LUXY

CYREX

Smart Contract Audit

Prepared for: **Christopher O'Shea, Finance Lead, Luxy**
Prepared by: **Tim De Wachter, CTO**, ***Cyrex Ltd***

01/12/2021

Table of Contents

Executive summary

Smart contract audit

Cyrex was contracted by Luxy to conduct a smart contract audit in order to determine its exposure to a targeted attack. All activities were conducted in a manner that simulated a malicious actor engaged in a targeted attack against the scope with the goals of:

- Identifying if a remote attacker could penetrate the scope its defences.
- Determining the impact and possibility of a security breach.

Efforts were placed on the identification and exploitation of security weaknesses that could allow a remote attacker to gain unauthorized access to organizational data. The attacks were conducted with all levels of access that a general internet user would have.

As Luxy provided Cyrex the source code, we can label this kind of test as a white box test. Cyrex was granted access to the application with all regular user privileges.

Regression test

With the regression testing we make sure the vulnerabilities discovered during the smart contract audit are patched in a correct manner and no other vulnerabilities have been introduced during the patching process.

What follows is a conclusion concerning the overall security maturity of the application and the tested vulnerability types.

We are confident that the penetration test and this report helps the customer to raise its security of the Luxy Finance Smart Contracts to a higher level, this by enforcing the principles of confidentiality, integrity and availability.

Conclusion

Overall

Cyrex determined that the overall security maturity of the smart contracts is great and will meet the risk appetite of any end user. No vulnerabilities were discovered during the audit. Various recommendations were implemented in a correct manner but more importantly the smart contracts were tested and validated thoroughly by Cyrex' smart contract security experts.

Security Controls

Security best practices have been implemented in different parts of the contract. A few examples of this are changing state before paying currency and leveraging of re-entrancy guard to avoid re-entrancy. Usage of solidity 8.x to prevent arithmetic underflow or overflows, leveraging SafeERC20 for secure token transfers and implementing proper access controls, using ownerOnly and proper require statements.

Scope of test

Cyrex performed a smart contract audit on the Luxy & Luxy Finance Smart Contracts, this test was performed starting the 25th of October 2021 up till and including the 4th of November 2021.

During the audit, strict protocols, guidelines and a unique workflow have been followed. Different frameworks were integrated into this process flow which are in line with the ethical hacking procedures. The process involved an active analysis of the application for any weaknesses, technical flaws or vulnerabilities.

During the entire smart contract audit testing life cycle, Cyrex performed the following actions in order to determine security issues within the application:

1. Analysis and testing of different methods
2. Tampering of different parameters within those methods
3. Identification of potential injection points, security flaws and vulnerabilities
4. Exploitation to provide Proof of Concept (PoC)

We are confident the maturity level of the smart contracts meets the security requirements of any end user; therefore, the smart contracts can be publicly exposed and be put into a production environment.

We want to thank Luxy for putting trust in our knowhow and expertise concerning ethical hacking specific to applications.

Fixed vulnerabilities and recommendations

This section lists all the vulnerabilities and recommendations that Cyrex determined to be successfully fixed within the scope of this regression test.

ID	Title
L001	Impossible to update tier discount settings
LR001	Constant state variables should be declared constant
LR002	Use the external attribute for functions never called from the contract.
LR003	Zero address validation
LR004	Dead Code in LibPart.sol

Tested Vulnerability Types

The smart contracts have been tested for the following types of vulnerabilities:

Access Control Flaws

Within the application's core security mechanisms, access controls are logically built upon authentication and session management. The application needs a way of deciding whether it should **permit** a given request to perform its attempted action or access the resources that it is requesting.

Access controls are a **critical defence mechanism** within the application because they are responsible for making these key decisions.
When they are defective, an attacker can often:

- Compromise the entire application
- Take control of administrative functionality
- Access sensitive data belonging to every other user.

Broken access controls are among the most commonly encountered categories of web application vulnerabilities.

Brute Force Attacks

A brute force attack can manifest itself in many different ways, but primarily consists in an attacker configuring predetermined values, making requests to a server using those values, and then analysing the response. For the sake of efficiency, an attacker may use a dictionary attack or a traditional brute-force attack.

Brute-force attacks are often used for attacking authentication and discovering content/pages within a web application. These attacks are usually sent via GET and POST requests to the server. In regard to authentication, brute force attacks are often mounted when an account lockout policy is not in place.

Broken Authentication

Confirmation of the user's identity, authentication, and session management are critical to protect against authentication-related attacks. There may be authentication weaknesses if the application:

- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords;
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers", which cannot be made safe.
- Uses plain text, encrypted, or weakly hashed passwords (see A3:2017-Sensitive Data Exposure).
- Has missing or ineffective multi-factor authentication.
- Exposes Session IDs in the URL (e.g., URL rewriting).
- Does not rotate Session IDs after successful login.
- Does not properly invalidate Session IDs. User sessions or authentication tokens (particularly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

Information Disclosure

Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Applications can also leak internal state via how long they take to process certain operations or via different responses to differing inputs, such as displaying the same error text with different error numbers.

Web applications will often leak information about their internal state through detailed or debug error messages. Often, this information can be leveraged to launch or even automate more powerful attacks.

Denial of Service (DoS)

The **Denial of Service (DoS)** attack is focused on making a resource (site, application, server) unavailable for the purpose it was designed. There are many ways to make a service unavailable for legitimate users by manipulating network packets, programming, logical, or resources handling vulnerabilities, among others. If a service receives a very large number of requests, it may cease to be available to legitimate users. In the same way, a service may stop if a programming vulnerability is exploited, or the way the service handles resources it uses.

Sometimes the attacker can inject and execute arbitrary code while performing a DoS attack in order to access critical information or execute commands on the server. Denial-of-service attacks significantly degrade the service quality experienced by legitimate users. These attacks introduce large response delays, excessive losses, and service interruptions, resulting in direct impact on availability.

Business Logic Flaws

Most security problems are weaknesses in an application that result from a broken or missing security control (authentication, access control, input validation, etc. ...). By contrast, business logic flaws are ways of using the legitimate processing flow of an application in a way that results in a negative consequence to the organization.

Security Misconfigurations

Security misconfiguration can happen at any level of an application stack, including the network services, platform, web server, application server, database, frameworks, custom code, and pre-installed virtual machines, containers, or storage. Automated scanners are useful for detecting misconfigurations, use of default accounts or configurations, unnecessary services, legacy options, etc

Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.

The business impact depends on the protection needs of the application and data. Attackers will often attempt to exploit unpatched flaws or access default accounts, unused pages, unprotected files and directories, etc to gain unauthorized access or knowledge of the system.

Improper Input Validation

Input validation is a frequently used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe processing within the code, or when communicating with other components. When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

It is important to emphasize that the distinctions between input validation and output escaping are often blurred, and developers must be careful to understand the difference, including how input validation is not always sufficient to prevent vulnerabilities, especially when less stringent data types must be supported, such as free-form text.

Re-entrancy Attacks

A reentrancy attack can occur when you create a function that makes an external call to another untrusted contract before it resolves any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the effects were resolved.

There are two main types of reentrancy attacks: single function and cross-function reentrancy.

Single function reentrancy attack
This type of attack is the simplest and easiest to prevent. It occurs when the vulnerable function is the same function the attacker is trying to recursively call.

Cross-function reentrancy attack
These attacks are harder to detect. A cross-function reentrancy attack is possible when a vulnerable function shares state with another function that has a desirable effect for the attacker.

Over- & Underflow Attacks

The uint overflow/underflow, also known as uint wrapping around, is an arithmetic operation that produces a result that is larger than the maximum above for an N-bit integer, or produces a result that is smaller than the minimum below for an N-bit integer.

Like mileage counters in cars, integers expressed in computers have a maximum value and once that value is reached they simply turn back to the beginning and start at the minimum value. Similarly, subtracting 4 from 3 in an unsigned integer will cause an underflow, resulting in a very large number.

Block Gas Limit

The block gas limit is Ethereum's way of ensuring blocks don't grow too large. It simply means that blocks are limited in the amount of gas the transactions contained in them can consume. Put simply, if a transaction consumes too much gas it will never fit in a block and, therefore, will never be executed.

This can lead to a vulnerability that we come across quite frequently: If data is stored in variable-sized arrays and then accessed via loops over these arrays, the transaction may simply run out of gas and be reverted. This happens when the number of elements in the array grows large, so usually in production, rather than in testing. The fact that test data is often smaller makes this issue so dangerous since contracts with this issue usually pass unit tests and seem to work well with a small number of users. However, they fail just when a project gains momentum and the amount of data increases. It is not uncommon to end up with unretrievable funds if the loops are used to push out payments.

Frontrunning

Potential frontrunning is probably the hardest issue to prevent on smart contracts. Frontrunnuing can be defined as overtaking an unconfirmed transaction. This is a result of the blockchain's transparency property. All unconfirmed transactions are visible in the mempool before they are included in a block by a miner. Interested parties can simply monitor transactions for their content and overtake them by paying higher transaction fees. This can be automated easily and has become quite common in decentralized finance applications.