



COMPOSABLE
SECURITY

NOBULLSH*T
**SECURITY
GUIDE**

FOR

**DApps CTO's, Lead Developers
and security enthusiasts**

A word of introduction:

As a Dapp CTO or Lead Developer, you've got no time for nonsense. We get it. In the dynamic landscape of Dapp development, ensuring security can often feel like navigating a minefield.

That's why we've created the **NO BULLSH*T Security Guide**. This forthcoming resource cuts straight to the chase, delivering practical, effective security strategies without the fluff.

Security is much more than audits, security contests or bug bounty programs. We want to equip you with the knowledge necessary to introduce a security centric approach.

The purpose of this guide is:

- giving you complete knowledge and understanding of the currently available options and tools to increase security
- providing practical examples of how to introduce them in the organization
- focus on practical aspects and implementation of knowledge in small, simple steps

Why do we know what we say and you can trust us?


- we spent over 6 years in traditional security, where we worked for international fintech and the largest Polish banks
- in 2019, we created the Smart Contract Security Verification Standard, which is the most comprehensive checklist for Solidity contracts to date
- we work with projects like FujiDAO, Enjin, Tellor, DIVA, Volmex Finance and our clients keep coming back
- there is no other guide like this one, so you have no choice anyway

Thank you!

We would like to thank all CTOs, Developers and friends from the security community for their feedback, tips and belief in the value of this project.

BUILDING SECURE TEAMS.....	5
Threat Modeling.....	7
Peer Review.....	9
Policies and Dependencies.....	11
Security Champions.....	13
Security Resources and Education.....	15
Phishing Campaign Trainings.....	17
Security Aware Culture.....	19
SECURITY SERVICES AT YOUR DISPOSAL.....	24
Security Consultations.....	26
Smart Contract Security Review by Professional Company.....	28
Security Contests.....	31
Smart Contract Security Review by Solo Auditors.....	34
Bug Bounties.....	35
Security Insurance (post audit).....	39
Formal Verification.....	43
Smart Contract Monitoring.....	46
INCREASING EFFICIENCY.....	52
Audit Readiness Checklist.....	54
Due Diligence.....	56
Maximizing the Use of Audit Reports.....	58
Security Patterns to Enforce.....	60
Automatic Tools.....	62
Common Smart Contract Vulnerabilities.....	78
Smart Contract Security Verification Standard.....	98
Secure Protocol Upgrades.....	103

BUILDING SECURE TEAMS



In this chapter, we focus on options and techniques aimed at your team.

Find out what should be done to increase interest in security, minimize the costs of fixes and the number of vulnerabilities found by auditors.

Threat Modeling

DESCRIPTION:

Threat modeling is a proactive approach to identifying, understanding, and managing potential security threats. You pick apart your system, find where it could break, and plan how to stop that. It's about keeping your system tough by tackling risks head-on before they turn into real problems.

GOAL:

Writing code with preserved security awareness

PROS:

- Reduced number of vulnerabilities in the code
- Save time and money spent on fixes

CONS:

- Additional development effort (Big session 3-9 hrs & 3-5 ppl, Small session 5-45 min & 1-3 ppl)

WHEN:

- Big session - before major change or initial architecture design
- Small session - before any change in the code that affects the logic

ACTIONS TO IMPLEMENT:

- Map Your Territory:** Sketch how data flows in your dApp. Include everything - user interfaces, contract functions, external calls. Know your playing field.
- Spot What's Valuable:** What does your smart contract protect? User data, tokens, private keys? Find out what's at stake.
- Spot the Weak Spots:** Look at your map. Where could things go wrong? Consider common screw-ups like reentrancy attacks or front-running. Figure out who would want to mess with your system, and how they'd do it.
- Get Your Priorities Straight:** Some threats are scarier than others. Figure out which ones could hurt you the most and deal with them first. This might mean fixing your code, adding extra security layers, or changing your system's layout.
- Keep It Up:** Don't rest easy. Your project changes, new threats pop up. Regularly revisit your threat model, adjust your map, reassess the risks, toughen up your defenses. Stay alert.

READ MORE:

[Threat Modeling for Smart Contracts: Best Step-by-Step Guide](#)
[ChatGPT-driven threat modeling for smart contracts](#)
[OWASP Threat modeling cheat sheet](#)

Peer Review

DESCRIPTION:

Peer review is when you get your team to scrutinize each other's smart contract code. They're on the hunt for security gaps or bugs that slipped through. It's your secret weapon for catching errors early, keeping your smart contracts solid, and saving you from security headaches down the line.

GOAL:

Improving code quality and catching repetitive bugs

PROS:

- Increases team knowledge
- Very little extra time per session

CONS:

- It takes a while before it brings the right results
- Requires maintaining and updating the knowledge base

WHEN:

Points you specify e.g. before major release, every week on Friday

ACTIONS TO IMPLEMENT:

- Set the Rules:** Make clear what you're looking for in the review. Bugs, gas optimizations, code clarity - define the targets.
- Pick Your Team:** Get the right eyes on each piece of code. Mix of skills, mix of perspectives. It's not just about finding problems, it's about learning from each other.

- ❑ **Keep it Regular:** Make code reviews a routine, not a last-minute scramble. After each major piece of work, or at set times - find what works for your team.
- ❑ **Make it Constructive:** This isn't about pointing fingers. It's about improving the code and the coders. Encourage feedback, don't punish mistakes.
- ❑ **Use the Feedback:** Found a common mistake? Turn it into a learning moment. Use the findings to improve your coding standards and practices. Expand knowledge base and peer review checklists. They are pointless if you don't act on them.

READ MORE:

[5 code review best practices](#)

[Google Code Review Developer Guide](#)

[SMARTBEAR Best Practices for Code Review](#)

Policies and Dependencies

DESCRIPTION:

Policies and dependencies are the rules of the game and the building blocks your smart contract relies on. Policies set the standards for how your team codes and checks for security. Dependencies, like libraries or external contracts, can be a weak link, so you need to know what you're using, keep them up-to-date, and check they're secure.

GOAL:

Reduce chaos and stay in control

PROS:

- The rules are clear and everyone knows what to do

CONS:

- You also need to monitor how well they are enforced
- Updates sometimes lead to time-consuming rework

WHEN:

The sooner the better, and then regularly

ACTIONS TO IMPLEMENT:

- Code Quality Policy:** Set a high bar for your code. It should be clean, readable, and well-documented. This isn't just about looking good, messy code hides bugs and security risks.
- Testing Policy:** Make testing non-negotiable. Unit tests, integration tests, basic fuzzing. Catch problems before they go live. Automation is your friend here, integrate semgrep and slither with additional detectors.
- Security Review Policy:** Code should be reviewed with a security lens. If it hasn't been reviewed, it doesn't get deployed. Period.

- ❑ **Dependency Policy:** Know what you're using and why. Only use trusted, well-vetted libraries and external contracts. Regularly update and check them for vulnerabilities. Dependency check is your friend here.
- ❑ **Incident Response Policy:** Have a plan for when things go wrong because one day they might. How you respond to a security incident can be the difference between a hiccup and a disaster. Keep it updated, make sure your team knows it and it was practiced.
- ❑ **Secure Upgrade Policy:** Changing your smart contract isn't as simple as pushing an update. You need a strategy for upgrades that maintains security and trust. Use upgradeable contracts wisely. Test the whole system thoroughly in a testnet/local environment first, and always review changes for security impacts. It's not just about adding new features, it's about keeping the old ones safe.

READ MORE:

[G2: Policies and procedures](#)

[White hack policy](#)

[SAFU standard proposition](#)

[Yearn Finance Emergency Procedures](#)

[ToB - Incident Response Recommendations](#)

Security Champions

DESCRIPTION:

Security Champions are your team's defensive linemen. They're the ones with the deep knowledge of your smart contract's security, ready to tackle issues and train your squad. They keep your team's security game sharp, foster a security-first mindset, and are your first line of defense against security threats.

GOAL:

Having internal go-to persons who handle security topics

PROS:

- They'll take a good chunk of work away from you
- More people will share your desire to secure the project

CONS:

- Development may slow down temporarily
- It takes time and some investments to start showing results

WHEN:

After a successful release and after the growth of the team

ACTIONS TO IMPLEMENT:

- Pick Your Champions:** Choose a few folks from your team who have a knack for security. They should be keen to dive deeper and share their knowledge.
- Train Them Up:** Invest in their security skills. Workshops, courses, certifications - make them the experts.

- ❑ **Give Them Time:** Let them focus on security. This isn't a side gig, it needs dedicated time. They're your security radar, catching threats before they get serious.
- ❑ **Make Them Teachers:** Champions should spread the security love. Regular team sessions, one-on-ones, code reviews - they should be lifting the whole team's game.
- ❑ **Listen to Them:** They're not just there for show. If they see a problem, or suggest an improvement, take it seriously. They're your canary in the coal mine.

READ MORE:

[OWASP - Security Champions Playbook](#)

Security Resources and Education

DESCRIPTION:

Security Resources and Education means keeping your team clued up and ready. It's about fostering a culture of continuous learning, accumulating knowledge over time and feeding them the latest info. It's a never-ending process - the more your team learns, the tougher your smart contracts get.

GOAL:

Increasing security awareness and staying up-to-date

PROS:

- Low cost and great effect
- Source of knowledge for new employees and great input for peer reviews

CONS:

- Requires regularity

WHEN:

Now

ACTIONS TO IMPLEMENT:

- Find the Good Stuff:** Sniff out the best resources - webinars, courses, tools, insightful blogs. It's about learning the right things, not just anything. Start with the Block Threat Intelligence Hacks section and of course our blog.
- Create an Internal Knowledge Base:** Build your own library of lessons learned, especially past bugs and how you squashed them. This isn't just a resource, it's your team's collective memory. It helps you avoid repeating mistakes and boosts your problem-solving power.

- ❑ **Make it Accessible:** Keep all resources, both external and internal, in a place your team can always reach. A shared drive, a wiki - make it easy to learn.
- ❑ **Set Aside Time:** Learning isn't an extra, it's part of the job. Give your team regular time to update their knowledge. Remember, an hour spent learning can save days of debugging.
- ❑ **Share and Discuss:** Learning shouldn't be a solo journey. Encourage your team to share their new insights. Regular chats about new findings keeps everyone in the loop and sparks ideas.

READ MORE:

[Composable Security Blog](#)

[Blockchain Threat Intelligence](#)

[Defi Security Summit](#)

Phishing Campaign Trainings

DESCRIPTION:

No matter how good your code is, humans can always be the weak link. Phishing campaign training is about arming individuals with the knowledge and skills to spot and thwart phishing attacks. It involves realistic simulations of phishing attempts, teaching people how to detect suspicious emails, links, or messages and take effective action. This training is a practical defense against phishing threats, empowering individuals to protect themselves and their organizations.

GOAL:

Increasing awareness of phishing attacks among the team

PROS:

- Increasing awareness and reducing the success rate of phishing attacks
- There are ready-made free tools to help with increasing awareness of phishing

CONS:

- Requires constant repetition and offers no guarantees
- Real attempts at conducting a controlled attack require the involvement of a third party company.

WHEN:

Get started with education and free resources ASAP

ACTIONS TO IMPLEMENT:

- Assessment of Vulnerabilities:** Begin by assessing the specific phishing risks and vulnerabilities within your smart contract project. Identify

potential entry points for phishing attacks, such as communication channels, email systems, or shared documents.

- ❑ **Tailored Training Plan:** Develop a customized phishing training program that addresses the unique needs and risks of your project. This plan should include simulated phishing campaigns, interactive workshops, and educational materials that resonate with your team's roles and responsibilities.
- ❑ **Simulated Phishing Exercises:** Conduct realistic phishing simulations to expose team members to phishing attempts. These exercises should replicate common tactics used by attackers. Track how team members respond and provide immediate feedback and guidance on best practices.
- ❑ **Continuous Education:** Implement an ongoing education strategy. Regularly update the training materials and simulations to reflect evolving phishing techniques and threats. Encourage team members to stay informed about the latest phishing trends and share this knowledge within the organization.
- ❑ **Reporting and Incident Response:** Establish clear reporting procedures for suspected phishing attempts. Ensure that team members know how and where to report such incidents promptly. Develop a robust incident response plan to investigate and mitigate any successful phishing attacks swiftly.
- ❑ **Introduce anti-phishing security measures:** Use two-factor authentication, establish verification process for key operations such as face-to-face verification / pgp-signed message. Utilize the filtering system and minimize exposure for non-essential members.

READ MORE::

[FREE Phishing quiz from Google](#)

[Phishing Resources \(Free tools, webinars\)](#)

[How to recognize and avoid phishing attacks](#)

If you need help or would like to conduct an external phishing attack on your organization, please contact us and we will help you organize it.

Security Aware Culture

DESCRIPTION:

A Security Aware Culture creates a team mindset where security is everyone's responsibility. It promotes constant vigilance, knowledge sharing, and proactive measures to safeguard the smart contract project. Learn a few techniques that allow you to work with your team faster and more effectively in this context.

TECHNIQUES:

Get to Know Your Team Fast: Utilize the "Mendeleev" technique for quick introductions among team members, fostering rapport and camaraderie. This technique promotes swift bonding and collaboration among team members.

- Instead of asking to introduce themselves, ask them to tell you what elements from the Mendeleev table they would use to describe themselves and why.
- Start with yourself.

Example:

Person 1

"Hey there, if I had to compare my work style to an element, it might be Gold. Not because I think I'm flashy, but because I love to think outside the box, coming up with fresh ideas and approaches. Just as Gold is versatile in its uses, I'm always open to trying new things and finding unique solutions."

Context: This person often brings new perspectives to discussions and projects. They're the one who's excited about exploring new avenues and isn't hesitant about suggesting a different approach if it might lead to a better outcome.

Person 2

"Hi, if I were to pick an element that reflects my approach, I'd probably go with Iron. Not in the 'strong as iron' sense, but more about being steady and consistent. I like to dive deep into details, ensure that everything is on track, and that we're building on a solid foundation."

Context: This individual values thoroughness and precision. They're the kind of person who double-checks everything, asks clarifying questions, and wants to make sure that all bases are covered before moving forward.

Thanks to the fact that you created the space, people described themselves as they see themselves. After a few sentences, you know exactly what people will want to pursue and in what matters you can rely on them.

Need an idea? - contact Person 1. Do you want to be sure that something will be thoroughly checked? Select Person 2.

No Stupid Questions: Cultivate an inclusive atmosphere where team members feel comfortable asking any security-related question without hesitation. Emphasize that seeking clarity and advice is vital for continuous learning and improving security practices.

- No blame or shame for such questions. Do not allow any team member to do this.
- Set an example, ask questions to the team and show that you don't know everything either.

Share What You Learn: Conduct regular security-focused meetings to discuss ongoing initiatives, recent incidents, and updates on best practices. These meetings serve as a platform for collaborative problem-solving and staying informed on security matters.

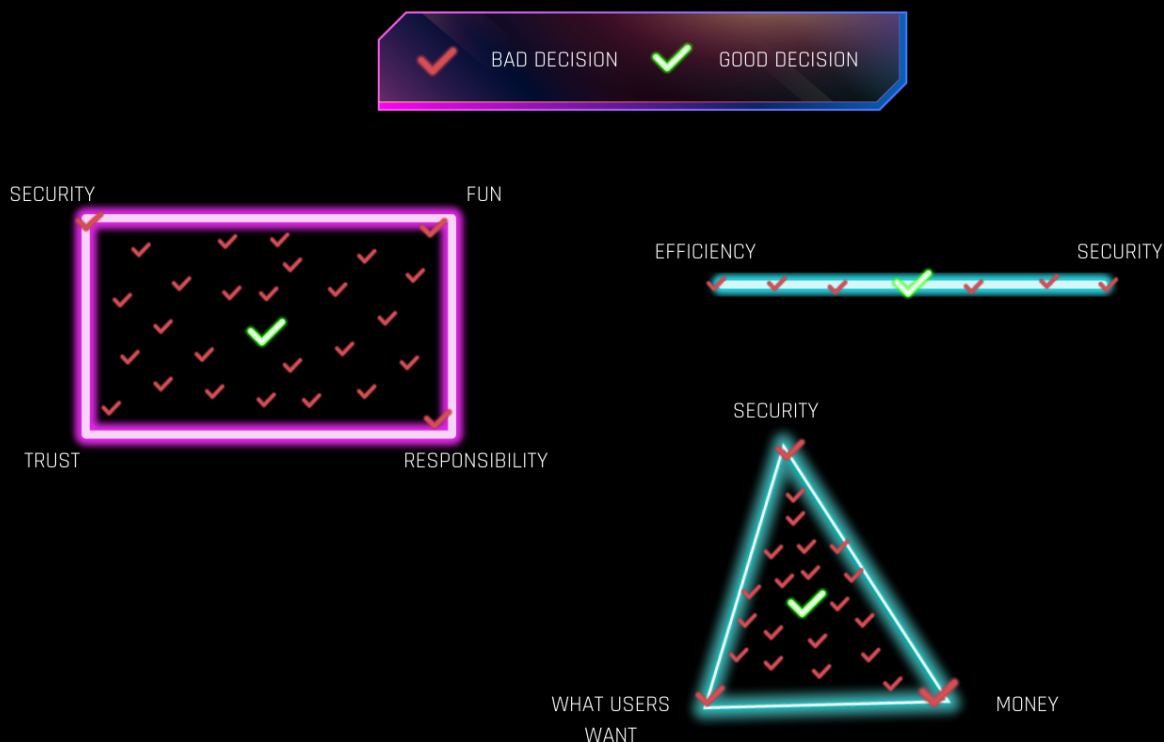
- Set aside a specific day and time each month for team discussions about recent security challenges, insights, or news

- ❑ Plan hands-on security workshops or bring in experts for team training sessions. Encourage team members to lead sessions on topics they're knowledgeable about.
- ❑ Meetings should not be mandatory, but people who engage in them should be noticed.

Values Visualization: Put security as one of the core values of your team. When making important decisions, visualize how you do it. Emphasize that none of the selected values is omitted in favor of the others. This reinforces the importance of security in the team's mindset.

- ❑ Use geometric figures for representing values to develop independence among team members in making the right decisions

Example:



Let's look at the line with two values: *Efficiency* and *Security*.

If we have such values, then facing the choice of whether to do something quickly or safely - we will choose the third option - as soon as possible without compromising on safety.

Let's look at a triangle with three values: *Security*, *Money* and *What users want*.

Users suggested 2 features. One is to add a new pool with tokens that many people want to trade, and the other is to change the architecture to a more trendy one. If we were only guided by the value - *What users want*, we would implement both features. Driven by *What users want* in conjunction with *Money*, we know that we should start by adding a new pool, because it will potentially bring us more profits. By adding *Security* on top of the previous two, we know that we can give users what they want as long as it will be profitable and will not expose anyone to loss.


Security Metrics and Recognition: Define and track security metrics to measure the team's progress in improving security practices. Recognize and reward team members who consistently contribute to enhancing security or identifying critical vulnerabilities. Publicly acknowledging their efforts motivates others to embrace a security-first mindset.

- Prepare a dashboard with key parameters** (such as: the number of issues detected during peer review and audit, their impact on risk, number of internal publications on security, time spent on fixes).
- Complete them regularly** and share progress with your team.
- If you reach a stage where you clearly save a lot of time by minimizing mistakes - spend it on something that will **reward the team's efforts**.

“ No matter how many tools or consultants you use, if you don't invest in developing your team's security awareness, you will still be vulnerable to attacks. ”

**Paul Kurylowicz
Co-Founder of Composable Security**

SECURITY SERVICES AT YOUR DISPOSAL



In this chapter, we focus on the options available on the market that you can use to expand your arsenal.

Find out about available services, examples of providers, and how they differ from each other.

Security Consultations

DESCRIPTION:

Security Consultation is about seeking expert guidance on smart contract security. It's getting an outside perspective to identify blind spots and uncover potential vulnerabilities. It provides valuable insights, recommendations, and strategies to bolster the security of your smart contract-based project.

GOAL:

Getting an outside perspective or delegating research to experts.

PROS:

- Outside perspective without any biases
- Your team can focus on other things

CONS:

- You won't always get the answer you want
- Can be time-consuming (expensive) for specialists who are not familiar with your project

WHEN:

- Before making a decision that considers several solutions
- Before starting work in a new unexplored direction

EXAMPLE PROVIDERS:

- Composable Security (<https://composable-security.com>)
- Prototech labs (<https://www.prototechlabs.dev/>)
- Dedaub (<https://dedaub.com/>)

EXPECTED COST:

- Usually time boxed
- \$

Composable Security average rates for security consultations at the end of 2022 (\$1000-\$8000)

ACTIONS TO IMPLEMENT:

- Find the Right Experts:** Seek out experienced professionals who specialize in smart contract security. Look for those with a proven track record and deep knowledge in the field.
- Define Your Objectives:** Clearly communicate your goals, requirements, and concerns to the security consultants. Establish the scope and focus areas for their consultation.
- Collaborate and Share:** Foster open communication and collaboration with the consultants. Provide them with the necessary access to your codebase, documentation, and relevant information. Engage in discussions to gain insights and address any questions or concerns.
- Act on the Recommendations:** Take the recommendations and suggestions provided by the consultants seriously. Implement the necessary changes, enhancements, or fixes identified during the consultation process.
- Continuous Engagement:** Establish an ongoing relationship with the security consultants. Schedule regular check-ins or periodic audits to ensure the continued security of your smart contracts. Stay proactive and adapt to emerging threats and evolving best practices.

READ MORE:

[Composable Security - Security Consultation](#)

Smart Contract Security Review by Professional Company

DESCRIPTION:

A Smart Contract Security Review by a professional company offers a thorough evaluation of your smart contract code by experienced experts. They dig deep to uncover vulnerabilities, offering practical insights and actionable recommendations to beef up your project's security. With a professional company on your side, you can trust in a thorough and expert-driven approach to safeguard your smart contracts.

GOAL:

Minimizing the risk of a hack and improving code quality

PROS:

- A team of specialists is working on your project
- Work on time, no dependence on individuals
- Access to many specialists
- Simple processing and communication

CONS:

- Can be expensive
- Quality depends not only on the selected company, but also on specialists assigned to your project

WHEN:

- Before the release
- Significant changes in the code
- Cross-check

EXAMPLE PROVIDERS:

- Composable Security (<https://composable-security.com>)
- Dedaub (<https://dedaub.com/>)
- BlockSec (<https://blocksec.com/>)
- Cyfrin (<https://www.cyfrin.io/>)
- more: (<https://defisafety.com/auditors>)

EXPECTED COST:

- Depends very much on the provider
- Usually priced based on:
 - lines of code
 - complexity
 - quality of documentation
- \$\$/\$\$\$

Composable Security average rates for security review at the end of 2022:

- *Single contracts* \$1500 - \$8000
- *Little project* \$5000 - \$18000
- *Medium and big projects* \$18000+

ACTIONS TO IMPLEMENT:

- Choose a Reputable Security Firm:** Do your due diligence and select a professional company with a strong track record in smart contract security. Look for expertise, industry recognition, and positive reviews.
- Define the Scope and Objectives:** Clearly communicate your project's scope, goals, and specific areas of concern to the security firm. Provide access to the separate audit branch, relevant documentation, and system architecture details.
- Collaborate and Share:** Foster open communication and collaboration with the security firm. Be responsive to their inquiries, provide additional context when needed, and engage in regular discussions to exchange insights and address any questions.

- ❑ **Act on the Recommendations:** Take the security firm's findings seriously and prioritize their recommendations. Implement necessary code fixes, architectural improvements, and security enhancements based on their expert guidance.
- ❑ **Periodic Reviews and Continuous Improvement:** Consider periodic reviews by the security firm to ensure ongoing security. Smart contract security is a continuous effort, so establish a long-term partnership to stay up-to-date with evolving threats and industry best practices. Keep refining your security posture based on their insights and emerging trends.

REQUEST AN AUDIT:

<https://composable-security.com/contact/>

Security Contests

DESCRIPTION:

Security Contests in smart contract security are like crowdsourced bug hunts for your smart contracts. By offering rewards, you motivate skilled security researchers and hackers to dig deep and uncover vulnerabilities that may have slipped through the cracks. It's a proactive way to catch and fix security issues before they become real problems, harnessing the power of the community to strengthen your project's security.

GOAL:

Minimizing the risk of a hack and improving code quality

PROS:

- The potential for a much larger number of people involved in finding bugs
- Work on time, no dependence on individuals

CONS:

- Expensive
- A lot of DM's, repetitive questions
- Different quality of findings

WHEN:

- Before the release

EXAMPLE PROVIDERS:

- Code4Arena (<https://code4rena.com/>)
- Hats Finance (<https://hats.finance/>)
- Sherlock (<https://audits.sherlock.xyz>)
- CodeHawks (<https://www.codehawks.com/>)

EXPECTED COST:

- Depends very much on the provider
- \$\$\$
- **Code4Arena** (<https://code4arena.com/sponsor>)
 - a 20% fee on top of the determined contest pool
 - a 40% fee on sponsor tokens
 - Pool determined by nSLOC (e.g, \$30000 per 1000 nSLOC)
- **Hats Finance** (<https://hats.finance/protocols>)
 - If no valid vulnerabilities are identified, the hats protocol gets nothing.
 - Reward only for the first unique submission
 - 20% service fees dependent on audit payout
- **Sherlock**
(<https://docs.sherlock.xyz/audits/protocols/audit-pricing-and-timeline>)
 - ~500 nSLOC - \$16500
 - ~1000 nSLOC - \$32000
 - ~2000 nSLOC - \$64000
- **CodeHawks** (<https://docs.codehawks.com/pricing/>)
 - The CodeHawks/Cyfrin team takes 20% of the total prize pool

ACTIONS TO IMPLEMENT:

- Define Contest Objectives:** Clearly outline the purpose, scope, and rules of the security contest. Specify the types of vulnerabilities you want participants to focus on and address all questions that might arise in advance.
- Set Attractive Rewards:** Offer enticing rewards that incentivize participants to invest their time and expertise. Ensure the rewards align with the significance and complexity of the vulnerabilities discovered. In addition to money, consider something whose perceived value is greater than its real value e.g. swag, promotional interview for the best.
- Engage a Trusted Platform:** Utilize reputable platforms to host your security contest. These platforms provide access to a diverse pool of

skilled researchers and a structured framework for managing submissions.

- **Promote the Contest:** Spread the word through security channels, social media, and relevant communities to attract talented participants. Engage with the community, answer questions, and create buzz around your contest.
- **Thoroughly Address Vulnerabilities:** Act promptly on the findings of the security contest. Prioritize and fix the identified vulnerabilities, and provide feedback to the participants on their submissions. This fosters a collaborative atmosphere and encourages future participation.

Smart Contract Security Review by Solo Auditors

DESCRIPTION:

Solo Audits in smart contract security mean putting your code in the hands of a skilled expert who goes through it with a fine-tooth comb. They dive deep, uncover vulnerabilities, and provide you with actionable insights to fortify your smart contract's security. With a solo audit, you get a focused and expert-led examination without the need for a big auditing team.

GOAL:

Minimizing the risk of a hack and improving code quality.

PROS:

- Relatively cheap compared to alternatives
- You know exactly who is auditing your code

CONS:

- Dependency on the individuals
- Only one expert reviews the code
- Narrow specialization and scope of services

WHEN:

- Mostly as cross-check or initial audit
- Before the release
- Significant changes in the code

EXAMPLE PROVIDERS:

- bytes032 (<https://bytes032.xyz/> Telegram: @bytes_032)
- pashov (<https://github.com/pashov/audits> Twitter: @pashovkrum)
- zachobront (<https://zachobront.com/> Twitter: @zachobront)

EXPECTED COST:

- Depends very much on the provider
- Usually priced based on:
 - amount of code
 - complexity
 - quality of documentation
- \$/\$\$

ACTIONS TO IMPLEMENT:

- Choose a Trusted Expert:** Find a seasoned professional with a proven track record in smart contract security. Look for individuals who have a deep understanding of the blockchain ecosystem and a strong background in auditing. Code4Arena profile or other rankings can be really helpful here.
- Define Audit Scope:** Clearly communicate the specific areas and components of your smart contract that you want the auditor to focus on. Set expectations and provide access to relevant documentation and code.
- Engage in Effective Communication:** Establish regular and open lines of communication with the auditor. Respond promptly to their queries and provide any additional information or clarification they may need to perform the audit effectively.
- Act on Audit Findings:** Take the auditor's findings seriously and prioritize the recommended fixes and enhancements. Implement necessary code modifications and security measures based on their expert insights.
- Continuous Improvement:** Treat the solo audit as part of an ongoing process. Consider conducting regular audits or engaging the auditor periodically to ensure that your smart contract's security remains robust. Stay proactive in addressing potential vulnerabilities and stay updated with emerging best practices.

Bug Bounties

DESCRIPTION:

Bug Bounties in smart contract security are programs that incentivize the community to find and report vulnerabilities in your smart contracts. By offering rewards, you tap into a diverse pool of skilled researchers and hackers, who actively search for and disclose vulnerabilities. Bug bounties provide a proactive approach to identifying and resolving security issues, strengthening the overall security posture of your smart contract-based project.

GOAL:

Creating a dilemma for the real attacker

PROS:

- You can minimize the effects of a real attack
- Pay for confirmed vulnerabilities
- Listing on some platforms is free

CONS:

- Requires very large rewards to attract the best

WHEN:

- After the release

EXAMPLE PROVIDERS:

- Immunefi
- HackenProof
- Hats Finance

EXPECTED COST:

- **Immunefi** (<https://immunefi.com/projects/>)
 - FREE listing
 - FREE triage services. They do automatic + manual triage
 - 10% of the reward paid for valid submission
- **HackenProof** (<https://hackenproof.com/bug-bounty-solutions>)
 - FREE listing
 - FREE triage services in case of smart contracts. They do manual triage
 - triage services cost \$500 per month if it's a big ecosystem with web, mobile, API, etc.
 - 10% bug fee from each valid report, but can be less if you approach them through a partner (contact us to get a discount).
 - They work with CER.LIVE and CoinGecko and as soon as their clients start bug bounty they send information to increase their security ratings
- **Hats Finance** (<https://docs.hats.finance/general/bug-bounties>)
 - On-chain submissions
 - Requires setting up the vault on-chain.

ACTIONS TO IMPLEMENT:

- Define Bug Bounty Scope:** Clearly outline the scope of your bug bounty program, specifying which types of vulnerabilities you are interested in and any specific rules or limitations. This ensures researchers know what to focus on and helps manage expectations.
- Determine Reward Structure:** Establish a fair and enticing reward structure that aligns with the severity and impact of the discovered vulnerabilities. Set clear guidelines on how rewards will be determined and distributed to incentivize researchers to actively participate.
- Choose a Bug Bounty Platform:** Engage a reputable bug bounty platform that specializes in smart contract security. Leverage their expertise, network, and infrastructure to manage the program effectively, ensuring secure and organized communication with researchers.

- ❑ **Promote the Bug Bounty:** Spread the word about your bug bounty program through security communities, social media, and relevant channels. Reach out to specialized forums and conferences to attract skilled researchers. Actively engage with researchers, answer questions, and provide prompt feedback on submissions.
- ❑ **Act on Valid Submissions:** Prioritize and validate bug reports promptly. Acknowledge and reward researchers for valid findings, and communicate with them to gain additional details if needed. Address the reported vulnerabilities swiftly to ensure the security of your smart contract project. Maintain a positive and collaborative relationship with the security community.

Security Insurance (post audit)

DESCRIPTION:

Security Insurance is like having a superhero sidekick ready to swoop in when disaster strikes your smart contract system. It offers financial protection and support to recover from hacks or breaches, compensating affected users and restoring their trust. With Security Insurance, you're prepared to combat the unexpected, minimize losses, and survive this difficult moment.

GOAL:

Smooth recover from the incident

PROS:

- Financial protection and easier recovery
- Increased community trust

CONS:

- Insurance policies often have specific terms, conditions, and exclusions
- Additional cost

WHEN:

- After the release

EXAMPLE PROVIDERS:

- InsurAce Protocol
- Sherlock
- UnoRe
- Chainproof

EXPECTED COST:

- **InsurAce** post-audit coverage
(<https://docs.insurace.io/landing-page/documentation/cover-products/post-audit-cover>)
 - Pricing based on risk assessment
 - The audit should be performed by one of the partners
 - Claim assessments are conducted by the Advisory Board and the Claim Assessors through investigation and community voting
 - Estimated 5-10% of TVL up to \$500k, 8 week cover
- **Sherlock** (<https://docs.sherlock.xyz/coverage/protocols/pricing>)
 - 4% of TVL per year (in monthly premium) for a public audit contest (assuming they meet all other coverage criteria such as a successful fix review, etc.).
 - 4.75% of TVL per year (in monthly premium) for a private audit contest.
 - requires a ~month of upfront payment before coverage can be activated.
 - Sherlock will cover up to 200k USDC of a Critical bug bounty payout.
- **UnoRe**
(<https://unore.gitbook.io/uno-re/08.-cover-definition/smart-contract-coverage>)
 - compensated in \$UNO or \$ETH within 10 working days of claim-filing
 - soon a Dune dashboard will be created to track real-time the revenue of the protocol
 - The calculation considers the risk level assumed by the Liquidity Providers and Uno Re protocol, while also strategizing the policy formulation to maximize the number of wallets protected.
- **Chainproof**
(<https://www.chainproof.co/blog/reaching-a-new-frontier-in-smart-contract-security-and-insurance>)
 - Maximum insurance \$10 mln

- The regulated cyber-insurance carrier for non-custodial smart contracts
- Backed by Sompó and reinsured by Munich Re
- Price based on risk analysis which among others includes: audit by a trusted partner, documentation, incident history, policy composition and more.
- Claims are sent via forms. A technical investigation takes place and an on-chain report is published, which shows whether the policy includes the cause of the hack.

ACTIONS TO IMPLEMENT:

- Research Reputable Security Insurance Providers:** Explore insurance providers with expertise in smart contract security coverage. Look for companies with a proven track record, strong financial backing, and a comprehensive understanding of the unique risks and requirements in the blockchain and smart contract space.
- Evaluate Coverage Options:** Assess the coverage options offered by different insurance providers. Analyze the terms, conditions, and limits of the policies to ensure they align with the specific needs and risks of your smart contract project. Seek clarification on any ambiguous or unclear aspects before making a decision.
- Understand Policy Requirements:** Familiarize yourself with the requirements outlined in the insurance policy. Ensure that your smart contract system meets the specified security standards and best practices to qualify for coverage. Take necessary steps to align your development processes and security protocols with the policy requirements.
- Implement Robust Security Measures:** Strengthen your smart contract security measures to reduce the likelihood of a breach and demonstrate a proactive approach to risk mitigation. Enhance access controls, apply thorough testing and auditing procedures, and regularly update your codebase to address any identified vulnerabilities or weaknesses.

- ❑ **Establish Incident Response Plan:** Develop a comprehensive incident response plan that outlines the steps to be taken in case of a security breach. Define clear roles and responsibilities, establish communication channels, and establish protocols for notifying the insurance provider in case of an incident. Regularly review and update the plan to reflect changes in your smart contract system and evolving security threats.

There are also other types of coverage such as: Protocol Cover, Yield Token Cover, Stablecoin Depeg Cover, Custody Cover. Some of them do not have to be a cost on your side. You can consider partnership and provide favorable conditions to your users.

RESOURCES:

[How to choose DeFi Cover](#)

[How to compare DeFi Cover providers](#)

Formal Verification

DESCRIPTION:

Formal Verification is like bringing a math genius to analyze your smart contract code. It uses mathematical methods to prove that your code is rock solid, free from vulnerabilities and errors. It's like having an ironclad guarantee that your smart contract will work as intended, no funny business. However, as others it may not cover all possible scenarios, as it relies on formal models and assumptions that may not capture every real-world situation.

GOAL:

Gain confidence in excluding specific risks

PROS:

- Provides a high level of confidence
- Systematic and thorough analysis of the smart contract code

CONS:

- Expensive
- If you have poor documentation, formal verification can make sure that your code and documentation have the same bugs ;)
- Relies on formal models and assumptions, which may not fully capture all real-world scenarios and edge cases
- Requires an expert to prepare the correct specification

WHEN:

- During the development (building the specification)
- Before the release (running the verification)
- Significant changes in the code (building the specification & running the verification)

EXAMPLE PROVIDERS:

- Certora
- Runtime Verification

EXPECTED COST:

- Depends very much on the provider and their business model (fixed price, pay as you go)
- Usually priced based on:
 - amount of code
 - needed expert to build the specification
- \$\$\$

ACTIONS TO IMPLEMENT:

- Select a Specialized Formal Verification Provider:** Choose a trusted third-party provider with a solid track record in Formal Verification for smart contracts. Look for experts who understand your programming language and have a proven ability to deliver reliable results.
- Provide Access to Code and Documentation:** Share the necessary codebase and relevant documentation with the third-party provider. Give them the resources they need to conduct a thorough formal analysis of your smart contract.
- Give yourself time for invariants:** The effect depends on how well-thought-out the models and invariants are. It's a structure you build and then use, so spend enough time on it.
- Collaborate Actively:** Engage in regular, open communication with the provider. Be responsive to their inquiries, provide clarifications when needed, and work together to ensure a comprehensive verification process.
- Iterate and adapt:** Focus on areas where there are a lot of vulnerabilities (what causes them and how to reduce them) and no vulnerabilities (has something been missed?)
- Implement Verified Recommendations:** Take the verification findings seriously. Address any vulnerabilities or issues identified by the

third-party provider promptly and effectively. Make necessary improvements to your smart contract code based on their recommendations.

RESOURCES:

[FORMAL VERIFICATION OF SMART CONTRACTS](#)

[Certora Technology White Paper](#)

Smart Contract Monitoring

DESCRIPTION:

Smart contract monitoring tools are like vigilant watchdogs for your blockchain-based contracts. They constantly watch and assess how your contracts are behaving and flag any unusual activities or security risks in real-time. By using these tools, you can quickly spot and respond to potential issues, ensuring the security and reliability of your smart contracts.

GOAL:

Post-deployment threat detection

PROS:

- Possibility of recovery after a mistake
- Reducing the need for manual oversight and enabling faster response to security incidents.

CONS:

- A high number of false positives can decrease security awareness by fatiguing your team.
- Continuous monitoring can consume a lot of resources.

WHEN:

- After deployment.

The most important things to understand:

1. CryptoTwitter is not monitoring.
2. You must have monitoring.

INTERNAL MONITORING:

- **Gearbox risk framework** is a project whose extensive internal monitoring can be a model and we encourage you to familiarize yourself with their Risk Framework dashboards and transparent approach to security.
 - <https://risk.gearbox.foundation/events>

EXAMPLE PROVIDERS:

- BlockSec Phalcon Block (<https://phalcon.xyz/block>)
The Phalcon Block pre-run transactions from the mempool and monitor them with detectors. If a transaction is marked as malicious, the team is alerted and the transaction blocked.
- Guardrail (<https://www.guardrail.ai/>)
Continuously analyzes interactions for threats across the entire stack using artificial intelligence. You create your own detection rules specific to your project using ready-made and customizable templates.
 - Supported networks:
 - Ethereum mainnet
 - BNB Chain
 - Polygon mainnet
 - Arbitrum
 - Avalanche
 - Fantom
 - Optimism
 - Gnosis
 - Boba
- Cyvers (<https://cyvers.ai/platform>)
AI-based platform that identifies patterns and anomalies across blockchains in real-time for proactive mitigation.
 - Small project \$500 per month
 - Medium project \$1000-3000 per month
 - Big project \$4000+ per month
- Dedaub Watchdog

Automated deep static analysis of contract code combined with dynamic protocol monitoring (all interacting/newly deployed contracts, current chain state, past and current transactions) and statistical learning of code patterns across all contracts ever deployed on EVM networks.

- Monitoring \$1000 per month
- Together with static analysis \$1000-2000 per month
- Forta (<https://docs.forta.network/en/latest/getting-started/>)

The Forta Network tracks blockchain actions live, spotting security risks and notable events through many detection bots, created by a Web3-dev community and security pros.

 - The General Plan
 - costs 250 FORT per month
 - provide subscribers with access to all bots except those designated as Premium Feeds
 - The Premium Feeds
 - in order for a feed to be included as a Premium Feed, they must undergo approval by the Forta Governance Council or request a community Snapshot vote
 - individually priced by their developers in either USDC or FORT
 - available detectors examples are: Scam, Spam and Rug Pull
- Lossless Aegis (<https://docs.lossless.io/aegis>)

The AI-powered smart contract threat monitoring system. Aegis continuously scans all mined block transactions and employs predictive analytics to identify malicious on-chain activity. Suspicious transactions and the associated addresses are flagged based on severity. Project teams are forewarned of potential threats via real-time alerts across multiple channels (email, slack, telegram, sms, webhooks).

 - Supported networks:
 - Ethereum mainnet
 - BNB Chain
 - Polygon mainnet
 - Avalanche
 - Fantom

- Harmony
- Elysium
- Pessimistic Spotter (<https://spotter.pessimistic.io/>)
- Hypernative (<https://www.hypernative.io/>)
- Hexagate (<https://www.hexagate.com/real-time-protection>)
- Ironblocks
(<https://docs.ironblocks.com/security-suite-docs/executive-summary>)

ACTIONS TO IMPLEMENT:

- Tool Selection:** Start by researching and selecting a reliable smart contract monitoring tool that aligns with your project's blockchain platform and security requirements. Ensure the tool provides real-time monitoring, security alerts, and comprehensive contract analysis features.
- Integration Planning:** Develop a clear plan for integrating the selected monitoring tool into your smart contract project. This plan should outline the deployment process, including any necessary configurations or customizations to ensure seamless compatibility with your contracts.
- Continuous Monitoring:** Implement continuous monitoring of your smart contracts. Configure the tool to track contract activity, transactions, and potential vulnerabilities in real time. Set up alerts and notifications to promptly flag any unusual or suspicious behavior.
- Incident Response Protocols:** Establish well-defined incident response protocols that specify how the team should react to alerts or identified security threats. Ensure that team members know their roles and responsibilities in addressing and mitigating contract-related issues.
- Regular Assessment and Optimization:** Periodically review the effectiveness of the monitoring tool and your monitoring strategy. Adjust configurations, update alert thresholds, and expand monitoring coverage as needed to adapt to changing security risks and the evolving smart contract landscape.

READ MORE:

[DSS 2023 Monitoring Panel](#)

[DSS 2023 Monitoring & Incident Response](#)

[DSS 2023 Proactive threat prevention](#)

[CIRCUIT BREAKER EIP-7265](#)

[Pessimism, an open source monitoring system designed to enhance the security of Base](#)

[Spotter Overview](#)

“ If you want to take your project's security to the next level, get help from people who do it every day “

**Damian Rusinek
Co-Founder of Composable Security**

INCREASING EFFICIENCY

In this chapter, we give you tools and tactics that will help you get started quickly and operate efficiently.

Explore templates and implement ready-made tools into your security arsenal.

Audit Readiness Checklist

DESCRIPTION:

Optimize the time and costs of a security audit for your smart contract code. By following this checklist, you can proactively address potential vulnerabilities and ensure your code is well-prepared, saving valuable time and reducing unnecessary back-and-forth with auditors. It streamlines the audit process, enabling you to achieve a thorough and cost-effective assessment while maximizing the chances of a successful outcome.

Composable Security - quick audit readiness checklist			
ID	TASK	RESPONSIBLE PERSON	STATUS
1. Code Clarity			
1.1	I have deleted unused code snippets and old comments.		
1.2	I have added at least 1-2 sentences describing the purpose of each of the smart contracts.		
1.3	I have described each variable and function.		
1.4	I followed a code style that was consistent throughout the codebase.		
1.5	I made sure that the variables and functions were named in a way that corresponds to their purpose and that they are easy to understand.		
2. Documentation			
2.1	I described the roles in the project and what they should have access to.		
2.2	I have described the main user's business flow in detail.		
2.3	I have created high-level diagrams of the protocol.		
2.4	I have updated the white paper.		
2.5	I briefly described the plans for the potential expansion of the project.		

3. Communications			
3.1	I chose a channel through which we will communicate and invited the required people to it.		
3.2	I have indicated the person who will be the main contact and to whom questions should be addressed.		
4. Materials for auditors			
4.1	I created a separate GitHub branch for auditing with files from defined scope (and froze the code).		
4.2	I provided reports from previous audits.		
4.3	I sent the auditors the documentation of our project (e.g. white paper, architecture diagram, flow diagrams).		
5. Automated tools			
5.1	I ran the slither with default settings and handled the reported bugs.		
5.2	I launched solidity-coverage to confirm excellent coverage.		
6. Final checks			
6.1	I compiled the code and ran tests from the audit branch in a fresh environment.		

Due Diligence

DESCRIPTION:

Choosing the right smart contract audit provider is crucial, so take the time to find a company that matches your needs and can demonstrate a proven track record of successful audits.

Experience and Reputation

- Which auditors will review our code and what is their experience?
- Can you provide references from past clients?
- Can we review some samples of your previous audit reports?
- Have any of your audited smart contracts been compromised? If so, please provide details about the breach and the steps you took afterwards.

Approach and Methodology

- Can you describe your approach to auditing smart contracts?
- Do you use manual auditing, automated tools, or a combination of both?
- What types of vulnerabilities do you focus on?
- How do you keep up with the latest vulnerabilities and attack vectors in the smart contract landscape?

Scope and Timing

- How long does an average audit take?
- What factors might extend the audit timeline?
- What is included in your standard audit scope? Can the scope be customized to our needs?
- How do you handle urgent audit requests?

After Audit Support

- What kind of post-audit support do you provide?
- If a vulnerability is discovered after the audit, how will you handle it?
- Do you provide a re-audit after the vulnerabilities are fixed? If so, is there an additional cost?

Pricing and Contract

- How do you determine pricing for the audits?
- Do you offer any guarantees, post-audit coverage or warranties?

Industry Knowledge and Thought Leadership

- What contributions has your team made to the smart contract auditing industry?
- Are any of your team members actively involved in the blockchain community, such as contributing to open-source projects, speaking at conferences, or publishing research?

**You can find our answers on our website.*

Maximizing the Use of Audit Reports

DESCRIPTION:

Maximizing the use of audit reports is about treating them as gold mines of valuable information. It's about extracting insights, leveraging the recommendations, and integrating them into your knowledge base. By embracing the audit report as a valuable resource, you can strengthen your smart contract security practices, continuously improve your code, and build a robust foundation for future development endeavors.

ACTIONS TO IMPLEMENT:

- See Further:** Mine the audit report for gold nuggets of knowledge. Look for patterns, similarities, and lessons that can be applied not only to your current project but also to ongoing initiatives. Share the detected issues and solutions with other teams to prevent similar pitfalls.
- Fortify Unit Tests:** Strengthen your unit tests to cover the new threats and vulnerabilities revealed during the audit. Let the findings guide you in expanding your test suite, ensuring it encompasses the discovered weaknesses and safeguards against future risks.
- Foster Internal Discussions and Learning:** Spark internal discussions about the bugs and vulnerabilities found during the audit. Create a culture of shared learning by encouraging your team to dissect the issues, learn from them collectively, and incorporate the insights into peer reviews. By embracing a collaborative approach, you build a stronger foundation for secure development. Don't blame anyone for mistakes if you want the team to grow!
- Leverage Different Perspectives:** Embrace diverse perspectives by engaging in cross-checks. Capitalize on alternative approaches, utilize complementary security tools, or involve external auditors to validate and corroborate the vulnerabilities and recommendations highlighted in the audit report. All the largest banks, despite constant cooperation, rotate

auditing companies. This multi-faceted assessment enhances the thoroughness and reliability of your security measures.

- **Document Lessons Learned:** Capture the insights and knowledge gained from the audit process. Document the vulnerabilities, their resolutions, and the lessons learned to build a comprehensive knowledge base. This knowledge will guide future development and help prevent similar issues from recurring.

Security Patterns to Enforce

Checks-Effects-Interactions (CEI):

This security pattern focuses thoroughly on validating and verifying input, and carefully managing interactions within your smart contract code. It helps minimizing the risk of malicious or unintended behavior, especially unsecure external calls, reentrancy.

- When designing function logic, plan the elements in the following order:
//CHECKS - if, require
//EFFECTS - balance[msg.sender]++
//INTERACTIONS - call.value()

Principle of Least Privilege (POLP):

The POLP security pattern emphasizes granting minimal access and privileges to entities within the smart contract ecosystem. By following this principle, you limit the exposure of sensitive functions and data, reducing the potential attack surface and preventing unauthorized entities from accessing critical components.

- Existing roles in the system should only have access to the functions that are necessary for them.

Core Invariant and Others (CIAO):

CIAO highlights the importance of identifying and preserving core invariants, which are fundamental properties that must remain true throughout the execution of your smart contract. By focusing on core invariants, you ensure the stability, consistency, and reliability of your smart contract system. Additionally, it encourages identifying and protecting other critical properties to maintain the overall security and integrity of your project.

- Define core invariant and additional ones if needed. Both in unit tests and when calling functions that change the state, check if the invariant has not been broken.

Zero Trust (ZT):

The ZT security pattern challenges the traditional perimeter-based security model and adopts an approach of continuous verification and authentication for all entities and actions within the smart contract ecosystem. It emphasizes the need to validate every request and transaction, regardless of the source or location, fostering a proactive and robust security posture that assumes potential threats exist both inside and outside the system.

- Do not trust external smart contracts. Think about what will happen when one of your internal components is compromised. Use several sources and check the data you receive. Monitor contracts that can be updated.

Pull Over Push (POP):

POP promotes a security pattern that favors a "pull" approach rather than a "push" approach when it comes to data and actions within a smart contract system. Instead of allowing external entities to directly modify or manipulate data, the system design encourages external entities to request and retrieve data through well-defined, controlled mechanisms.

- Don't send funds to users, let them withdraw funds themselves,

Automatic Tools

DESCRIPTION:

Automatic security tools play a pivotal role in fortifying the defenses of Web3 projects. As the decentralized web evolves, the complexity and potential vulnerabilities of smart contracts and dApps increase. These tools offer smart contract analysis, detecting vulnerabilities through both static and dynamic analysis.

The ease of integrating these tools into CI/CD workflows (e.g. via GitHub Actions) means that even as projects scale, security remains a consistent and automated priority, ensuring the trust and safety that Web3 promises its users. By integrating these tools into the Continuous Integration/Continuous Deployment (CI/CD) pipeline, developers can ensure that security checks are conducted automatically with every code change, making the process seamless and efficient. This not only reduces the risk of deploying vulnerable code, but also fosters a proactive security culture.

GOAL:

Start using automatic tools within your development process.

Tools:

We are firstly going to cover two tools that cannot be easily integrated into CI/CD, but are very helpful for developers to gather insights about the contracts, especially when the codebase is large.

Solidity metrics

Solidity metrics brings an overall summary of the contracts, including such numbers as:

- number of lines,
- number of functions with their types (payable, external, etc.),
- complexity and risk profiles,
- other delivered by the next tool described below (Surya).

Its biggest advantage is that it has a VSCode extension and presents all details in human readable format (HTML). This is particularly useful when you want to roughly estimate the cost of an upcoming audit, as one of the important factors is the size of contracts in scope.

Installation

Solidity metrics can be installed as node package:

```
npm install solidity-code-metrics
```

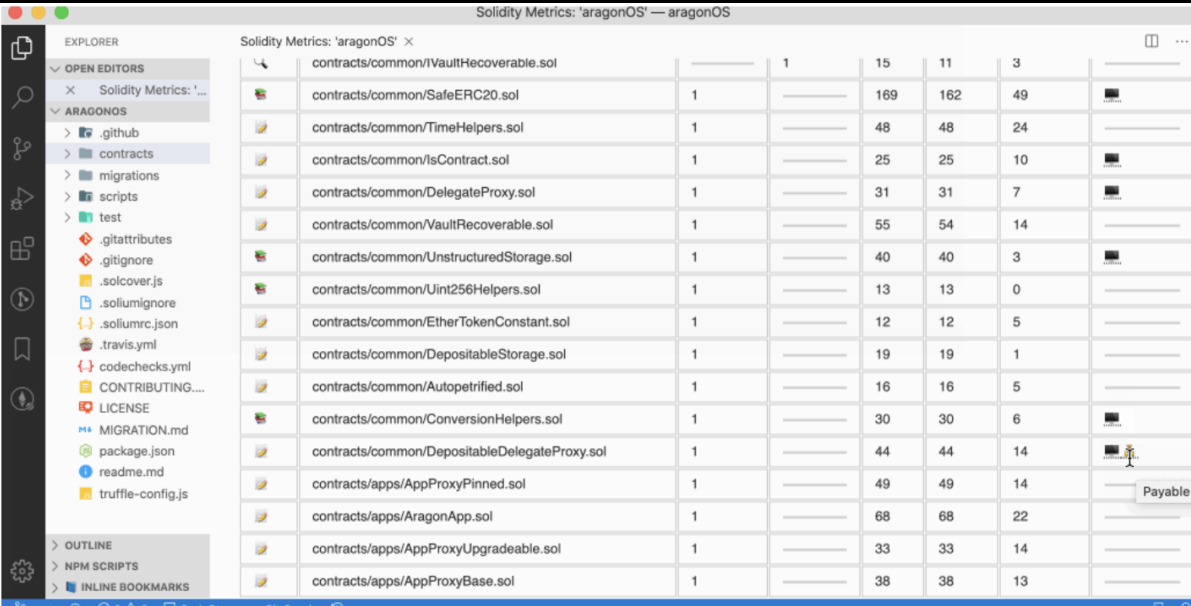
Additionally, it has a VSCode extension that can be installed from the VSCode extensions list. This is the best option as you have it integrated in your IDE (if your IDE is VSCode).

Usage

One way is to use it as binary in the terminal and save report to HTML file:

```
solidity-code-metrics myfile.sol --html > metrics.html
```

However, an easier and recommended way is to use it as a VSCode extension.



Contract Name	Lines	Other Metrics
contracts/common/VaultRecoverable.sol	1	15, 11, 3
contracts/common/SafeERC20.sol	1	169, 162, 49
contracts/common/TimeHelpers.sol	1	48, 48, 24
contracts/common/IsContract.sol	1	25, 25, 10
contracts/common/DelegateProxy.sol	1	31, 31, 7
contracts/common/VaultRecoverable.sol	1	55, 54, 14
contracts/common/UnstructuredStorage.sol	1	40, 40, 3
contracts/common/Uint256Helpers.sol	1	13, 13, 0
contracts/common/EtherTokenConstant.sol	1	12, 12, 5
contracts/common/DepositableStorage.sol	1	19, 19, 1
contracts/common/Autopetrified.sol	1	16, 16, 5
contracts/common/ConversionHelpers.sol	1	30, 30, 6
contracts/common/DepositableDelegateProxy.sol	1	44, 44, 14
contracts/apps/AppProxyPinned.sol	1	49, 49, 14
contracts/apps/AragonApp.sol	1	68, 68, 22
contracts/apps/AppProxyUpgradeable.sol	1	33, 33, 14
contracts/apps/AppProxyBase.sol	1	38, 38, 13

Link

<https://github.com/Consensys/solidity-metrics>

<https://github.com/Consensys/vscode-solidity-metrics>

Surya

Surya provides developers with a visual representation of the structure and flow of their Solidity contracts, aiding in understanding and analyzing the codebase. With functionalities like generating a function call graph or showing inheritance chains, Surya simplifies the process of reviewing and auditing complex smart contracts.

Surya provides the following outputs:

- contracts' inheritance graph,
- control flow graph,
- treefied function call trace,
- and most importantly - a summary of the contracts and methods in the files provided as output from the *describe* command.

Installation

Surya can be installed as a node package. The example below shows how to install it globally so that you will not have to install it in each project.

```
npm install -g surya
```

Surya does not have a VSCode extension, but afore-mentioned tool - Solidity metrics - presents most of its results.

Usage

Depending on the output you would like to get, you must use a specific subcommand. For example, if you want to see the contracts summary, use the *describe* command:

```
surya describe contracts/*.sol
```

The output:


```

+ ExchangeRateProvider (usingOraclize)
- [Pub] <Constructor> #
- [Ext] setCallbackGasPrice #
- [Pub] sendQuery ($)
- [Prv] setQueryId #
- [Pub] __callback #
- [Pub] selfDestruct #
- [Pub] <Fallback> ($)

```

```

($) = payable function
# = non-constant function

```

The other option to get a full report in Markdown format.

```
surya mdreport report_outfile.md contracts/*.sol
```

However, the same result you would get with Solidity metrics within VSCode.

Link

<https://github.com/Consensys/surya>

Solhint

Solhint is a linter tool specifically designed for Solidity. Linters are essential in software development as they analyze code to detect potential errors, enforce coding standards, and ensure consistent styling.

Solhint provides both security and style guide validations, helping developers identify and rectify potential vulnerabilities, anti-patterns, and breaches of coding conventions in their Solidity code. By integrating Solhint into their development workflow, Solidity developers can enhance the security, readability, and maintainability of their smart contracts, ensuring they adhere to best practices and reduce the risk of costly mistakes in the decentralized environment.

Installation

```
npm install -g solhint
```

Usage

First initialize a configuration file, if you don't have one:

```
solhint --init
```

This will create a `.solhint.json` file with the default rules enabled. Then run `solhint` with one or more files as arguments. For example, to lint all files inside `contracts` directory, you can do:

```
solhint 'contracts/**/*.sol'
```

You can also **integrate solhint with the following IDEs** (see the GitHub link for more details):

- Sublime Text 3
- Atom
- Vim
- JetBrains IDEA, WebStorm, CLion, etc.
- VS Code: Solidity by Juan Blanco
- VS Code: Solidity Language Support by CodeChain.io

It is also recommended to create a **GitHub action** that will execute `solhint` on each pull request commit. That would be the last layer of detection before the code lands on the main branch.

Link

<https://github.com/protofire/solhint>

Slither

Slither is a static analysis tool, designed to detect vulnerabilities, coding mistakes, and inefficiencies in Solidity code. Slither examines the contract's abstract syntax tree and control flow graphs to provide precise insights. With

its comprehensive set of detectors, it can identify a wide range of issues, from reentrancy attacks to incorrect visibility settings.

By integrating Slither into the development workflow, Ethereum developers can proactively address potential security threats, ensuring that smart contracts deployed on the blockchain are robust, efficient, and secure. Its ability to produce actionable recommendations makes Slither an invaluable asset in the toolkit of anyone serious about smart contract security.

Installation

Slither was written in Python and can be installed using *pip* command:

```
pip3 install slither-analyzer
```

There are other options to install and use Slither, e.g. using Docker or build from scratch.

Usage

Run Slither on a Hardhat/Foundry/Dapp/Brownie application:

```
slither .
```

This is the preferred option if your project has dependencies as Slither relies on the underlying compilation framework to compile source code.

However, you can run Slither on a single file (or multiple files) that does not import dependencies:

```
slither StandaloneContract.sol
```

Slither will analyze the contract and provide an output of potential vulnerabilities, coding mistakes, or inefficiencies. The output might look something like this:

```
INFO:Detectors:  
Factory.createToken(TokenInput,address[],uint256[]).market  
(contracts/Factory.sol#50) is a local variable never  
initialized
```

```
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation
#uninitialized-local-variables
```

```
INFO:Detectors:
Token.constructor(string,string,uint256).name
(contracts/Token.sol#11) shadows:
  - ERC20.name()
(node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#61-63) (function)
  - IERC20Metadata.name()
(node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#16) (function)
```

This is a basic example, and the actual output will vary based on the contract's complexity and the vulnerabilities detected.

Besides the detectors provided by the Slither's core team, there are initiatives to create additional ones, like the **slitherin** (see links section), which is an easy to install set of detectors that will be automatically added to Slither.

Moreover, you can use the **VSCode extension** for Slither that will run Slither directly from VSCode, list findings in GUI and open the specific file and line where the issue has been found, put simply - it makes using Slither very easy and comfortable.

What is more, you can use existing **GitHub Action** (see links section) that will execute Slither on each pull request commit. That will automatically find the low-hanging fruits. However, be aware of some false positives that you will have to take care of manually.

Link

<https://github.com/crytic/slither>

<https://github.com/pessimistic-io/slitherin>

<https://marketplace.visualstudio.com/items?itemName=trailofbits.slither-vscode>

[e](#)

<https://github.com/marketplace/actions/slither-action>

Semgrep

Semgrep is a versatile, open-source static analysis tool that enables developers to write and run custom code patterns on their codebase. Unlike traditional linters or static analysis tools that come with predefined rules, Semgrep allows users to define their own rules using the same syntax as the code they're analyzing. This makes it particularly powerful for identifying complex code patterns, anti-patterns, or security vulnerabilities specific to a project or organization.

With support for multiple languages, including Solidity and Python, Semgrep can be integrated into CI/CD pipelines, ensuring that code reviews are automated and potential issues are flagged before they make it to production.

Note: Semgrep has a cloud platform that can be used on-line and a paid version that performs more in-depth analysis, but here we will cover the [free version](#) that you can run locally.

Installation

Semgrep can be installed locally using multiple package managers:

```
# For macOS
brew install semgrep

# For Ubuntu/WSL/Linux/macOS
python3 -m pip install semgrep
```

You can also use Docker to run semgrep locally without installation:

```
# To try Semgrep without installation run via Docker
docker run --rm -v "${PWD}:/src" returntocorp/semgrep semgrep
```

Usage

Semgrep has a registry that includes different packs of rules. In order to use the built-in solidity rules, run the following command:

```
semgrep scan --config r/solidity --no-git-ignore contracts/*
```

Here is an output example:

Findings:

```
test.sol

solidity.best-practice.use-abi-encodecall-instead-of-encodewithselector.use-abi-encodecall-instead-of-encodewithselector
    To guarantee arguments type safety it is recommended to use `abi.encodeCall` instead of `abi.encodeWithSelector`.
    Details: https://sg.run/9K1y

12 | (success, data) =
collection.call(abi.encodeWithSelector(ITest.divide.selector,
(a, b)));
: | -----
30 | (bool status, ) = transferManager.call(
31 |     abi.encodeWithSelector(
32 |
managerSelectorOfAssetType[assetType].selector,
33 |     collection,
34 |     sender,
35 |     recipient,
36 |     itemIds,
37 |     amounts
38 |     )
39 | );
```

Ran 49 rules on 1 file: 2 findings.

You can also **use other, not official, rules** provided by external teams. Here is an example how to use cloned repo by Decurity (see links below):

```
semgrep scan --config
<PATH-TO-DIR-WITH-REPO>/semgrep-smart-contracts/solidity/*.ya
ml --no-git-ignore contracts/*
```

We also encourage you to **write your own, internal rules** based on the bugs you find internally.

Last, but not least, it is again recommended to create a **GitHub action** that will execute semgrep with predefined rules (possibly multiple runs) on each pull request commit.

Link

<https://semgrep.dev/>

<https://github.com/returntocorp/semgrep>

<https://semgrep.dev/r>

<https://github.com/Decurity/semgrep-smart-contracts>

<https://github.com/Decurity/compound-semgrep-rules>

<https://github.com/Decurity/semgrep-rules>

Echidna

Echidna is a sophisticated fuzzing tool tailored for testing Ethereum smart contracts. It employs property-based testing to automatically generate inputs that probe contracts for vulnerabilities and unexpected behaviors.

Instead of writing traditional unit tests, developers define properties that the contract should always uphold, and Echidna attempts to find counterexamples that violate these properties. This approach is particularly effective for discovering edge cases that manual testing or static analysis might overlook.

By integrating Echidna into the development and auditing workflow, Ethereum developers can gain increased confidence in the robustness and security of their smart contracts, ensuring that they can withstand a myriad of inputs and scenarios once deployed on the blockchain.

Installation

Prerequisites: Before you install Echidna, make sure you have Slither installed. It is described above.

Echidna can be either downloaded as statically linked binaries for Linux or MacOS from the releases page or installed using Homebrew:

```
brew install echidna
```

You can also use Docker to run echidna without installation:

```
docker run --rm -it -v `pwd`:/src  
ghcr.io/crytic/echidna/echidna
```

Usage

The core Echidna functionality is an executable called `echidna`, which takes a contract and a list of invariants (properties that should always remain true) as input.

Invariants are expressed as Solidity functions with names that begin with `echidna_`, have no arguments, and return a boolean. For example, if you have some balance variable that should never go below 20, you can write an extra function in your contract like this one:

```
function echidna_check_balance() public returns (bool) {
    return(balance >= 20);
}
```

Here is a full example contract (available [here](#)):

```
contract Test {
    event Flag(bool);

    bool private flag0 = true;
    bool private flag1 = true;

    function set0(int val) public returns (bool){
        if (val % 100 == 0)
            flag0 = false;
    }

    function set1(int val) public returns (bool){
        if (val % 10 == 0 && !flag0)
            flag1 = false;
    }

    function echidna_alwaystrue() public returns (bool){
        return(true);
    }

    function echidna_revert_always() public returns (bool){
        revert();
    }

    function echidna_sometimesfalse() public returns (bool){
        emit Flag(flag0);
        emit Flag(flag1);
        return(flag1);
    }
}
```



```
}
```

To run it, you should execute:

```
echidna tests/solidity/basic/flags.sol
```

Here is the output:

[Echidna 2.2.1]		
Time elapsed: 16s Workers: 0/1 Seed: 7749810078530502207 Calls/s: 2942 Total calls: 50014/50000	Unique instructions: 345 Unique codehashes: 1 Corpus size: 1 seqs New coverage: 16s ago	Chain ID: - Fetched contracts: 0/0 Fetched slots: 0/0
Tests (3)		
echidna_sometimesfalse: FAILED! with ReturnFalse ^		
Call sequence: 1.set0(0) 2.set1(0) Event sequence: Flag(false) from: 0xa329c0648769a73afac7f9381e08fb43d8ea72 Flag(false) from: 0xa329c0648769a73afac7f9381e08fb43d8ea72		
echidna_revert_always: passing		
echidna_alwaystrue: passing		
Log (3)		
[2023-09-26 13:46:43.48] [Worker 0] Test limit reached. Stopping. [2023-09-26 13:46:26.92] [Worker 0] New coverage: 345 instr, 1 contracts, 1 seqs in corpus [2023-09-26 13:46:26.81] [Worker 0] Test echidna_sometimesfalse falsified! v		
Campaign complete, C-c or esc to exit		

The echidna fuzzing tests are the part of all project's tests, next to unit tests. Therefore, whenever you run tests locally, you should include echidna tests.

Link

<https://github.com/crytic/echidna>

<https://github.com/crytic/echidna/blob/master/tests/solidity/basic/flags.sol>

<https://github.com/Uniswap/v3-core/tree/main/contracts/test>

Foundry fuzzer

Foundry is a smart contract development toolchain, which manages your dependencies, compiles your project, runs tests, deploys, and lets you interact with the chain from the command-line and via Solidity scripts.

In this section we will focus on Foundry tests, specifically fuzzing tests as it was getting more and more popular recently.

Installation

Precompiled binaries are available from the GitHub releases page. However, these are better managed by using Foundryup, which is the Foundry toolchain installer.

Open your terminal and run the following command:

```
curl -L https://foundry.paradigm.xyz | bash
```

This will install Foundryup, then simply follow the instructions on-screen, which will make the *foundryup* command available in your CLI.

Running *foundryup* by itself will install the latest (nightly) precompiled binaries: *forge*, *cast*, *anvil*, and *chisel*.

```
foundryup
```

Usage

Even though Foundry provides a wide variety of development tools, here we are focusing on the tests, specifically fuzzing. The simple fuzzing tests are very similar to typical unit tests in Foundry.

Here is a simple unit test in Foundry that checks whether the balances are correct before and after withdrawing from the *safe* contract.

```
function test_Withdraw() public {
    payable(address(safe)).transfer(1 ether);
    uint256 preBalance = address(this).balance;
    safe.withdraw();
    uint256 postBalance = address(this).balance;
    assertEq(preBalance + 1 ether, postBalance);
}
```

Here is the same test but in the form of a fuzzing test.

```
function testFuzz_Withdraw(uint256 amount) public {
```

```
vm.assume(amount > 0.1 ether);
payable(address(safe)).transfer(amount);
uint256 preBalance = address(this).balance;
safe.withdraw();
uint256 postBalance = address(this).balance;
assertEq(preBalance + amount, postBalance);
}
```

There are 3 differences:

- name of the function,
- added parameter (this is fuzzed by Foundry),
- added a condition that will accept only those values for *amount* that are greater than *0.1 Ether*.

To run both of these kinds of tests, you must run the following command (assuming the tests are in *SimpleTest.t.sol* file):

```
forge test --fuzz-runs 10000 -vv --mp test/SimpleTest.t.sol
```

The above example is a simple fuzzing test, however Foundry also supports two other types, that is invariant fuzzing and differential fuzzing. These are worth checking in the Foundry book.

Invariant tests are simple functions with names starting with *invariant_* that always return boolean and are asserted after each function call is made. Here is a simple conditional (checked only if the *protocolCondition* is *false*) invariant:

```
function invariant_example() external {
    if (protocolCondition) return;

    assertEq(val1, val2);
}
```

Differential tests compare the result of a tested function with another - assumed to be correct - function. Here is a simple example for checking custom Merkle tree implementation against OpenZeppelin's implementation:

```
import
"openzeppelin-contracts/contracts/utils/cryptography/MerklePr
oof.sol";
//...
```

```

function testCompatibilityOpenZeppelinProver(bytes32[] memory
_data, uint256 node) public {
    vm.assume(_data.length > 1);
    vm.assume(node < _data.length);
    bytes32 root = m.getRoot(_data);
    bytes32[] memory proof = m.getProof(_data, node);
    bytes32 valueToProve = _data[node];
    bool murkyVerified = m.verifyProof(root, proof,
valueToProve);
    bool ozVerified = MerkleProof.verify(proof, root,
valueToProve);
    assertTrue(murkyVerified == ozVerified);
}

```

Similarly to the previous section, Foundry fuzzing tests are the part of all project's tests, next to unit tests. Therefore, whenever you run tests locally, you should include fuzzing tests.

Link

<https://github.com/foundry-rs/foundry>

<https://book.getfoundry.sh/>

<https://www.rareskills.io/post/foundry-testing-solidity>

<https://www.rareskills.io/post/invariant-testing-solidity>

ACTIONS TO IMPLEMENT:

- Learn & try tools:** Go through the descriptions of all tools and try them in your project to find out the most comfortable for you.
- Share knowledge:** Add the descriptions of tools to the internal Knowledge Base and ask new developers to get familiar with that.
- Add tools to CI/CD:** Use GitHub Actions for the tools that have them mentioned in the description.
- Extend your Knowledge Base:** Build your own rules and detectors used by the automatic tools.
- Estimate costs:** Use the metric tools to get familiar with the codebase size and complexity. That will help you estimate the potential cost of security reviews.

- **Focus on invariants:** Identify invariants when designing the protocol. They are gonna be great inputs for invariant fuzz testing.

Common Smart Contract Vulnerabilities

DESCRIPTION:

Smart contracts, pivotal in the Web3 landscape, are susceptible to several common vulnerabilities. Much like the Pareto principle, where 80% of effects come from 20% of causes, diligently addressing these vulnerabilities can mitigate a large portion of potential threats. As smart contracts become more prevalent, awareness and mitigation of these vulnerabilities are paramount.

The goal of this chapter is to learn how to avoid common vulnerabilities.

Price oracle manipulation

A price manipulation attack occurs when an actor intentionally alters the price of an asset on a decentralized exchange or oracle to benefit from trades or contracts that rely on that price. By exploiting vulnerabilities or using large trades, the attacker can skew the price temporarily, execute favorable trades or trigger specific contract conditions, and then return the price to its normal range, securing a profit in the process. Such attacks underscore the importance of robust and tamper-proof price oracles and liquidity in decentralized platforms.

Ironically, this is a relatively rarely described vulnerability even though it is the most common and appearing on everyone's lips.

The causes of the price manipulation might be the following:

1. **Spot Price:** When the oracle calculates the price based on current values (e.g., reserves) that can be easily changed (e.g., using large swap, like in case of Uniswap), it is very easy to imbalance those values and shift the price. The effect can be maximized with flash loans.

2. Shallow Liquidity: When a decentralized exchange (DEX) or a liquidity pool has low liquidity, it's easier for an attacker to move the price with a relatively small amount of capital.
3. Reliance on Single Oracle: If a smart contract or a platform relies on a single price oracle, it becomes a point of vulnerability. An attacker can manipulate or spoof this oracle to feed incorrect price data.
4. Delayed Oracle Updates: Contracts that rely on oracles which update prices infrequently can be exploited. If there's a significant delay, the oracle might be out of sync with real market prices, allowing attackers to capitalize on the discrepancy.
5. Centralized Points of Control: Some "decentralized" platforms might still have centralized components or admin keys that can influence prices. If these are compromised, prices can be manipulated.

Example

Let's consider a scenario where a smart contract determines the price of an asset based on the ratio of assets in a liquidity pool, similar to how automated market makers (AMMs) like Uniswap work. In this example, the price is derived from the ratio of two assets in a liquidity pool, and an attacker can manipulate the price by adding or removing assets from the pool.

Warning: This example has other vulnerabilities and is just a simplified version of the price manipulable contracts.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleAMM {
    uint256 public tokenA; // e.g., ETH
    uint256 public tokenB; // e.g., DAI

    // Users can add liquidity to the pool
    function addLiquidity(uint256 amountA, uint256 amountB)
external {
    tokenA += amountA;
    tokenB += amountB;
}
}
```

```

    // Users can remove liquidity from the pool
    function removeLiquidity(uint256 amountA, uint256
amountB) external {
        require(tokenA >= amountA && tokenB >= amountB, "Not
enough liquidity");
        tokenA -= amountA;
        tokenB -= amountB;
    }

    // Get the price of tokenA in terms of tokenB
    function getPriceOfTokenA() external view returns
(uint256) {
        return tokenB / tokenA; // This is a simplified price
calculation
    }
}

contract VulnerableContract {
    SimpleAMM public amm;

    uint256 public thresholdPrice = 100; // Example threshold
price in terms of tokenB

    constructor(address _amm) {
        amm = SimpleAMM(_amm);
    }

    function executeTrade() external {
        uint256 currentPrice = amm.getPriceOfTokenA();
        if (currentPrice > thresholdPrice) {
            // Execute some logic, e.g., sell tokenA
        }
    }
}

```

An attacker can manipulate the price of *tokenA* by adding or removing large amounts of *tokenA* or *tokenB* to or from the SimpleAMM pool. By skewing the ratio of the assets, they can influence the price returned by *getPriceOfTokenA()* and potentially trigger the *executeTrade* function in the *VulnerableContract* under false pretenses.

In real-world AMMs, the price calculation is more complex and involves additional factors, but the principle remains: the price is influenced by the ratio of assets in the pool, and large trades or liquidity actions can move the price.

Moreover, another common example is when the oracle calculates the prices based on the current balances of the tokens. The attacker can then simply transfer the token to the oracle to skew the ratio.

How to be safe?

Here are some strategies to mitigate such attacks:

1. **Use Time-Weighted Average Prices (TWAP):** Instead of relying on the spot price, which can be easily manipulated with a single large trade, use the average price over a period of time. This makes it more costly and difficult for an attacker to manipulate the price.
2. **Multiple Oracles:** Instead of relying on a single source of truth, use multiple oracles or price feeds and take an average or median. This reduces the risk associated with any single oracle being manipulated.
3. **Slippage Protection:** Implement slippage protection in your smart contracts. This means that a trade will only go through if the price slippage (the difference between the expected price and the price at which the trade is executed) is below a certain threshold.
4. **Volume Limits:** Implement limits on the volume of trades relative to the liquidity pool's size. This can prevent large trades from causing drastic price changes.
5. **Circuit Breakers:** Similar to traditional financial markets, implement mechanisms that can pause trading or certain contract functions if abnormal price movements are detected.
6. **Regular Monitoring and Alerts:** Set up monitoring tools to track abnormal price movements, large trades, or sudden liquidity changes. Immediate alerts can help in taking quick actions.
7. **Decentralized Oracles:** Consider using decentralized oracle networks, which aggregate data from multiple sources, reducing the risk of price manipulation.

Arithmetic errors

Arithmetic bugs are common vulnerabilities in programming, and in the context of web3 and smart contracts, they can have severe financial implications. Here are some of the arithmetic bugs that can occur in web3:

1. **Integer Overflow:** This occurs when an arithmetic operation results in a number larger than the maximum size the variable can hold. In Solidity, if a `uint256` variable reaches its maximum value and you add any positive integer to it, it will wrap around to zero. This is currently checked in Solidity (starting from 0.8 version) and the transaction reverts, but there are cases where this check is not present, e.g., when you cast `uint256` to `uint8` or when you explicitly insert your code in the *unchecked* block.
2. **Integer Underflow:** The opposite of overflow, this happens when an arithmetic operation results in a number smaller than the minimum size the variable can hold. For a `uint256` in Solidity, subtracting from zero will wrap around to its maximum possible value. Similarly to overflows, it is currently checked in Solidity (starting from 0.8 version) but there are cases when it still happens.
3. **Rounding Errors:** When performing division or other operations that result in fractional values, rounding can lead to small discrepancies. Over many transactions, these can accumulate or be exploited. It is particularly important to correctly handle the divisions where the numerator might be lower than the denominator, because the result of that operation would be zero.
4. **Mismatched orders of magnitude:** Discrepancies in the expected scale or unit of a value, often leading to significant errors in calculations or operations. A common source of mismatched orders of magnitude arises from different tokens having different decimal precisions.
5. **Reordering of Operations:** Due to the commutative property of addition and multiplication, the order in which operations are performed can affect the result, especially when combined with other arithmetic bugs. Particularly important is the ordering of multiplication and division, the former should be executed first to keep the precision because the division in Solidity is the integer division (without the remaining).

Example

Suppose you're building a smart contract that accepts both ETH and an ERC-20 token (let's call it TOKEN). ETH has a precision of 18 decimals, which means 1 ETH is represented as 10^{18} wei. On the other hand, TOKEN is designed with only 6 decimals of precision.

Now, imagine a function in the contract that's supposed to swap 1 ETH for an equivalent amount of TOKEN based on a 1:1000 price ratio.

```
pragma solidity ^0.8.0;

interface IERC20 {
    function transferFrom(address sender, address recipient,
uint256 amount) external returns (bool);
    function transfer(address recipient, uint256 amount)
external returns (bool);
}

contract TokenSwap {
    address public owner = msg.sender;
    IERC20 public token;

    constructor(address _tokenAddress) {
        token = IERC20(_tokenAddress);
    }

    function swapEthForToken() external payable {
        require(msg.value == 1 ether, "Send exactly 1 ETH");

        // Expected: Send 1000 TOKENs for 1 ETH
        // But due to mismatched decimals, this will actually
send 0.001 TOKENs
        token.transferFrom(owner, msg.sender, 1000);
    }
}
```

In the above contract, the `swapEthForToken` function expects the user to send exactly 1 ETH. In return, it tries to send 1000 TOKENs. However, due to the mismatch in decimals, it will actually send only 0.001 TOKENs, which is not the intended behavior.

To fix this, the contract should account for the decimal difference:

```
function swapEthForToken() external payable {
    require(msg.value == 1 ether, "Send exactly 1 ETH");

    // Adjust for the decimal difference
    uint256 tokenAmount = 1000 * 10**6; // 10^6 for 6
decimals
    token.transferFrom(owner, msg.sender, tokenAmount);
}
```

How to be safe?

Here are some strategies to mitigate such attacks:

1. **Explicit Checks:** Always check for potential overflows, underflows, and other arithmetic anomalies before performing operations. For instance, before subtracting, check if the value being subtracted is less than or equal to the original value.
2. **Thorough Testing:** Ensure that your smart contract is rigorously tested. Include edge cases, such as maximum and minimum input values, to test for potential overflows and underflows.
3. **Explicit Rounding:** When performing operations that might result in fractional values, decide explicitly how to handle the rounding (e.g., always round down, always round up, or round to the nearest value). Document this choice and ensure it's consistently applied.
4. **Keep Precision:** When doing division operations remember to make sure that any multiplications were executed before and if there is a possibility that the numerator might be lower than the denominator, add multiplication by some power of 10 (e.g., 10^4) and divide by the same value at the end of operations.

Reentrancy

Reentrancy vulnerability arises when external contract calls are allowed to be made before a function completes its execution. In such scenarios, the called contract can call back into the calling contract before the first function call is finished. This recursive behavior can lead to unintended consequences, such as funds being withdrawn multiple times. The most infamous exploitation of this vulnerability was the DAO attack in 2016, where an attacker drained tens of

millions of dollars worth of Ether by repeatedly calling a function before the original call could update the contract's state.

A specific and quite novel version of the reentrancy vulnerability is the read-only reentrancy. In this case the called contract calls back a function that does not change the state - a view function that usually is not protected from reentrancy. The most common case of read-only reentrancy is the Curve pool that protects `add_liquidity` and `remove_liquidity` functions but does not protect `get_virtual_price` view function. See the second example below to understand the consequences.

Example

Here is the simplest and most classic example of a reentrancy bug often demonstrated using a vulnerable Ether "withdraw" function in a smart contract.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VulnerableBank {
    mapping(address => uint256) public balances;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() external {
        uint256 amount = balances[msg.sender];
        require(msg.sender.call{value: amount}(""),
"Withdrawal failed");

        // Update balance only after sending Ether
        balances[msg.sender] = 0;
    }
}
```

In this example, the *withdraw* function first sends Ether to the *msg.sender* and then sets their balance to zero. This is vulnerable to reentrancy because the external contract can call *withdraw* again before the first call has finished, effectively draining more Ether than they should be able to.

On the other hand, the *get_virtual_price* function in some Curve Finance pools is a classic example of a read-only reentrancy vulnerability. This function is intended to provide a view of the virtual price of the pool's assets, but it can be influenced by preceding state-changing operations within the same transaction.

Here's a simplified example to illustrate the vulnerability:

```
pragma solidity ^0.8.0;

interface ICurvePool {
    function remove_liquidity(uint256 _amount, uint256[]
calldata min_amounts) external;
    function get_virtual_price() external view returns
(uint256);
}

contract LendingProtocol {
    ICurvePool public curvePool;
    mapping(address => uint256) public userCollateral;

    constructor(address _curvePool) {
        curvePool = ICurvePool(_curvePool);
    }

    function depositCollateral(uint256 curveLPTokenAmount)
external {
        uint256 collateralValue = curveLPTokenAmount *
curvePool.get_virtual_price();
        userCollateral[msg.sender] += collateralValue;
    }
}

contract Attacker {
    ICurvePool curvePool;
    LendingProtocol lendingProtocol;

    constructor(address _curvePool, address _lendingProtocol)
{
        curvePool = ICurvePool(_curvePool);
        lendingProtocol = LendingProtocol(_lendingProtocol);
    }

    function attack(uint256 amount, uint256[] calldata
min_amounts) external {
        // Remove liquidity from the Curve pool
        curvePool.remove_liquidity(amount, min_amounts);
    }
}
```

```
    }  
  
    function fallback() {  
        // During the removing liquidity from Curve, the  
        lending protocol queries  
        // the get_virtual_price which might be inconsistent  
        due to the ongoing  
        // remove_liquidity transaction  
        lendingProtocol.depositCollateral(100 ether);  
    }  
}
```

In this scenario:

1. The attacker initiates the attack by calling the *attack* function, which removes liquidity from the Curve pool.
2. The Curve pool, when removing liquidity, might have a mechanism to call back to the attacker's contract (this happens when the user withdraws Ether).
3. During this callback (fallback function), the attacker makes the lending protocol query the *get_virtual_price*.
4. Since the Curve pool's *remove_liquidity* transaction hasn't fully settled, the *get_virtual_price* might return an interim or inconsistent value.
5. The attacker exploits this inconsistency to manipulate the perceived value of their collateral in the lending protocol.

How to be safe?

Protecting against reentrancy and read-only reentrancy vulnerabilities requires a combination of coding best practices, design patterns, and external safeguards. Here are some strategies to mitigate these risks:

1. Checks-Effects-Interactions Pattern: Always update the state variables before making external calls. This ensures that even if a reentrancy occurs, the contract's state is consistent. There might be cases where it is not achievable and those should be covered by case-by-case consultations.

2. Reentrancy Guard: Use a mutex or a state variable to lock the contract's state while it's performing critical operations. OpenZeppelin's `ReentrancyGuard` is a popular choice for this.
3. Reviewing External Protocols: Many vulnerabilities arise when two (or more) separate protocols are integrated. You should check whether the other protocol that you integrate with is vulnerable to read-only reentrancy and that risk is present in your protocol as well.

Unsafe external call

Unsafe external calls are a significant security concern in smart contract development, particularly in the Ethereum ecosystem. These vulnerabilities arise when a contract makes calls to external contracts or addresses in an unsafe manner, without adequate checks or protective measures (e.g. lack of checks on the callee and calldata).

The external called contract could be malicious and execute operations that compromise the calling contract's integrity. For example, it could trigger reentrancy attacks, manipulate state variables, or cause the calling contract to get stuck in an indefinite loop.

An unsafe *delegatecall* can be particularly devastating, as it executes the code of another contract within the context of the calling contract, including its storage. If the target contract of the *delegatecall* contains malicious code, it could overwrite crucial storage variables or manipulate the contract logic in the calling contract. In the worst-case scenario, this could lead to the *self-destruction* of the contract or the permanent locking of its funds.

Example

One of many examples could be an abuse of token approvals. Let's consider a simplified example where a user approves a contract that has an unsafe external call. This contract also has a function that calls `transferFrom` on a token contract. An attacker can exploit this to drain tokens from the user's account.


```

pragma solidity ^0.8.0;

contract VulnerableContract {
    ERC20 public token;
    address public owner;

    constructor(address _token) {
        token = ERC20(_token);
        owner = msg.sender;
    }

    function unsafeExternalCall(address to, bytes calldata
data) external {
        require(msg.sender == owner, "Only owner can call");

        // Unsafe external call
        (bool success, ) = to.call(data);
        require(success, "External call failed");
    }

    function withdrawTokens(address to, uint256 amount)
external {
        require(token.transferFrom(msg.sender, to, amount),
"Transfer failed");
    }
}

```

In this example:

1. The *VulnerableContract* has an *unsafeExternalCall* function that makes an external call without any checks on what the external contract might do.
2. Users approve *VulnerableContract* to spend tokens on their behalf using the ERC-20 *approve* function.
3. The attacker calls *unsafeExternalCall*, passing in the address of the token and data that represents a call to *transferFrom* function with the victim's address as *from*, attacker's address as *to* and the amount approved by the victim as *amount*, effectively transferring the approved tokens to the attacker's address.

This example demonstrates how an attacker can exploit unsafe external calls in a contract that users have approved to interact with their tokens.

How to be safe?

Here are some strategies to mitigate this vulnerability:

1. **Validation:** If that's possible, and in most cases it should be, make sure that your contract can call only approved contracts and that the function to be called is explicitly specified.
2. **Trust:** Only interact with well-audited, reputable external contracts. If you must interact with an unknown contract, ensure you have adequate checks and validations.
3. **Least privilege:** If you need to allow calls to untrusted contracts, make sure that the contract that makes that call is not privileged and does not hold any funds.
4. **Sanity Checks:** Before making a delegatecall, perform checks to ensure that the target contract's code is as expected (e.g., check contract bytecode, use a registry of trusted contracts).
5. **Self-destruction:** Make sure you never allow to delegatecall to untrusted contracts and that they do not have a self-destruct functionality.

Insufficient Access control

Insufficient access control is a critical vulnerability in smart contracts that can lead to unauthorized actions. This flaw occurs when a contract fails to properly restrict who can execute certain functions.

Without adequate access controls, malicious actors can exploit the contract to perform actions like changing ownership, withdrawing funds, or altering key parameters, all of which should typically be restricted to certain roles like the contract owner or authorized administrators.

The consequences can range from financial loss to complete loss of control over the contract.

Example

Let's consider an example involving a smart contract that manages a decentralized organization (DAO) with multiple roles: owner, admin, and

member. The contract has a vulnerability in its access control that allows anyone to elevate their role.

```
pragma solidity ^0.8.0;

contract VulnerableDAO {
    address public owner;
    mapping(address => bool) public admins;
    mapping(address => bool) public members;

    constructor() {
        owner = msg.sender;
    }

    // Only owner should be able to add or remove admins, but
    this is not enforced
    function setAdmin(address account, bool status) public {
        admins[account] = status;
    }

    // Only admins should be able to add or remove members,
    but this is not enforced
    function setMember(address account, bool status) public {
        members[account] = status;
    }

    function withdrawFunds() public {
        require(msg.sender == owner || admins[msg.sender],
            "Not authorized");
        payable(msg.sender).transfer(address(this).balance);
    }

    // Other functions that should be restricted to specific
    roles
}
```

In this example, the *setAdmin* and *setMember* functions are supposed to be restricted to certain roles (owner for *setAdmin* and admin for *setMember*), but they are improperly designed, allowing anyone to elevate their role.

To fix this vulnerability, the *setAdmin* and *setMember* functions should include *require* statements to ensure that only authorized roles can change the status.

How to be safe?

Here are some strategies to help secure your smart contracts:

1. **Use Libraries:** Utilize well-audited libraries like OpenZeppelin's `AccessControl` to manage roles and permissions.
2. **Access Control Matrix:** Prepare a matrix of roles and functionalities in the system to mark which roles are allowed to perform specific operations.
3. **Least Privilege Principle:** Assign only the permissions that are necessary for each role. Avoid giving full control unless absolutely necessary.
4. **Robust access control mechanism:** Select a proper one, such as role-based access control (RBAC), to enforce expected behavior and maintain control over business flows and handled data.
5. **Custom Modifiers:** Create custom function modifiers that check the sender's permissions before executing the function.
6. **Explicit Checks:** Always include explicit *require* statements to check permissions at the beginning of functions.

Unchecked low-level calls

Unchecked low-level calls in smart contracts refer to the use of low-level methods like `.call()`, `.delegatecall()`, or `.staticcall()` without proper validation or error handling. These low-level calls are more flexible but also more dangerous than their high-level counterparts, such as `.transfer()` or `.send()`.

When a low-level call fails, it doesn't automatically throw an exception; instead, it returns a boolean value (*false*) to indicate failure. If the contract doesn't check this return value, the contract may continue executing as if the call were successful, leading to unexpected behavior or vulnerabilities.

Example

Below is a simple example of a smart contract that uses an unchecked low-level `.call()` to send Ether. If the `.call()` fails, the contract does not check the return value, leading to a vulnerability. In this example, the contract has a balance mapping that tracks the Ether balance of each user.

```

pragma solidity ^0.8.0;

contract VulnerableBalanceTracker {
    mapping(address => uint256) public balance;

    function deposit() public payable {
        balance[msg.sender] += msg.value;
    }

    function withdrawFunds(uint256 amount) public {
        require(balance[msg.sender] >= amount, "Insufficient
balance");

        // Unchecked low-level call
        (bool success, ) = msg.sender.call{value:
amount}("");

        // Additional logic after the call
        // This will execute even if the call fails
        balance[msg.sender] -= amount;
    }
}

```

If there was a situation when the transfer call fails the *withdrawFunds* function would continue to execute, reducing the attacker's balance in the contract even though no Ether was actually sent. The user would lose their Ether because they would not be able to withdraw more than the stored balance.

How to be safe?

The way to protect from unchecked low-level calls is quite straightforward. Always check the boolean return value when using low-level *.call()*, *.delegatecall()*, or *.staticcall()* methods. If you do not need to use those functions directly, consider using libraries that check the return value by default.

Front running

In a front-running attack, a malicious actor observes a pending transaction and quickly submits another transaction with a higher gas fee, aiming to get it included in the blockchain before the original transaction.

In addition to front-running, other related attack vectors like back-running and sandwich attacks also exploit the public nature of blockchain transactions. Back-running involves an attacker placing a transaction immediately after a targeted transaction in the same block, aiming to benefit from the state changes made by the targeted transaction. For instance, if a user triggers a function that increases the price of a token, a back-runner could place a sell order to immediately capitalize on the price increase.

Sandwich attacks are more complex and involve an attacker placing transactions both before and after a targeted user's transaction. In a typical sandwich attack on a decentralized exchange, the attacker first buys the token to increase its price (the "front" of the sandwich), waits for the targeted user's transaction to execute at this inflated price, and then sells the token to decrease its price (the "back" of the sandwich), profiting from the price differences.

Example

Below is an example of a decentralized exchange (DEX) contract that sets the token price based on reserves of Ether and tokens. It also includes a slippage protection mechanism. However, it's still simplified and vulnerable to front-running and sandwich attacks.

```
pragma solidity ^0.8.0;

interface IERC20 {
    function transfer(address recipient, uint256 amount)
    external returns (bool);
    function transferFrom(address sender, address recipient,
    uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns
    (uint256);
}

contract VulnerableDEX {
    IERC20 public token;
    uint256 public etherReserve;
    uint256 public tokenReserve;

    constructor(address _token, uint256 _initialTokenReserve)
```

```

{
    token = IERC20(_token);
    tokenReserve = token.balanceOf(address(this));
    etherReserve = msg.value;
}

function getPrice() public view returns (uint256) {
    return etherReserve / tokenReserve;
}

function buyTokens(uint256 maxSlippage) public payable {
    uint256 initialPrice = getPrice();
    uint256 tokenAmount = msg.value / initialPrice;
    uint256 newPrice = (etherReserve + msg.value) /
(tokenReserve - tokenAmount);

    require(tokenAmount <=
token.balanceOf(address(this)), "Not enough tokens in
contract");
    require(newPrice <= initialPrice * (100 +
maxSlippage) / 100, "Slippage too high");

    // Update reserves
    etherReserve += msg.value;
    tokenReserve -= tokenAmount;

    // Transfer tokens to the buyer
    token.transfer(msg.sender, tokenAmount);
}

// Other functions like sellTokens, addLiquidity, etc.
}

```

An attacker can observe a pending *buyTokens* transaction and place two transactions: one before and one after the targeted transaction. The first transaction could buy tokens to artificially inflate the price, and the second transaction could sell them to profit from the price difference.

The slippage protection mechanism in the provided DEX contract example is designed to ensure that the price movement caused by a user's trade does not exceed their specified tolerance (i.e., the *maxSlippage*). However, this protection doesn't prevent sandwich attacks, because the price movement is checked against the price at the beginning of the user's trade. It doesn't account for price manipulations that occur immediately before or after the user's

transaction. In a sandwich attack, the attacker's transactions temporarily manipulate the price, but the user's slippage protection doesn't detect or prevent this.

How to be safe?

Protecting against front-running and sandwich attacks requires a combination of design patterns, platform-level solutions, and user awareness. Here are some strategies to help secure smart contracts against these types of attacks:

1. **Slippage protection:** When a user initiates a trade, instead of (or in addition to) specifying a slippage percentage, they specify the minimum amount of tokens they are willing to receive in return for their trade. If the actual amount of tokens they would receive falls below this threshold due to price changes, the transaction is automatically reverted. This value would be calculated on the front-end side and accepted by the user.
2. **Commit-Reveal Schemes:** Implement a two-phase commit-reveal mechanism. Users first commit to a transaction without revealing the specifics. After a certain number of blocks, they reveal the transaction details, and the transaction is executed. This approach makes it difficult for attackers to front-run a transaction since they won't know the specifics until the reveal phase.
3. **Time-Weighted Average Price (TWAP):** Instead of executing trades at current market prices, use a time-weighted average price over a longer period. This reduces the profitability of short-term price manipulations.
4. **User Education:** Educate users about the risks of front-running and sandwich attacks. Encourage them to use lower slippage tolerances and to be cautious during times of high network congestion.

ACTIONS TO IMPLEMENT:

- Read materials:** Go through the resources to learn what are the common vulnerabilities and how to fix them.

- Share knowledge:** Add the descriptions of common vulnerabilities and their mitigations to the internal Knowledge Base and ask new developers to get familiar with that.
- Try it for yourself:** Create contracts that are vulnerable to learnt common vulnerabilities and try to fix them manually.
- Schedule peer reviews:** Add reviewers to Pull Requests that will mainly focus on detecting the common vulnerabilities to let your team train their skill.

READ MORE:

- [The role of access control](#)
- [SCSVS KNOWN ATTACKS](#)
- [DASP TOP 10](#)
- [SWC Registry](#)

Smart Contract Security Verification Standard

DESCRIPTION:

Smart Contract Security Verification Standard (v2) is a checklist created to standardize the security of smart contracts for developers, architects, security reviewers, and vendors. This list helps to avoid the majority of known security problems and vulnerabilities by guiding during every stage of the development cycle of smart contracts (from design to implementation).

A member of your team or an external auditor can go through check after check and give you clear information about the status of your project.

General

Common and general security problems including, among others: design, upgrades, and policies.


G1: Architecture, design and threat modeling

#	Description	Status	Comment
G1.1	Verify that every introduced design change is preceded by an earlier threat modeling.	Out of scope	The scope of the service did not include verification of internal procedures.
G1.2	Verify that the documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).	Passed	
G1.3	Verify that the SCSVS, security requirements or policy is available to all developers and testers.	Passed	
G1.4	Verify that the events for the (state changing/crucial for business) operations are defined.	Failed	Not all state changing operations emit events, see recommendation 5.1
G1.5	Verify that there exists a mechanism that can temporarily stop the sensitive functionalities of the contract in case of a new attack. This mechanism should not block access to the assets (e.g. tokens) for their owners.	Not applicable	
G1.6	Verify that the amount of unused cryptocurrencies kept on the contract is controlled and at the minimum acceptable level so as not to become a potential target of an attack.	Passed Failed Out of scope	
G1.7	Verify that if fallback function can be called by anyone, it is included in the threat modeling.	Not applicable	
G1.8	Verify that the business logic in contracts is consistent. Important changes in the logic should be allowed for all or none contract.	Not applicable	
G1.9	Verify that code analysis tools are in use that can detect potentially malicious code.	Not applicable	
G1.10	Verify that the latest version of the major Solidity release is used.	Not applicable	
G1.11	Verify that, when using the external implementation of contract, you use the current version which has not been superseded.	Not applicable	
G1.12	Verify that when functions are overridden to extend functionality, the super keyword is used to maintain functionality.	Not applicable	
G1.13	Verify that the order of inheritance is carefully specified.	Not applicable	
G1.14	Verify that there is a component that monitors contract activity using events.	Not applicable	
G1.15	Verify that the threat model includes whole transactions.	Not applicable	
G1.16	Verify that the leakage of one person's private key does not compromise the security of the entire project.	Not applicable	

G2: Policies and procedures

#	Description	Status	Comment
G2.1	Verify that the system's security is under constant monitoring (e.g. the expected level of funds).	Not applicable	
G2.2	Verify that there is a policy to track new security bugs and to update the libraries to the latest secure version.	Not applicable	

Additionally, you can calculate your security score and see the areas that are worth taking care of.



The compliance of the project with the SCSVS v2.0 standard was checked. The verification is intended to act in the spirit of transparency, the score should indicate the overall security status of the project.

The version used corresponds to the following commitID: *2a9c500a6fbe2d55d42c41769003c82be38e9c79*

The detailed information and current version of the standard can be found here:
<https://github.com/ComposableSecurity/SCSVS/>

Project name	Test Project	Compliance score	94%
Passed	Failed	Not applicable	Out of scope
183	5	18	7

General	Components	Integration
G1: Architecture, design and threat modeling	C1: Token	I1: Basic
G2: Policies and procedures	C2: Governance	I2: Token
G3: Upgradeability	C3: Oracle	I3: Oracle
G4: Business logic	C4: Vault	
G5: Access control	C5: Bridge	
G6: Communications	C6: NFT	
G7: Arithmetic	C7: Liquid staking	
G8: Denial of service	C8: Liquidity pool	
G9: Blockchain data		
G10: Gas usage & limitations		
G11: Code clarity		
G12: Test coverage		

Legend
NEEDS ATTENTION
FULLY PASSED
NOT APPLICABLE

Detailed information

Auditing company	Composable Security
Verification date	30.02.2023
Verified commitID	<i>cba26ab250d81f63313b1b075eb45c5295a0e03e</i>
	Smart contracts:
	/contracts
	Token.sol
	Vault.sol
	/libraries
	...
Scope of tests	Architecture documentation Procedures verification Upgradeability process SCDLC

GOAL:

Building security by design and expanding the list of attack vectors

PROS:

- comprehensive security coverage of the whole development process
- knowledge of what exactly was covered during the review
- free if used internally
- developing high-quality code
- a very wide and updated threat database

CONS:

- requires intensive workshop
- requires to gather team from different departments
- requires appropriate competences to verify certain attack vectors

WHEN:

At all stages, from design to post-deployment. Ideally as:

- source of threats for threat modeling sessions
- full coverage - before release of major change
- selected coverage - before releasing new components or integrating with a new protocol

ACTIONS TO IMPLEMENT:

- Find the Right Team Member:** Select a team member that will be responsible for SCSVS compliance, ideally, that would be the Security Champion selected before.
- Schedule internal audits:** Decide when to conduct an internal audit and save that date.
- Audit your components:** Verify the security checks from the general and component categories.
- Audit your integrations:** When you are implementing integration with some external protocol, verify the security checks from the particular integrations category.
- Check compliance:** Check whether the protocols that you integrate with are compliant with SCSVS.

RESOURCES:

<https://github.com/ComposableSecurity/SCSVS/>

Secure Protocol Upgrades

DESCRIPTION:

The ability to update business logic, parameters and expand the architecture of a project based on smart contracts is one of the areas particularly exposed to various risks. During the update, in addition to the many benefits and opportunities they bring, bugs that were not previously present in contracts may be introduced.

It is worth mentioning that the upgradeability is also considered antithetical to Web3, and by some, it is considered to be a vulnerability itself. However, it has also helped to fix some projects and protect funds.

GOAL:

Perform a secure upgrade that does not change the logic unexpectedly and do not introduce security bugs.

PROS:

- covers upgradeability risks
- low cost and effort
- reusable

CONS:

- needs time to implement
- usually touches whole protocol

WHEN:

Before the release of the new version.

ACTIONS TO IMPLEMENT:

- Use existing code:** There are production-ready libraries and packages (see references) that can help build upgradeable contracts with many potential security risks covered.
- Create upgrade tests:** Create the test cases that verify whether the storage layout and business logic was not changed unexpectedly after the upgrade. Test it on the local mainnet fork.
- SCSVS G3: Upgradeability:** Check compliance with the G3: Upgradeability category.

READ MORE:

[SCSVS G3: Upgradeability](#)

[OpenZeppelin: Upgrades](#)

**GET HELP AT
ANY STAGE**

We hope that after reading our guide, you now have a broader perspective and understanding of how much can be done to minimize risks.

Security is too important to treat it as an unpleasant obligation. If you think about your project in the long term, don't let anyone hinder your plans due to negligence that can be avoided.

As **Composable Security**, we will help you take care of your smart contract security and prioritize activities.

We offer:

- Smart contract audits
- Security consultations
- Threat modeling
- Security overviews and more!

Contact us and let's start improving your security.

<https://composable-security.com/contact>

P.S.

If you haven't found something in this guide that you think will be useful to others, please contact us.

We plan to develop this project further.



Thank you for your time, now get to work.