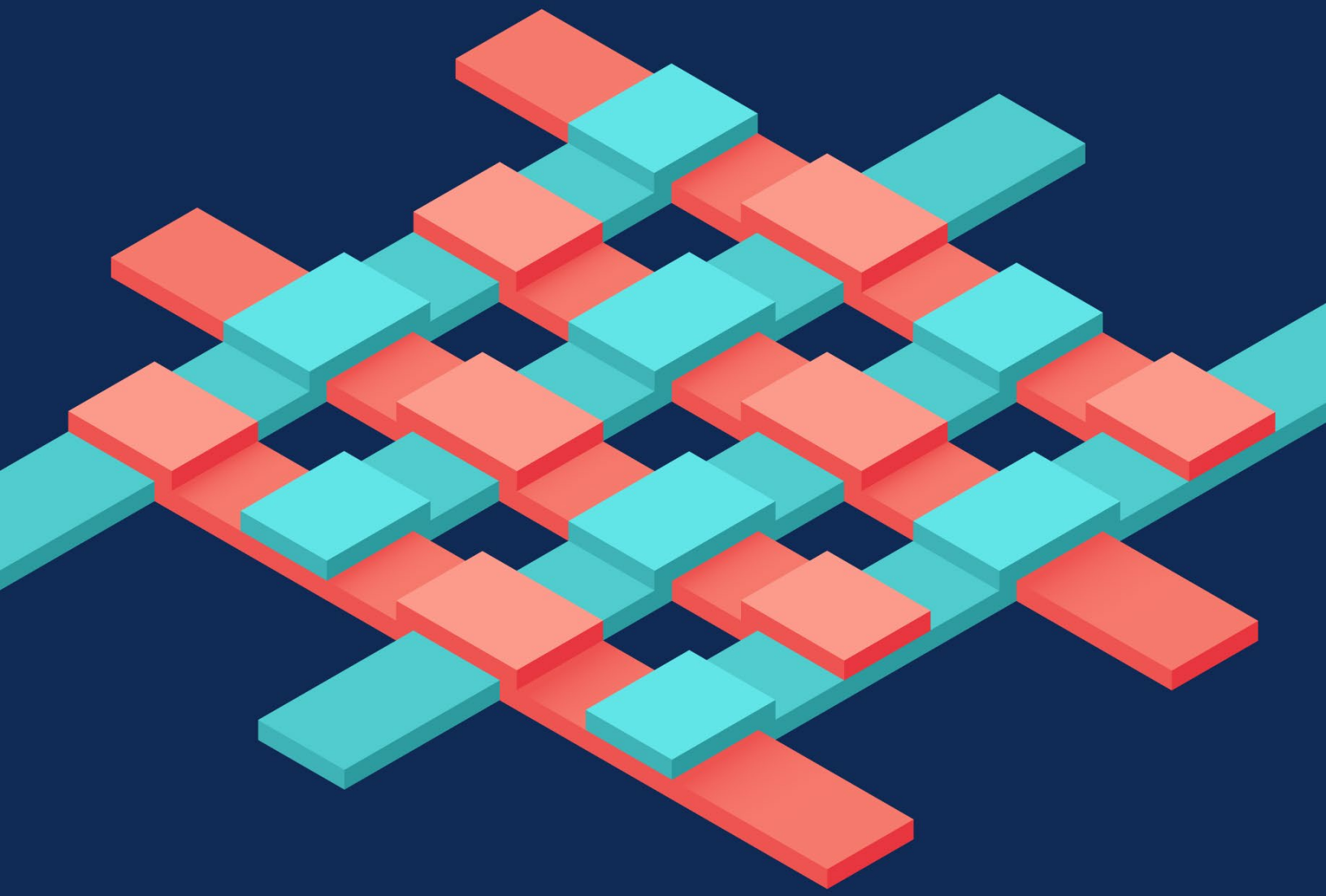
Best Practices for Smart Contract Security

Hyperledger Fabric 2.0



cloud security alliance®

Acknowledgments

Lead Authors

Nnamdi Osuagwgu

Key Contributors

Gustavo Nieves Arreaza
John Carpenter
Jyoti Ponnapalli
Michael Theriault
Kurt Seifried
Raul Documet
Ulf Mattsson

Reviewers

Goni Sarakino
Kurt Seifried

CSA Program Manager, Research

Hillary Baron

Table of Contents

Executive Summary

Intended Audience

This paper is for technologists and stakeholders interested in adopting and deploying smart contracts within their enterprise. These stakeholders may include C-Level executives that seek to learn the corporate benefits of smart contracts, as well as the benefits and challenges for those technologists who are deploying a smart contract solution.

Benefits of the Paper

The reader will gain an understanding of the benefits, challenges, and opportunities for deploying smart contracts within their organization. The reader should also gain an understanding of many of the legal, regulatory, and security considerations that must be taken to count when using any smart contract.

Scope and Assumptions

This paper will only support chaincode programmed in Golang.

1. Smart Contract Overview

Blockchain technology enables everybody involved in a transaction to know with certainty what happened, when it happened, and to confirm other parties are seeing the same thing—without an intermediary providing assurance, and without having to reconcile the data afterwards.

Smart contracts are programs that run "on the blockchain." They allow the creation of completely distributed and decentralized applications and are designed to be self-enforcing. The code of a smart contract is designed to be the final authority on the agreement that it encodes, meaning that any contractually valid interaction is considered "fair use" including exploiting a logical or programming flaw in the contract. Such "code as law" functionality of certain blockchain platforms may present unique legal challenges that will need to be addressed before implementing a smart contract solution in an industry.

Smart contract platforms (sometimes called "world computers") are designed to have blocks with transactions that consist of instructions to execute. Each node in the blockchain network runs a copy of the blockchain's virtual machine and executes the code contained in transactions. This creates a parallelized and sometimes hugely inefficient distributed computer since the parallelization is used to maintain synchronization (each node runs the same code in the same order) rather than speed processing. Some performance efficiency issues may be addressed by implementing a blockchain solution that utilizes a "proof of stake" or "proof of elapsed time" methodology.

Smart contracts are designed to be Turing-complete, meaning that anything that is possible for a program on a computer should be possible in a smart contract.

2. Business Objectives of Smart Contract

2.1 What Are the Goals of the Smart Contract Being Developed?

It's important that we consider the goal or purpose of the smart contract we are considering. The number of higher order objects we must consider will be tied to the use case. The more detailed and robust the smart contract, the more surface area exposed to risk. Additionally, smart contracts, such as those developed through the Accord Project, can combine both computer executable code as well as a natural language legal framework from which to work within. Therefore, any practitioner deploying a natural language smart legal contract should understand the risks associated with the execution of such a hybrid legal agreement.

2.2 Who Are Your End Users?

Why do we want to know who the end users of your smart contract are? Because we want to understand what behaviors we can attribute to a particular demographic, cultural, or sentimental marker. If we know the types of behaviors to expect of our users, we may be able to mitigate those behaviors, or at least baseline them to unlock added value from knowledge of our users. Additionally, we need to understand if we are dealing with a cross-border transaction and the associated treaty, statutory, regulatory and case law that affects the parties involved in each jurisdiction. Also, a basic technical understanding of smart contracts by the end users will drive how smart contracts will be received. If an end user understands the efficiencies gained through the use of a smart contract, as well as the associated legal and technical risks, they will be better suited for adopting a smart contract for their use case.

2.3 Does Your Smart Contract Transact Currency?

If your smart contract transacts currency, there are additional legal considerations to take into account, such as compliance with Know Your Customer (KYC) and Anti-Money Laundering (AML) regulatory requirements. Cross-border transactions involving currency (including cryptocurrencies) must comply with international law, and all specific legal requirements from any governmental entity that may have jurisdiction over the matter. This may include required reporting to tax authorities around any capital gains around the sale or transfer of any cryptocurrency.

2.4 Does Your Smart Contract Transfer Asset Titles?

Asset transfers may be governed by international, national, state/province, and even local jurisdictional laws. One of the primary items to consider when working with blockchain smart contracts is not only whether the transfer of title can be accomplished through a blockchain solution,

such as Hyperledger Fabric, but also whether such a transfer is supported by the relevant statutes, regulations, treaty, or local law. There also needs to be a consideration of the broader legal picture of how asset titles are transferred. In the case of real estate transactions in the United States, if an automated blockchain smart contract were to transfer title without first checking to make sure that the individual was not on active duty with one of the branches of the military, or that the individual had not filed for bankruptcy protection under the United States Bankruptcy Code, that automatic transfer would be in violation of the federal laws of the United States and the parties and counsel would likely be subject to sanction and legal penalties. Additionally, in real estate law, there is the concept of a bundle of rights, and when a title transfer occurs, whether it transfers all of the rights or just one from the bundle, and is that properly documented in the smart contract. If this is not done correctly, this can result in severe legal consequences.

3. Hyperledger Smart Contract Ecosystem

A smart contract is essentially business logic running on a blockchain. Smart contracts can be as simple as a data update, or as complex as executing a contract with conditions attached.

There are two different types of smart contracts. The first one being smart contracts that install business logic on the validators in the network before the network is launched. The second being the on-chain smart contracts deploy business logic as a transaction committed to the blockchain and then called by subsequent transactions. With on-chain smart contracts, the code that defines the business logic becomes part of the ledger.

3.1 Hyperledger Smart Contracts Layer and Other Architectural Layers

As an umbrella project, Hyperledger does not have a single architecture per se. However, all Hyperledger projects follow a design philosophy that includes a modular extensible approach, interoperability, an emphasis on highly secure solutions, a token agnostic approach with no native cryptocurrency, and ease of use. The Hyperledger architecture has distinguished the following business blockchain components:

- **Consensus Layer:** Responsible for generating an agreement on the order and confirming the correctness of the set of transactions that constitute a block.
- **Smart Contract Layer:** Responsible for processing transaction requests and determining if transactions are valid by executing business logic.
- **Communication Layer:** Responsible for peer-to-peer message transport between the nodes that participate in a shared ledger instance.
- **Data Store Abstraction:** Allows different data stores to be used by other modules.
- **Crypto Abstraction:** Allows different crypto algorithms or modules to be swapped out without affecting other modules.
- **Identity Services:** Enables the establishment of a root of trust during setup of a blockchain instance, the enrollment and registration of identities or system entities during network operation, and the management of changes like drops, adds, and revocations. Also, provides authentication and authorization.
- **Policy Services:** Responsible for management of various policies specified in the system, such as the endorsement policy, consensus policy, or group management policy. It interfaces and depends on other modules to enforce the various policies.
- **APIs:** Enables clients and applications to interface to blockchains.
- **Interoperation:** Supports the interoperation between different blockchain instances.

3.2 Smart Contract Layer's Interaction with Other Architectural Layers

The smart contract layer works very closely with the consensus layer. A proposal request is sent from consensus layer to the smart contract layer which specifies the contract to execute details of the transaction, including the identity and credentials of the entity asking to execute the contract, and any transaction dependencies. The smart contract layer uses the current state of the ledger and input from the consensus layer to validate the transaction.

While processing the transaction, the smart contract layer uses the identity services layer to authenticate and authorize the entity asking to execute the smart contract. This ensures two things: that the entity is known on the blockchain network, and that the entity has the appropriate access to execute the smart contract. After processing the transaction, the smart contract layer returns whether the transaction was accepted or rejected. If the transaction was accepted, the smart contract layer also returns an attestation of correctness, a state delta, and any optional ordering hints needed to ensure the transaction dependencies are taken into account. The state delta includes the change sets and any side effects that should take place when the transaction is successfully committed by the peers. Below is the generalized view of the smart contracts layer and its interaction with other layers, which is a generalized view since Hyperledger Framework under consideration may choose to implement steps differently.

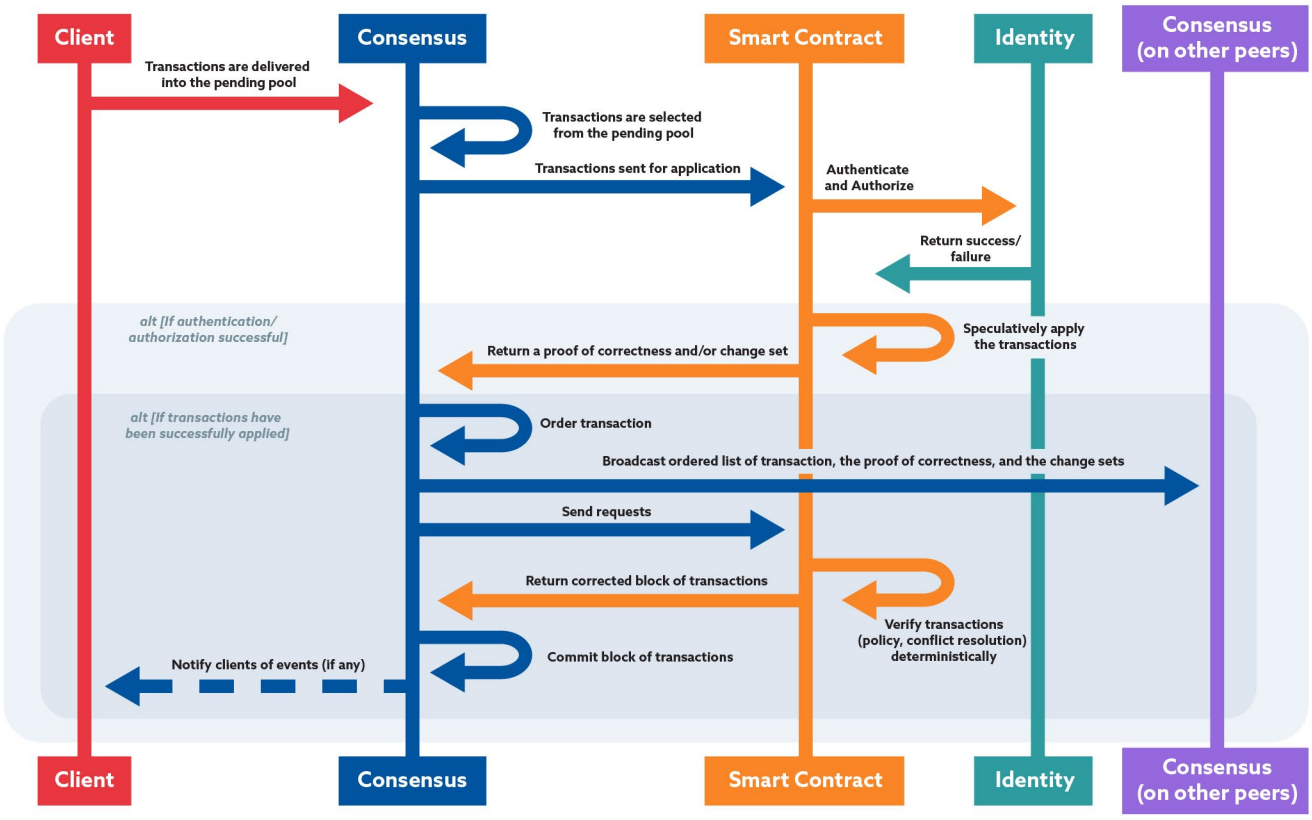


Figure 1. How Smart Contracts interact with other architecture layers

3.3 Hyperledger Frameworks Supporting Smart Contracts

Hyperledger blockchain frameworks supporting smart contracts are Hyperledger Burrow, Hyperledger Fabric, Hyperledger Iroha, Hyperledger Sawtooth and more recently, Hyperledger Besu. Across all these frameworks, the smart contract layer processes transaction requests and determines if transactions are valid by executing business logic. Each framework supports smart contracts in a slightly different way. The Frameworks table lists the smart contract implementations in Hyperledger Frameworks.

Frameworks	Smart Contract Technology	Smart Contract Type	Language(s) for writing smart contracts
Hyperledger Burrow	Smart contract application engine	On-Chain	Native language code
Hyperledger Fabric	Chaincode	Installed	Golang (> v1.0) or Javascript (> v1.1),DAML
Hyperledger Indy	None	None	None
Hyperledger Iroha2	Chaincode	On-Chain	Native language code
Hyperledger Sawtooth	Transaction families	On-Chain and Installed	C++, Go, Java, JavaScript, Python, Rust, or Solidity (through Seth), DAML
Hyperledger Besu	Transaction families	On-Chain and Installed	DAML, Solidity, Vyper

4. Threat Modeling

4.1 What Is Threat Modeling?

Threat modeling is a process of identifying potential vulnerabilities and weaknesses in a system that could be exploited by an attacker. In other words, an attacker with an intent and a capability that can exploit a vulnerability, is what we call a threat. The vulnerabilities, weaknesses, and other risks can then be addressed through mitigation tactics, by revising architecture, by accepting them, through adding compensating controls, and so on. Technical threat modeling usually (but not always) involves the creation of a data-flow diagram (DFD).

Please note that while this article is focused on technical threat modeling of information systems, it can also be applied to other systems and security, such as physical buildings, locks, and even abstract things like business processes.

4.2 Why Create a Threat Model?

A threat model allows you to identify potential vulnerabilities and weaknesses, from architectural issues all the way down to operational levels. When threats are identified, you can then deal with them in a structured and methodical way. There are several reasons threat models are now more important than ever:

1. Systems are more complex now. Twenty years ago, a web application was comprised of 5-10 parts typically (a three-tier architecture, authentication, logging, connection to a handful of other systems, etc.), whereas nowadays a web app like Netflix or eBay has literally thousands of complex components and integration with dozens or hundreds of external systems (payment, orders, shipping, often from multiple providers, etc.).
2. We need to be able to validate that the model actually reflects reality, especially as things change over time. This means you need it written down, ideally in a way that can be shared easily with multiple people (operations, auditors, etc.) and supports versioning, collaboration, and so on.
3. Threat models can be created and applied not just at design time, but both prior to design (e.g. initial scope of work phase) and post-design, such as during development, deployment, operations, and end-of-life.
4. There is now tooling for some specific use cases (mostly web applications and mobile applications) that can be used to simplify and speed the process up significantly.

When you have a good threat model, you can now look at the following activities:

1. You can examine the system for weak points or vulnerable areas that may need additional work; you can maximize the ROI of your time and effort.
2. You can make better predictions based on proposed changes ("What happens if we supplement passwords with 2FA? What happens if we make it mandatory?") with a greater degree of certainty.

3. You can map (sometimes automatically, depending on the tooling) compliance standards and regulations to your threat model. For example, you can identify which parts of the model GDPR might apply to, such as account creation, data backups, etc.
4. Tools exist now that allow you to attach weightings (such as CVSS vulnerability scores) and values (such as the need for AIC/CIA) to various aspects of the system which can also help guide spending on security efforts.
5. Tooling now exists that can create a threat model that also has real world information like software and versions used, when vulnerability (e.g. CVE, vendor advisories) data is released that affects your system, it can be flagged. This can generate reporting on severity and importance of applying fixes or otherwise dealing with vulnerabilities.

4.3 Isn't a Threat Model Just a Data-Flow Diagram (DFD) with Some Added Vulnerability Info?

Short answer: Yes.

Longer answer: Many existing systems do not have a good data-flow diagram (DFD), or one that is up to date, or that reflects the operational realities of how it now behaves and works. If you do have such a diagram, please go get it and start there, as most of the hard work has already been done. You'll simply need to look at things like permission boundaries to turn it into a working threat model. Once you have a good DFD of your system you'll be able to start mapping out trust boundaries, naming potential weaknesses, and so on.

4.4 When Should You Create a Threat Model?

Ideally, you would create a threat model at the beginning of the lifecycle, when you are designing the system, as this can help you make informed design choices and create a more secure system.

However, not all is lost if you have an existing system without a good threat model. Threat modeling is also extremely valuable for existing systems as it allows you to better understand them by looking at assumptions and the actual as a built and operated system, and making sure they are still correct.

- You want to support pre-system architecture threat modeling, helping to inform decisions, like whether or not you will handle payments or simply outsource it entirely, use a PaaS or an IaaS, and so on. Early on, it can provide significant value.
- You want to support the system during the major formative stages such as architecture, development and deployment, this is where most threat modeling focuses currently.
- You want to support the system through the rest of the lifecycle, operations, end of life, and so on. (Also, it should be noted that most systems are not static and have review and modification phases.)
- Your threat modeling strategy needs to deal with entirely new classes of attacks that may not have existed when you started. In other words, you'll need to be able to deal with unknowns that become known in future; failure to do so leaves you exposed.
- You need to account for how modern systems are built. This means broad and deep supply chains in both software terms (i.e., libraries) and service terms (i.e., IaaS, PaaS, SaaS, etc.)

4.5 Threat Modeling Basics

A threat model is only as good as its understanding and representation of the underlying system(s). In general, a threat model starts with the high-level data-flow diagram (DFD) of the system, followed by the data flows and then additional things, such as possible user actions and interactions with external systems. Alternatively, you can work bottom-up; this is an especially effective technique for dealing with existing systems where you may no longer know the original assumptions/data flow architecture.

While you can look at only the surface of a system (e.g. publicly exposed components) and create a threat model from that, it is far more valuable to understand and include the underlying system. Does the system use SQL? NoSQL? Is there caching? A single point of failure? A load balancer? Also modeling the data flows (Are there backups that could result in data exposure?) as well as user actions (Can a user request a data export? Can they delete their account? How is deleted data handled?) can provide insight into possible exposures and vulnerabilities, as well as their potential severity.

One major challenge in threat modeling is striking a balance between having necessary details, and abstracting things away that are less important, or not important at all and simply a distraction. For example, if you are constructing a threat model for a datacenter, you will most likely need to include power delivery related items, but if you are building a threat model for a mobile application, you probably won't need to worry about power delivery issues, you can safely assume that is handled by the datacenter. However, you may want to consider what happens if a server(s) is suddenly unavailable, regardless of reason (i.e., Where is state data held? Is it replicated to other nodes?).

4.6 Shostack's Four-Question Frame for Threat Modeling

Adam Shostack has developed a simple set of four questions that help frame threat modeling:

> 1. What are you building?
> 2. What can go wrong with it when it's built?
> 3. What should you do about those things that can go wrong?
> 4. Did you do a decent job of analysis?

4.7 Combining Threat Modeling with the OODA Loop

Most decision making processes can be done using the OODA loop model:

Observe > Orient > Decide > Act

However, there are nuances to threat modeling that can be applied to make it more effective.

4.8 Threat Modeling Process Overview

Most threat modeling methodologies, be they risk-based, application-based, asset-based, privacy-based, and so on, share several traits in common:

- Define business objectives/requirements
- Define scope of threat model coverage
- Identify assets and create architecture diagrams, data-flow diagrams (DFD)
- Identify weaknesses, vulnerabilities, and threats
- Analyze risk and impact
- Mitigation, remediation, and so on

Several of the threat modeling methodologies essentially include all the above steps (e.g., P.A.S.T.A. aka Process for Attack Simulation and Threat Analysis) and several only address specific steps (Spoofing, Tampering, Repudiation, Information disclosure (privacy breach or data leak), Denial of service, Elevation of privilege aka STRIDE, for example, helps identify weaknesses, vulnerabilities, and threats).

4.8.1 Define business objectives/requirements

Realistically, if your project hasn't started out with defining business objectives and requirements, then it will be extremely expensive, if at all possible, to retrofit this on after something has been built. This is an integral part of any project but it must be mentioned as it is often not as clearly documented as it should be, or has been subject to changes that are not documented. Getting a current copy of the business objectives and requirements, and getting a commitment that any ongoing updates or changes are communicated quickly to the threat modeling team is critical as it impacts all aspects of the threat model.

One note: Due to the distributed and multi-organizational nature of many blockchain and smart contract projects, it should be noted that different organizations may have different requirements. This is guaranteed to be true if the project is international in scope due to different regulatory and legal environments.

4.8.2 Define scope of threat model coverage

When you have the business objectives and requirements, you'll be able to do a rough scope/architecture plan. This is a good point to decide how far down the rabbit hole you want to go. For example, you may decide to limit threat model coverage to the smart contracts and blockchain network itself, and limit coverage of the underlying computational infrastructure to the virtual machines, trusting that the provider of the actual hardware, network and so on are secure. Alternatively, a more risk averse threat model may also consider the hardware, networks, and even go so far as to consider power delivery and physical security of the infrastructure.

In general, a good threat model MUST model all the components that are controlled directly, and should consider the lower layers of abstraction so that if nothing else a statement such as "a hosting provider must be chosen that provides security at the hardware and below layer that meets [some standard, e.g. SOC 2]" can be made and implemented. If the threat model ignores components that are directly controlled it should be noted why they are out of scope and what, if any, compensating measures are being considered, for example it may be as simple as "Smart Contract security is not considered as a risk due to this specific project being a private Blockchain network with a governance model that includes mechanisms for deciding if a transaction is incorrect and must be undone."

4.8.3 Identify assets and create architecture diagrams, data-flow diagram (DFD)

I think, in general, the threat modeling team should not identify assets and create architecture diagrams, data-flow diagrams (DFD), and so on. This should be done by the group(s) actually creating and implementing the project, and the data should be shared with the threat modeling group. Especially as the project changes, the updated information must be communicated to the threat model group in a timely manner.

If the threat modeling team is expected to identify assets and create architecture diagrams, data-flow diagrams (DFD) then it will most likely be incorrect or incomplete (and certainly out of date) as they attempt to forensically determine exactly what is being done. Additionally, it shows a lack of cooperation and support for the threat modeling effort that will cripple it, if not prevent it completely.

4.8.4 Identify weaknesses, vulnerabilities, and threats

Traditional threat modeling has a wealth of information to draw from, the Common Weakness Enumeration (CWE) list of vulnerabilities, scanning tools, and so on. Such data and tools are still in their infancy for Blockchain and Smart Contracts (as evidenced by the existence of this paper). In general, there are three types of weaknesses, vulnerabilities, and threats in blockchain and smart contracts:

> 1. "Traditional" issues that have a traditional impact "Traditional" issues that have a new and interesting impact
> 2. New issues specific to blockchain and smart contracts technology

The first is relatively easy. For example, Blockchains and smart contracts must run on some sort of compute infrastructure and network, so things like DNS security, operating system patches, and so on, must be applied or else an attacker will be able to gain access through traditional attacks and then execute further attacks. This class of issue can generally be dealt with either directly or through the abstraction and implementation of policy such as "all computer infrastructure must conform to standard [X]" (PCI, SOC 2, etc.).

The second issue is more interesting. There are many traditional vulnerabilities, such as integer overflows or improperly created cryptographic material that have new and novel impacts on Blockchains and Smartcontracts. For example, depending on the consensus algorithm in use, an

attacker can leverage denial-of-service attacks to isolate parts of the Blockchain network and make subsequent attacks against voting systems or Proof-of-Work (PoW) much easier. The third class of attack is of real interest and there is some research going on. Blockchains and smart contracts present some novel approaches to technology, and this includes novel attacks. For example, the support for multi-step atomic transactions, such as "Buy asset A on exchange Y, and then sell asset A on exchange Z for asset B, all at current prices" can be combined with flash loans, and price manipulation to create a situation where the attacker only has to put up several hundred dollars (in fees and loan costs) to leverage tens or even hundreds of thousands of dollars, a situation in which even a minor price manipulation (of less than a percent) can result in a windfall for the attacker.

One method to create a list of weaknesses, vulnerabilities, and threats is to essentially conduct a tabletop exercise and walk through what an attacker might want to do and how they might accomplish it. This allows the use of other existing assets such as persona non grata and lists of existing vulnerabilities (https://csaurl.org/blockchain-vulnerabilities) and incidents (https://csaurl.org/blockchain-incidents).

4.8.5 Analyze risk and impact

Similar to the "Identify weaknesses, vulnerabilities and threats" there is a wealth of traditional information but not a lot of blockchain and smart contract-specific information. Additionally there is the issue of blockchains and smart contracts having some very different behaviors; for example, backups. Some people claim that backups are not required in a blockchain world, if you have a trusted copy of the genesis block (the root of the Blockchain) and the data, such as public keys, needed to validate it. But this ignores some common failures:

1. You will want a backup of your data so you don't have to download all the records in the event of a failure. Recovery is much faster if you have most of the data locally on a near line backup system.
2. In the event of a potential major flaw in the Blockchain software that results in network-wide deletions or alterations, having trusted backups will be beneficial (and your Blockchain software may have unseen flaws).
3. If a transaction needs to be removed and the network is rolled back (an unlikely, but not impossible, scenario) it is a good idea to have a copy of the bad transactions for later forensics and investigation.
4. You also need non-committed data, such as transaction memory pools where pending transactions exist.

Backups are an integral part of IT, however blockchain changes this, especially as a it may be configured to have private records, or records that are shared among a limited subset of the network. Some blockchains (like Bitcoin) have a single global state that is public and everyone has access to, and some like HyperLedger support much more complicated implementations with private data and semi-private data.

In line with this, blockchain and smart contract technologies are still young and rapidly evolving. Additionally, they present complex, interrelated systems where it can be difficult to predict second and third order effects from a risk or an impact on one aspect of the system. We have additional bias in the data available, public financial blockchains that have been attacked and compromised cannot generally hide it (Hey, who did this $600 million dollar transaction?) while private industry blockchains do not generally share data about being attacked and compromised.

To complicate matters worse, there are a variety of recovery options that may or may not be available. For example, there have been documented attacks on public financial blockchains that succeeded, and because of the nature of the attack, recovery involved picking a specific time and re-issuing a new set of tokens to reflect the state of the Blockchain network at that time, using an updated smart contract that didn't have the exploited flaw. This can work but now attackers are increasingly targeting cross-chain systems so that they can move their stolen tokens to other blockchains where they cannot be easily recovered (if at all).

It should also be noted that of the availability, integrity, and confidentiality (AIC) triad of information security, you can recover 'availability' and 'integrity' (indeed, Blockchain can make this much easier), but that recovering 'confidentiality' can be impossible because once it's out, it's out.

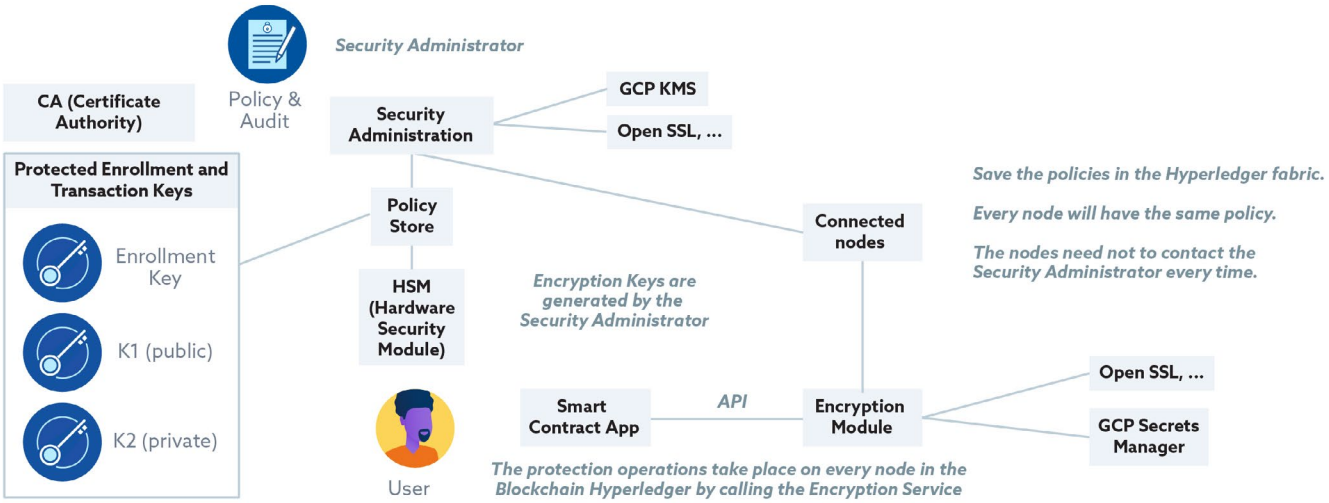
4.8.6 Current attacker sophistication and capabilities

There is currently no good source of data for attacks against private and corporate blockchain systems. However, the public crypto currency ecosystem does have a relatively complete list of attacks, and by virtue of attacks being publicly recorded, they can be verified easily. The level of technical sophistication exhibited by some attackers is quite substantial with several attacks resulting in losses of over $100 million USD (at market value). One site tracking these incidents and providing post mortem details is rekt.news (https://rekt.news/leaderboard/) which includes losses at market value, if the contract was audited or not, and post mortem details.

4.8.7 Protection of Encryption Keys

Keys for data in smart contracts are stored at the smart contract and encrypted per party the keys are shared with. Encryption keys for smart contracts can be secure if the programmer is knowledgeable in this field. An integrated data security and privacy layer can be applied. All accounts can have a profile in which they store keys they use for communicating and sharing keys for encrypted data with other accounts. This profile itself can be encrypted with a key the user holds off-chain and only known to the user. Keys for data in smart contracts can be stored at the smart contract and encrypted per party the keys are shared with.

This is an example of implementation for Hyperledger Fabric with key management for smart contracts on Google Cloud Platform with an architecture based on open-source libraries and cloud services:



- Key generation: An asymmentric key pair K1 (public) and K2 (private) is generated by GCP KMS. Symmetric key encryption key K3, K4, and K5 are generated by OpenSSL.
- Key encryption: OpenSSL encrypts K2 and K4 with K3.
- Data encryption: OpenSSL encrypts data fields with K4.
- Policy encryption: OpenSSL encrypts policy with K5.

Figure 2. Encryption for Smart Contracts. Source: Ulf Mattsson 2022

4.8.8 Compliance with Data Privacy Regulations

Smart contracts offer promise for facilitating and streamlining transactions in many areas of business and government. However, they also may be subject to the provisions of relevant data protection laws such as the European Union's General Data Protection Regulation (GDPR) if personal data is processed. GDPR, has compelled companies to comply with new data privacy regulations if they want to do business in the European Union.

Smart contracts can be secure if the programmer is knowledgeable in this field. Data protection by design and data protection by default should be applied. A compliance data sharing scheme can be applied, aiming to promote compliance with GDPR and other regulations. An integrated data security and privacy layer can be applied.

4.9 Technical Aspects of Smart Contract Security

4.9.1 Guidance on best security practices while creating the Smart Contract

Prepare for failure:

- Pause the contract when things are going wrong ('circuit breaker') to buy time to deal with any problems
- Manage the amount of money at risk (rate limiting, maximum usage)
- Have an effective upgrade path for bug fixes and improvements

Roll out carefully:

- Test contracts thoroughly, and add tests whenever new attack vectors are discovered
- Provide bug bounties starting from alpha testnet releases
- Roll out in phases, with increasing usage and testing in each phase

Keep contracts simple:

- Ensure the contract logic is simple
- Modularize code to keep contracts and functions small
- Use already-written tools or code where possible
- Prefer clarity to performance whenever possible
- Only use the blockchain for the parts of your system that require decentralization

Stay up to date:

- Check your contracts for any new bug as soon as it is discovered
- Upgrade to the latest version of any tool or library as soon as possible
- Adopt new security techniques that appear useful

Be aware of blockchain properties:

- Be careful about external contract calls, which may execute malicious code and change control flow.
- Public functions may be called maliciously and in any order. The private data in smart contracts is also viewable by anyone.
- Keep gas costs and the block gas limit in mind.
- Be aware that timestamps are imprecise on a blockchain, miners can influence the time of execution of a transaction within a margin of several seconds.
- Randomness is non-trivial on blockchain, most approaches to random number generation are gameable on a blockchain.

Fundamental tradeoffs–implicity versus complexity cases:

- Exceptions where security and software engineering best practices may not be aligned.
- Rigid versus upgradeable
- Monolithic versus modular
- Duplication versus reuse

5. Common Hyperledger Smart Contract Security Patterns and Vulnerabilities

Static code analysis tools have been determined to be effective at detecting Hyperledger vulnerabilities. The study did not discuss possible countermeasures for the attack examined, but revealed that the smart contract verification tools are able to detect most of the identified vulnerabilities.

A 2019 IEEE paper, "Potential risks of Hyperledger Fabric smart contracts," discussed several security risks in Hyperledger Fabric smart contracts that are attributed to Go and other general-purpose languages programming language that are used to write smart contracts in Hyperledger as opposed to domain-specific languages (DSLs) like Solidity.

Using a relevant literature going back to 2019, three smart contract vulnerabilities in Hyperledger Fabric (Rich Queries, Pseudorandom Number Generators, and Global Variables) were selected. Smart contracts containing these vulnerabilities were deployed on a test network, and the vulnerable contract features were exploited. What was determined was that proposed countermeasures at a minimum mitigate the impact severity of these vulnerabilities.

5.1 A Closer Look at Two of the Vulnerabilities

5.1.1 #1 - Updates using rich queries

Also referred to as "range query risk". The vulnerability was implemented in a smart contract based on the "asset-transfer-ledger-queries" contract provided with the test network.

- The smart contract can perform rich queries on assets stored in the ledger by using CouchDB as the state database.
- The **ChangeColorByOwner** function, (line 1) introduces the vulnerability into the contract. The color attribute is changed on (line 10), and the asset is written back to the ledger on (line 12).
- The asset obtained from CouchDB is not checked for consistency with the latest committed ledger value which is exploitable.

```
1   func (t *SimpleChaincode) ChangeColorByOwner(ctx, owner string,
          color string) error {
2       queryString := fmt.Sprintf(`{
3           "selector":{"docType":"asset","owner":"%s"}}`, owner)
4
5       // Call wrapper method for getQueryResult()
6       result, _ := getQueryResultForQueryString(ctx, queryString)
7
8       // Change color for all assets and write back the change
9       for _, asset := range result {
10          asset.Color = color
11          assetBytes, _ := json.Marshal(asset)
12          ctx.GetStub().PutState(asset.ID, assetBytes)
13      }
14      return nil
15  }
```

Figure 3. Rich query update example.

5.1.1.1 Exploit: Illegal value propagation

The default access controls provided with CouchDB are no good. If not changed, a malicious actor on the network could access a peer's CouchDB instance with little difficulty.

The attacker could then directly modify and control the world state perceived by that peer without invoking any ledger transactions. Thus, it is difficult for the peer to know whether its state has been changed since there is no record of the changes.

```
4  "ID": "asset1",
5  "value":  "-1",
6  "color": "blue",
7  "ID": "asset",
8  "ID": "Tom",
```

Figure 4. The value has been modified from 300 to -1 through the CouchDB GUI accessible on localhost:5984/ utils.

If combined with the rich queries vulnerability, illegal changes made in the state database can propagate to the ledger.

The attacker needs to modify the state database of a sufficient number of endorsing peers to pass consensus to propagate the change.

5.1.1.2 Countermeasures

Rich query methods like **GetQueryResult** should only be used for query transactions, including Hyperledger Fabric's own documentation, because the query results are not verified in the validate phase.

Besides avoiding rich queries within update transactions altogether, the following design pattern can be adopted:

> - Use **rich query** to retrieve the appropriate keys from the state database.
> - Use a safe key-based query (e.g., GetState) to retrieve the latest committed values from the ledger.
> - This pattern was based on community discussions on the Hyperledger Fabric forums Validating transactions including rich queries.

5.1.2 #2 - Pseudorandom number generator

A secure random number generator (RNG) should be unpredictable. The outcome of invoking a smart contract must be deterministic. These conflicting properties make implementing secure RNGs in smart contracts a challenging problem.

An unpredictable RNG generates a new random number every time it is called. As a result, unpredictable RNG cannot be used in smart contracts; every endorsing peer will calculate a different random number and the contract will be non-deterministic.

To prevent nondeterminism, the outcome of the RNG must be predictable across all peers, for example by using a pseudorandom number generator (PRNG). However, this makes the contract vulnerable to exploitation.

PRNGs generate number sequences by following a deterministic algorithm: Given the same input, or seed, the sequence output by the PRNG is the same every time it is run.
Since all peers need access to the same seed, it must be available on the blockchain.

> - The seed will also be available to a potential attacker.
> - If an attacker knows the seed used by the PRNG, he can predict the outcome and exploit the contract.

This PRNG vulnerability was implemented into a simple lottery smart contract, (see code below).

- The smart contract generates a random number using the math/rand PRNG seeded with the transaction timestamp which is the vulnerable feature of this contract.
- The number is hashed and stored on the ledger for future reference. The participants can then invoke the contract to try and guess the random number to win the lottery.

```
func (s *SmartContract) generateNewWin(ctx) error {
    // Generate and has random number
    timestamp, _ := ctx.GetStub().GetTxTimestamp()
    rand.Seed(timestamp.GetSeconds())
    randomNumber := rand.Int()
    hashedRandomNumber := ...
    // Create and store new winning number
    asset := Asset{
        ID: "current_win",
        Number: hashedRandomNNumber,
        Won: false,
    }
    assetBytes, _ := json.Marshal(asset)
    ctx.GetStub().PutState(asset.ID, assetBytes)
    return nil
}
```

Figure 5. Function used by the vulnerable lottery contract to generate the new winning number using a PRNG.

5.1.2.1 Exploit: Predicting the outcome

The transaction timestamp and the smart contract are available to everyone on the channel.

- A malicious participant can re-calculate the random number and predict the lottery with the information available on the ledger.

To reproduce this scenario, the ledger was inspected using Hyperledger Explorer (v1.1.5) (see image).

- Current Hyperledger Explorer, (v1.1.8).
- CLI can also be used but navigating to the relevant information can be difficult.

The winning number is hidden, but the timestamp used to calculate the number is visible.

- The attacker can then use the same PRNG method to re-calculate the winning number and invoke the contract to win the lottery.

```
Time:  2021-06-06T13:24:29.288Z

Writes:
        ▼ root: [] 2 items
          ▼ set:  [] 1 item
            ▼ 0:  {} 3 keys
                key: "current_win"
                is_delete: false
                value: "{"id":"current_win","number":"158aa5e022a47
                    c1750","won":false}"
```

Figure 6. PRNG method used to re-calculate winning lottery number.

5.1.2.2 Counter measures

Transient data is not recorded on the ledger, so passing the seed as transient input data allows using a PRNG while keeping the seed secret from the blockchain.

- Moves the security issue from the smart contract to the invoking client.
  - The client becomes a single point of failure.
  - Client must generate and store the seed securely.
- If the client is compromised, the smart contract is also compromised.

5.1.2.3 Centralized oracles

The smart contract can then request the random number from a centralized oracle (without deterministic constraints) using a high entropy function to calculate the random number.

- Solution relies on a third-party oracle being trusted and secure since it controls the outcome of the RNG.
- Relying on a (centralized) trusted third-party.
- If the oracle is compromised, so is the smart contract.

5.1.2.4 Decentralized oracles

Distributed RNGs or decentralized oracles can be used. Random numbers are generated based on random input from several participants. The input is then combined in some deterministic way to create the final random number.

- All participants can influence the outcome, no single participant can fully control it, so every endorsing peer of the RNG contract can use a high entropy function to generate a random number stored in private data.
- Endorsing peers of the lottery contract can then retrieve and combine the numbers to create the winning lottery number.

6. Security Tools Embedded in the Smart Contract Development Life Cycle (DevSecOps)

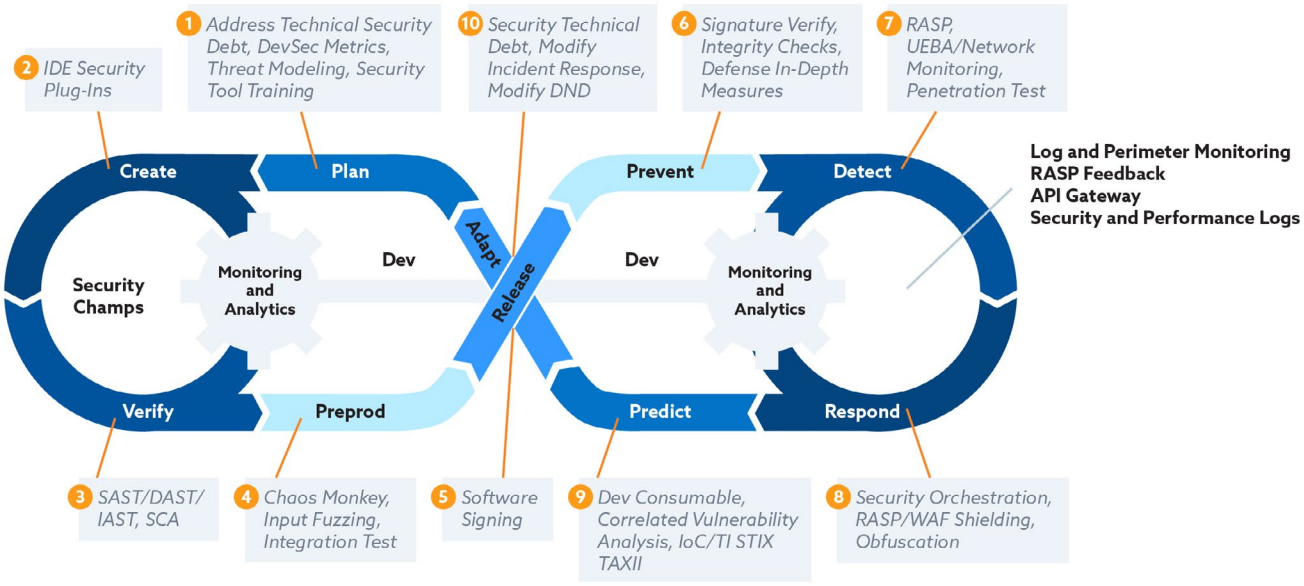


Figure 6. The DevSecOps Toolchain

Chaincode is a program that typically handles business logic agreed to by members of the network and is sometimes called a *smart contract*.

Chaincode in Hyperledger Fabric is similar to smart contracts. It is a program that implements the business logic and is run on top of blockchain. The application can interact with the blockchain by invoking chaincode to manage the ledger state and keep the transaction record in the ledger. This chaincode needs to be installed on each endorsing peer node that runs in a secured Docker container. The Hyperledger Fabric chaincode can be programmed in Golang, Node.js, and Java. Our research effort will only support Golang (Go).

Every chaincode program must implement the Chaincode interface. In this recipe, we will explore chaincode implementation using Go.

6.1 Smart Contract Development Lifecycle

This paper does not attempt to prescribe or recommend a specific lifecycle nor does it intend to replace any existing lifecycle that you or your enterprise employs. The purpose of this section is to provide useful guidance in a manner that is additive and augments your existing solution delivery lifecycle.

One of the keys to securing your lifecycle is by ensuring that it has the appropriate checkpoints, also known as milestones or security gates, into your processes so that a set of security function(s) can interrogate and attest to the security posture of the smart contract or software that is being built at a given point in time prior to deployment. There is much debate around when it is most useful or appropriate to apply a specific security control during a given lifecycle; however, we can rely on well-understood principles of security and software engineering to guide us as to which controls should be applied and when.

A common phrase you may have heard is the idea of "shifting left" where introducing security controls into the leftmost or earliest phase of the lifecycle has been shown to be the most effective time to apply controls and catch software vulnerabilities. In addition to the security implications of identifying and fixing bugs earlier in the lifecycle, there is also an economic and project cost implication where the sooner bugs are identified, the lower the repair cost. Continuous delivery and quality enables your development team to automate propagation of new versions of code through multiple runtime environments.

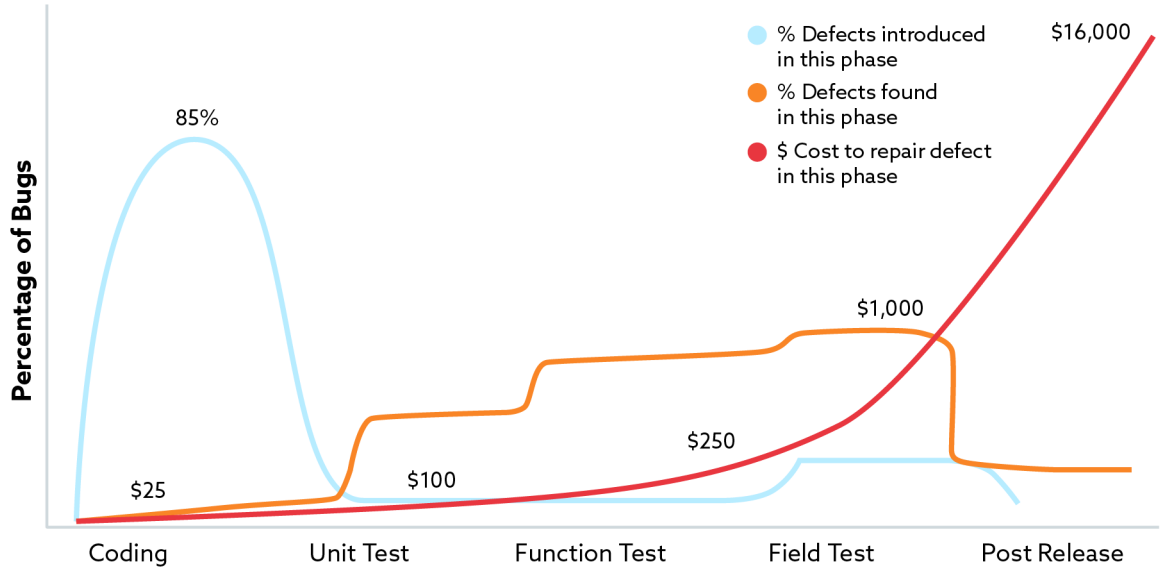


Figure 8. Repair costs and the introduction and discovery of bugs throughout a lifecycle.

The cost of remediation can expand by a factor of 5x for each phase, but more importantly fostering continuous quality can have greater impacts and benefits, such as greater customer satisfaction, promotion of a quality-first mindset, and fewer disruptions in the business. It should be everyone's responsibility to ensure quality is built into the smart contracts and any dependent systems that rely on them. It is critically important to realize the false assumptions that the more bugs you find in smart contracts, the better the contract will be.

6.2 Smart Contract Auditing

Security auditing is an essential part of smart contract development.

Strategies for auditing Hyper ledger Smart Contracts include:

- **Interview Developers:** understand if the business logic is aligned with the code
- **Expert Code Analysis:** have a smart contract expert perform a review
- **Control Flow Analysis:** Generate a graph of the program's control flow and look for anomalies.
- **Dynamic Code Analysis:** Line-by-line review of the smart contract code for logical or programming languages.
- **Vulnerability-Based-Scanning:** Scan the code for known smart contract vulnerabilities
- **Symbolic Execution:** Determine the inputs that cause certain parts of the code to execute
- **Taint Analysis:** Check what variables within the smart contract are controllable by the person or smart contract executing it
- **Test Coverage:** Ensure that unit tests cover all of the smart contract's code
- **Linting:** Ensure that the contract meets style requirements and has no grammatical errors
- **Comments:** Smart contract should have clear comments on functionality

Hyperledger

Patching Smart Contracts

Work prior to contract vulnerability detection
- Symbolic execution ()
- Static Analysis (Go-Tools, Bandit, CppCheck, Cargo Audit, Cargo Clippy)
- Dynamic Analysis (Qira)

Source: Hyperledger.org

6.3 Secure Smart Contract Development Lifecycle (SSCDL)

To talk about DevSecOps, we must not lose focus that this is only the process of automating the tools used by the departments (QA, AppSec, CI) to automate a process. Due to this, to cover the smart contract security process well, we must not only focus on the tools to carry out this process, but also on the methodologies and techniques of these areas in the smart contracts process. For the purposes of this document, only the secure development and development process in the first four layers of a blockchain will be addressed (Application, Contract, Execution and Data).

6.3.1 Application Security for Application Layer

6.3.1.1 Core security training

Documentation for Developers and Team Leads
The part of training for safe secure development by programmers and project leaders in two or three important points to reinforce, the security in code of old languages (Java, Javascript) is common, but for languages like Golang and the security and Microservices is still difficult due to the lack of integration and understanding of what can be done at the code security level and what can be done at the architectural security level.

For the Golang language security training, API and Microservices, the following links with guides and practical exercises can help your team in this area.

Golang, because it is one of the most common languages used in smart contracts with Java and Javascript, and because this last one has bigger communities and is easy to find documentation on the internet like CIS, OWASP, and other documentation about this secure language:

- OWASP Go Secure Coding Practices -Guide- Golang secure coding
- Golang Dynamic training -Laboratory- Intentionally vulnerable go app
- API Cheat Sheet OWASP -API- List of vulnerabilities to hunt and avoid in API
- Microservices Architecture Pattern -Guide- Secure microservices

Of course for automated testing in API and microservices or even training developers, pay tools already exist that all depend on the size of the infrastructure you are running and the cost you want to spend on it.

Golang code libraries for mitigating more common problems in applications:

- OWASP Go Secure Coding Practices Guide

The list of libraries in Go, are only for the purpose of mitigating the most common extensions you need to configure the Blockchain application layer to avoid common attacks.

6.3.1.2 Multifactor authentication

Two-Factor Authentication, or 2FA, is an extra layer of protection used to ensure the security of online accounts beyond just a username and password. For more information, refer to TwofactorGolang,One Time Password utilities Go / Golang.

6.3.1.3 Decentralized authority laboratory

A decentralized system is one in which complex behavior emerges through the work of lower level components operating on local information. For more information, see Decentralized and Distributed Systems Research.

6.3.1.4 Reputation-based method

Reputation systems are programs or algorithms that allow users to rate each other in online communities in order to build trust through reputation. For more information, see this list of available reputation systems and aggregation algorithms.

6.3.1.5 Redundancy

Engineering redundancy is the duplication of critical components or functions of a system with the intention of increasing reliability of the system, which is important to avoid denegation service attack in layer seven of the model OSI Redundancy.

Note: All these tools are referential and only seek to guide the reader on which points to test, regenerating on these and creating their own tools is allowed, since all the tools listed in this paper are open sources.

6.3.2 Application Security for Contract and Execution Layer

Build Your Own is the general idea here, but not build it from zero. The following recommendations are based on the experience in Application Security
Threat Modeling Tools.

- Raindance Project - DevSecOps - Use attack maps to identify attack surface and adversary strategies that may lead to compromise.
- Threatspec - Threat Spec - Define threat modeling as code work like an interpreter.

You need to complement the threat modeling task using SBOM, which is a list of components in a piece of software. Software vendors often create products by assembling open source and commercial software components.

- Awesome-sbom - Software Bill of Materials - There are several links to community developed documents in the NTIA's website.
- SBOM - Software Bill of Materials - It is analogous to a list of ingredients on food packaging.

6.3.3 Static Application Security Testing (SAST)

By definition a compiled language is a programming language whose implementations are typically compilers (translators that generate machine code from source code), and not interpreters (step-by-step executors of source code, where no pre-runtime translation takes place).

Knowing if the language we use is compiled or interpreted in application security allows us to know how the static code tool should work, so if we are in a blockchain project and we get a Golang or Java language scanning tool, it is not enough. It must be a Golang language, but focused on the logic of smart contract because Golang, based on experiences in this area, should use two–one focused on the language and another with that language focused on the technology in which it is used.

- SAST tool for Golang Language
- SAST tool for Smart Contract to use Golang

6.3.4 Software Composition Analysis (SCA): Review Dependencies or Code Libraries in Golang

The SCA is running continuously like an antivirus database but this case is for insecure code third-party libraries.

It is not currently used in the majority of the projects to create wallets, or even smart contracts, but let's understand that this is a tool that searches folder directories for all the code libraries attached to it; it does not matter if they are from the front end or the back end. They can use an API such as NVD National Vulnerability Database.

Using libraries like OWASP Dependency Check, or creating your own integration API with NVD and automating this with Jenkins, will not take more than a couple of days and can save you the process of a reengineering pass due to insecure Java, Golang, JavaScript Libraries.

Dependency-Track - *OWASP* - Monitor the volume and severity of vulnerable dependencies across multiple projects over time.

Dependabot - *GitHub* - Automatically scan GitHub repositories for vulnerabilities and create pull requests to merge in patched dependencies.

6.3.5 Dynamic Application Security Testing (DAST)

For this type of testing, let's maintain the practice that the tests should be based on the functionality of the application that we are trying to test. In this case, they are the smart contracts in the hyperledgers. For this, let's do dynamic tests with a Python script and a runtime server as Jenkins on five different Docker Container images:

First, we need to test the messages associated with the cryptography process. Chipset Strength, Hash, and Random Numbers.

In the second image(Path selection), the reconnection process must be tested–the parallel connection between and the strength of the mathematical algorithm of the path selection tree for nodes.

The third (Attachments) checks for unsafe components such as malware, files and interactions with other technologies.

The fourth (User access) permit the user to access the code, fake authorized user modify the code.

The last image (or data layer) used in the container is for mixed integration tests, where all these tests are executed together periodically.

For wallets tested as if they were in a mobile or desktop application, examples of dynamic testing tools thinking broad spectrum is important here, not only to test smart contracts with each other, but within a network and between nodes, for this use emulates the interactions of the contracts using this lab fabric test network. See Fabric test network.

Mininet - Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native), in seconds, with a single command.

6.3.6 Fuzzing

Fuzzing (also known as Fuzz testing) is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer.

The success in the operation of the fuzzing tests will be proportional to the knowledge of the person who creates them and also the dictionary of definitions the test use and the point to apply. This last one is currently not working because they are not expanding to test the resistance of the algorithm search tree of the contracts in the hyperledger or the possible combinations of cryptography and are relying more on validating things like key length and random number generation.

The fuzzing tests need to also cover:

- Cryptography use between nodes
- Redundancy of the algorithm search tree
- Strength of random numbers for session and keys
- Strength of the hashes

Gofuzz-fuzz - is a library for populating Go objects with random values.

Oss-fuzz-fuzz-OSS-Fuzz aims to make common open source software more secure and stable by combining modern fuzzing techniques with scalable. For the data layer, perform fuzzing tests to alter indexing, weaken hash, and memory space overflow.

Creating your own fuzzing is also allowed using fuzzing-markdown-parser.

Note: All these tools are referential and only seek to guide the reader on which points to test, regenerating on these and creating their own tools is allowed, since all the tools listed in this paper are open sources.

6.3.7 Quality Assurance for Golang Language

unused - Checks Go code for unused constants, variables, functions and types.

goconst - Find repeated strings that could be replaced by a constant.

gocyclo - Computes and checks the cyclomatic complexity of functions.

errcheck - Detects unchecked errors in Go programs.

go-QA - Test framework designed to be used as a base for more specialized test frameworks. For instance, GUI engine, performance/stress, or API testing.

Not only test the code and the functionalities, test the performance.

Caliper - Hyperledger Caliper is a blockchain benchmark tool, it allows users to measure the performance of a blockchain implementation with a set of predefined use cases.

6.3.8 Continuous Integration (CI) and Continuous Delivery (CD)

Tools to identify vulnerabilities in the DevSecOps pipeline:

Insider - Make source code analysis to find vulnerabilities right in the source code, focused on an agile and easy to implement software inside your DevOps pipeline.

huskyci - Makes it easy to find vulnerabilities inside your CI process

gitHubSARIF - Integrate the scans tools (source, dependency or dynamic tool) at you repository like GitHub, Gitlab

vscode-attack - This extension provides IntelliSense-like support for MITRE ATT&CK in the IDE Visual Studio Code, this allow to the developers fix the vulnerabilities and the same time the write the code in the IDE.

Checkov - *Bridgecrew* - Scan Terraform, AWS CloudFormation and Kubernetes templates for insecure configuration.

Clair - *Quay* - Scan App Container and Docker containers for publicly disclosed vulnerabilities.

Dagda - *Elías Grande* - Compares OS and software dependency versions installed in Docker containers with public vulnerability databases, and also performs virus scanning.

Docker-Bench-Security - *Docker* - The Docker Bench for Security is a script that checks for dozens of common best-practices around deploying Docker containers in production.

Manual Testing - The manual security testing needs to be tested against the most famous attacks in smart contracts:

- Blockchain Ingestion
- Wallet theft
- Double spending
- Crypto jacking
- Smart contract DoS
- Reentrancy attacks

- Overflow attacks
- Replay attacks
- Short address attacks
- Balance attacks

Additional notes: Other technologies, on the other hand, such as Runtime Application Self Protection (RASP) artificial intelligence are still too unstable to be considered for use in smart contract technologies.

7. Smart Contract Patching Process

Fabric v1.4 (Legacy) allows the upgrade of an instantiated smart contract to add or remove functionality from its code, but the new version of the smart contract must be installed on all peers in the channel that are running the previous level of the smart contract.

7.1 New Members Joining the Channel

If a new member joins the channel and runs the smart contract, an update of the smart contract is necessary by installing a new version on all the peers, and instantiating it on the channel with a modified endorsement policy that includes the new member.

7.2 Risks Related to Quorum for Endorsement

If a channel peer does not upgrade to the newest version of the smart contract, that peer and any other peer that is not running the newest version will no longer be able to endorse transactions for the smart contract which depending on the endorsement policy creates a risk of not having enough endorsements to commit transactions. Note: It is possible for peers to catch up by installing the new smart contract version.

7.3 Private Data and Removal of Peers

When an upgrade removes an organization from private data collection, those removed peers continue to store data in the private data collection until the block that removes their membership is reached, at which point, the peers will no longer receive private data, and related clients will no longer be able to query the private data.

Fabric v2.0 uses a distributed process to manage the lifecycle of a smart contract and the allowed updates to a channel as part of the decentralization instead of a single organization's administrator making decisions for all organizations about when a smart contract is updated.

An organization desiring to update a smart contract proposes a new smart contract package including definition to the channel and installs the package on one or more of its peers.
Channel members are able to review the source code, but need only approve the smart contract definition, (not the package itself) and are not required to install the package on their peers, thus they may:

> - Approve the definition.
> - Approve the definition and install the package on one or more of their peers.
> - Approve the definition and upload their own package, provided it does not change any agreed to business logic.

After the approvals specified in the channel lifecycle endorsement policy are satisfied, any approving channel member can commit the policy.

8. Smart Contract Freeze/Unfreeze Process

As an emergency measure to deal with ongoing incidents, smart contracts can support a freezing process. The smart contract can be built with a "pause" variable that can be set (e.g., by the owner, a majority consensus, etc.) and unset, as needed. While this "pause" variable is enabled, the contract will not process transactions other than unsetting the "pause" variable to make it live again and possibly also allow an upgrade of the contract while in the paused state.

If implemented, this feature can prevent incidents. For example, if an attack or a vulnerability is spotted, the contract can immediately be paused until the vulnerability is confirmed or not, and the contract can potentially be upgraded to solve the issue if such an action is supported.

Please note that this pause feature is of limited use in cases where an attacker sends a single transaction to launch an attack. It's possible that by the time the attack is spotted, it is over (i.e., all funds may have been removed and there is nothing left to steal).

9. Smart Contract Termination Process

On February 20th/21st 2022, a phishing attack was launched against opensea.io (a large NFT marketplace) users. Several users (17 reported) fell for the attack and lost NFT's from their portfolios to the attacker. The attack relied on email phishing, a fake website, wallet user interfaces that did not make it clear as to what exactly was going on, and most importantly, a 4-year-old abandoned contract from opensea.io that had a vulnerability within it. Had this 4-year-old abandoned contract been terminated, this specific attack would not have been possible.

Much like the satellite industry deorbiting a satellite costs money, terminating a contract takes time and effort, especially if there are any assets still associated with it, it's not clear on the best way to allow forced updates for non responsive users or users that do not want to migrate away from the old contract. It is also not clear what defines an "abandoned" contract (e.g. timeframe? Replacement by another?) and these questions will need to be discussed more by the industry before a solution, if any exists, can be found.

10. Smart Contract Blocklisting Process

One possible alternative to terminating old abandoned and unsafe contracts is to allow wallets to blocklist these contracts and either prevent user interaction with them, or at least warn them. Again it is unclear exactly what constitutes an unsafe or abandoned contract, who should decide this, distribute the information, and so on. These questions will need to be discussed more by the industry before a solution, if any exists, can be found.

11. Case Study

11.1 Accord Project - Hyperledger Fabric and Trade Finance Use Case

The Accord Project is part of the Linux Foundation, a non-profit and open source ecosystem. The mission of the Accord Project is to provide a coding framework to enable natural language smart legal contracts and commercial agreements. The term "legal" should be emphasized, as this platform, along with other solutions, such as OpenLaw.io (OpenLaw makes it easy to create legal agreements that work with Ethereum), are targeted toward legal agreements between parties, not just executable smart contract chaincode.

According to their website, accordproject.org, the Accord Project is "an open ecosystem enabling anyone to build smart agreements and documents on a technology-neutral platform." The goal of the Accord Project is to provide universal technology and tooling that:

> - Introduces a common format for smart agreements, reducing the need to adopt and learn different technologies and futureproofing templates.
> - Enables sharing and reuse of agreement templates.
> - Facilitates the use of executable agreements across any infrastructure: cloud, blockchain, IoT.
> - Interoperates with any distributed ledger and blockchain platform, today and in the future.

The Accord Project is designed to support commercial legal agreements and connects with prominent blockchain platforms including Hyperledger Fabric and R3's Corda platform. Because of their blockchain technology-agnostic stance, templates developed using the Accord Project code base can be used across many environments, including smart contract connectors for Microsoft Word that are being developed.

There are several core components of the Accord Project, including Cicero, which is a domain-specific language used to develop templates for agreements; Ergo, which allows a smart contract developer to write executable business logic for contracts; and Concerto, which enables a smart contract developer to build data models for smart documents. The Accord Project also offers a Template Studio, which enables a smart contract developer to build, edit, and test their smart contract using an online editor.

An Accord Project smart contract use case template entitled Supply Agreement Loc was developed by John Carpenter for the purpose of demonstrating the power of the Accord Project development framework as well as providing the authors of this paper with a fully vetted and tested smart contract template to deploy to Hyperledger Fabric for security testing.

The smart contract that was developed contains a supply agreement with trade finance between two cross-border parties. The primary contracting parties to the agreement are an importer and an exporter, although there are many other parties involved in the smart contract. These include an importer's bank, a shipper, and other relevant parties.

Some of the primary considerations that were taken into consideration when drafting the natural language smart contract were: What is the currency used for the cross border transaction; how does the importer demonstrate that they have the necessary creditworthiness to undertake the transaction; the quantities and delivery method; and the acceptance criteria for the agreement. Additionally, there was a focus on the criteria that could affect the valid delivery and acceptance of the goods, such as IoT sensor readings within a temperature and humidity controlled shipping container.

One of the benefits of using an Accord Project template for developing a smart contract is that there are a large number of existing models that can be utilized in order to reduce the amount of coding required to produce a viable natural language smart contract. The Accord Project models used for this particular smart contract included:

- org.accordproject.contract.*
- org.accordproject.runtime.*
- org.accordproject.geo.Country
- org.accordproject.money.MonetaryAmount
- org.accordproject.time.Duration
- org.accordproject.time.Period
- org.accordproject.time.TemporalUnit

A diagram for the Template Model:

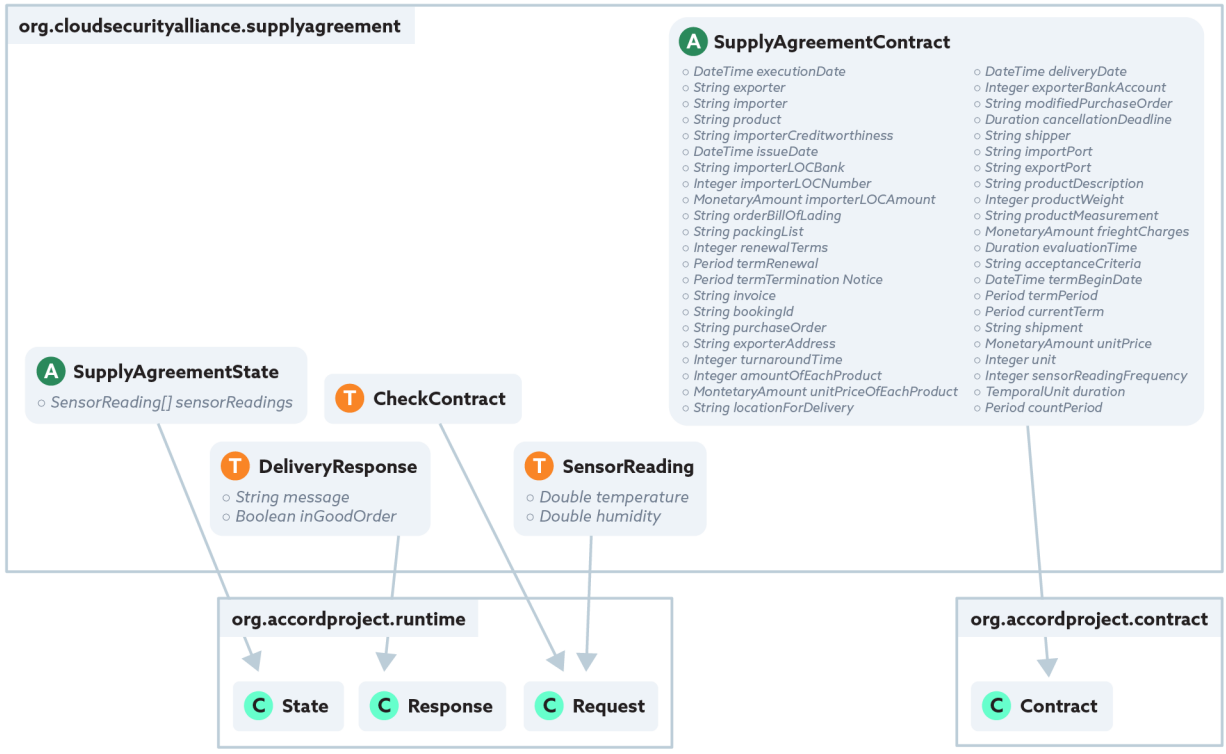


Figure 9. Template model diagram.

12. References

Accord Project. (n.d.). *Accord Project*. Retrieved July 13, 2021, from https://accordproject.org Carpenter, J. (n.d.). Supply Agreement Loc. Accord Project. Retrieved May 12, 2022, from https://templates.accordproject.org/supply-agreement-loc@0.1.0.html

Dawson, R., PegaSys, & Baxter, M. (2019, August 29). *Announcing Hyperledger Besu*. Hyperledger Foundation. Retrieved May 12, 2022, from https://www.hyperledger.org/blog/2019/08/29/announcing-hyperledger-besu

Diligence, C. (n.d.). *Ethereum Smart Contract Best Practices*. GitHub. Retrieved May 12, 2022, from https://consensys.github.io/smart-contract-best-practices/general_philosophy/

Ding, M., Li, P., Li, S., & Zhang, H. (2021, June 21). *HFContractFuzzer: Fuzzing Hyperledger Fabric Smart Contracts for Vulnerability Detection*. Semantic Scholar. Retrieved May 12, 2022, from https://scholarphi.semanticscholar.org/?file=https://arxiv.org/pdf/2106.11210v1.pdf

Hyperledger. (n.d.-a). *A Blockchain Platform for the Enterprise — hyperledger-fabricdocs main documentation*. Hyperledger Fabric. Retrieved July 12, 2021, from https://hyperledger-fabric.readthedocs.io/en/release-2.2/

Hyperledger. (n.d.-b). *Hyperledger Architecture, Volume II: Smart Contracts*. Retrieved May 12, 2022, from https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf

Hyperledger Foundation. (n.d.). *Hyperledger – Open Source Blockchain Technologies*. Retrieved July 12, 2021, from https://www.hyperledger.org/

Hyperledger Whitepaper. (n.d.). *Blockchain Lab*. Retrieved May 12, 2022, from https://blockchainlab.com/pdf/Hyperledger%20Whitepaper.pdf

Isaiah, A. (2022, April 8). *A guide to linting Go programs*. Freshman — Articles and Tutorials on Software Development. Retrieved May 12, 2022, from https://freshman.tech/linting-golang/

Lal, C., & Marijan, D. (2021, March 18). *Blockchain Testing: Challenges, Techniques, and Research Directions | Semantic Scholar*. Semantic Scholar. https://www.semanticscholar.org/paper/Blockchain-Testing%3A-Challenges%2C-Techniques%2C-and-Lal-Marijan/ce2a579214fdf6d6d63144503b10c4236e6313c9

mattumd. (2019, June 28). *Writing Hyperledger Fabric Chaincode Using Go Programming Language*. GitHub. Retrieved May 12, 2022, from https://github.com/IBM/IBMDeveloper-recipes/blob/main/writing-hyperledger-fabric-chaincode-using-go-programming-language/index.md

Muscara, B. (2022, April 3). *Key Terminology: - Learning Materials Development Working Group - Hyperledger Foundation*. Hyperledger Foundation. https://wiki.hyperledger.org/pages/viewpage.action?pageId=2394127

Pal, T. (2018, June 1). *Focusing on the DevOps Pipeline - Capital One Tech*. Medium. https://medium.com/capital-one-tech/focusing-on-the-devops-pipeline-topo-pal-833d15edf0bd

Paulsen, C. (2021, July 2). *Revisiting smart contract vulnerabilities in Hyperledger Fabric | TU Delft Repositories*. TUDelft. Retrieved May 12, 2022, from https://repository.tudelft.nl/islandora/object/uuid:dd09d153-a9df-4c1b-a317-d93c1231ee28

Smalling, E., & Dillon, G. (2022, February 9). *Go Security cheatsheet | Snyk Blog*. Snyk. Retrieved May 12, 2022, from https://snyk.io/blog/go-security-cheatsheet-for-go-developers/

SmartBear. (n.d.). *What Is Code Review? Smartbear.Com*. Retrieved May 12, 2022, from https://smartbear.com/learn/code-review/what-is-code-review/#:%7E:text=Code%20Review%2C%20also%20known%20as,like%20few%20other%20practices%20can.

Wikipedia contributors. (2022, March 26). *Bundle of rights*. Wikipedia. https://en.wikipedia.org/wiki/Bundle_of_rights

Wright, A., Roon, D., & ConsenSys AG. (n.d.). *OpenLaw*. OpenLaw. Retrieved May 12, 2022, from https://www.openlaw.io

Yamashita, K., Nomura, Y., Zhou, E., Pi, B., & Jun, S. (2019, February 1). *Potential Risks of Hyperledger Fabric Smart Contracts*. IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/8666486/