, and contract upgrades may require a decentralized governance mechanism.

8. Acknowledgments

We would like to thank Travin Keith, Neville Grech, Markus Perdrizat, for the helpful comments in preparing this whitepaper. We also Nikoletta Csanyi for preparing the final version and distributing the whitepaper.

Building the World's Leading

Blockchain Ecosystem

www.cryptovalley.swiss