Least Authority

PRIVACY MATTERS

Plumo Protocol: Arithmetic Optimizations

Security Audit Report

cLabs

Final Report Version: 30 June 2021

Table of Contents

Overview

Background

cLabs has requested that Least Authority perform a security audit of Plumo, a SNARK-based protocol for achieving an ultra-fast light client for the Celo blockchain. Plumo is based on the `Groth16` SNARK. Specifically, Celo uses the [BW6-761]() curve and the Groth16 proving system, as implemented in [ZEXE](), a Rust library for decentralized private computation. A key feature of Celo is the use of advanced speed optimizations for computations on BW6-761.

This audit is the first of three consecutive reviews, as follows:

1. Plumo Protocol Arithmetic Optimizations (this report)
2. Plumo Protocol Underlying ZEXE Gadgets
3. Plumo Protocol High-Level Gadgets

Project Dates

- **October 19 - November 11**: Arithmetic Optimizations Code Review *(Completed)*
- **November 13**: Delivery of Arithmetic Optimizations Initial Audit Report *(Completed)*
- **January 18 - 25**: Verification *(Completed)*
- **January 27:** Delivery of Final Audit Report *(Completed)*
- **June 30**: Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Ann-Christine Kycler, Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Plumo Protocol Arithmetic Optimizations followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following pull requests are considered in-scope for the review:

- GLV:
  - https://github.com/celo-org/zexe/pull/16
  - https://github.com/celo-org/zexe/pull/24
- Batch inversion + batch subgroup membership checks
  - https://github.com/celo-org/zexe/pull/11
- Assembly:
  - https://github.com/celo-org/zexe/pull/14
  - https://github.com/celo-org/zexe/pull/20
  - https://github.com/celo-org/zexe/pull/32

Specifically, we examined the Git revisions for our initial review:

```
2d389ddfd84a6b7c632700050c473e5edb0a74e8
```

For the verification, we examined the Git revisions:

`5a845949200a0aacc4e315ecf12500e0496eebf1`

`679fd44ac690426b84704decf3b984ae6019e6a0`

`199dcade443bb506825fe06ec5b824e4e9619dbf`

`B8a4fbb6895a7006d6cacb67b5c6ac77fa4b4365`

`5c59339000365e8eae266ea46e059e097746e743`

For the review, these repositories were cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/celo-bls-snark-rs

https://github.com/LeastAuthority/plumo-zexe

All file references in this document use Unix-style paths relative to the project's root directory.

Supporting Documentation

The following documentation was available to the review team:

- Specification for ZEXE Algorithmic/Performance Optimisations: https://hackmd.io/@celo/S10UeWUBD
- S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra and H. Wu, 2020, "ZEXE: Enabling Decentralized Private Computation," *2020 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, pp.947-964. [Bowe et. al. 20]
- A. Gabizon, K. Gurkan, P. Jovanovic, G. Konstantopoulos, A. Oines, M. Olszewski , M. Straka, E. Tromer, and P. Vesely, "Plumo: Towards Scalable Interoperable Blockchains Using Ultra Light Validation Systems." (document shared with Least Authority via email on September 4, 2020) [Gabizon et. al. 20]
- R.P. Gallant, R.J. Lambert, S.A. Vanstone, 2001, "Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms." *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pp.190–200. [GLV01]
- Y.E. Housni, A. Guillevic, 2020, "Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition." *IACR Cryptol. ePrint Arch*, pp.351. [HG20]
- C. Koc, K. Kaya, T. Acar, and B.S. Kaliski, 1996, "Analyzing and comparing Montgomery multiplication algorithms." *IEEE micro 16.3*, pp.26-33. [KAK96]
- B. Möller, 2003, "Improved Techniques for Fast Exponentiation." *Information Security and Cryptology – ICISC 2002. LNCS 2587. Springer-Verlag,* pp.298–312. [M03]
- E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom, 2019, "The 9 Lives of Bleichenbacher's CAT: New Cache Attacks on TLS Implementations." *2019 IEEE Symposium on Security and Privacy (SP)*, pp.435-452. [Ronen et. al. 19]
- P.L. Montgomery, 1987, "Speeding the Pollard and elliptic curve methods of factorization." *Mathematics of computation*, 48(177), pp.243-264. [M87]
- D.J. Bernstein, and B.Y. Yang, 2019. "Fast constant-time gcd computation and modular inversion." *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp.340-398. [BY19]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;

- Performance problems or other potential impacts on performance;
- Changes (optimizations) made to ZEXE;
- Data privacy, data leaking, and information integrity;
- Vulnerabilities in the code leading to adversarial actions and other attacks;
- Protection against malicious attacks and other methods of exploitation;
- Performance problems or other potential impacts on performance;Anything else as identified during the initial analysis phase.

Findings

General Comments

The cLabs team has optimized Plumo's field and curve arithmetic code by implementing a new pairing friendly elliptic curve, BW6-761, which is suited for one layer proof composition of pairing based SNARKs, a property that is required in the Plumo protocol [Bowe et. al. 20]. BW6-761 is designed to create a pairing-friendly amicable chain on top of BLS12-377, further explained in [HG20]. The key contributions of the code are highly speed-optimized versions of batched inversion, addition, scalar multiplication, and subgroup checks in these curves. The cLabs team achieved this by implementing the Montgomery trick (See the Specification for ZEXE Algorithmic/Performance Optimisations) for faster batched addition of curve points, by using special Endomorphisms [GLV01], and window NAF exponentiation [M03], along with general strategies for highly efficient code, such as using tight loops, ensuring memory locality, and avoiding frequent heap allocations. In addition, the cLabs team optimized the base field arithmetics by writing the addition, subtraction, and multiplication in Assembly. The Assembly code was optimized by performing computations in the Montgomery product and using the Separated Operand Scanning from [KAK96].

We commend the cLabs team for carefully and thoroughly implementing the new BW6 curve as a replacement of CP6 in ZEXE, which demonstrates strong considerations for maintaining security. While our team did not identify any immediate security issues, we have made several suggestions on maintaining the integrity of the code base.

Review Scope

The scope of the review was sufficient. It covered all relevant parts of the code concerning the implementation of the new curve, as well as the necessary field arithmetics. Our review concentrated on the core areas described below.

GLV Implementation

During our review of the cLabs team's Plumo protocol implementation of the new BW6 curve as a replacement of CP6 in ZEXE, we compared the GLV implementation against the specification as described in [GLV01] and found no errors in the implementation. The Plumo code uses a particular optimization to offload expensive division to a precomputation phase, which is not outlined in the original reference [GLV01]. This optimization is explained in detail in the Specification for ZEXE Algorithmic/Performance Optimisations, from which we analyzed the given correctness proof and identified no errors. Furthermore, we found no errors in its corresponding implementation.

wNAF implementation

Our team compared the wNAF implementation, including the table, decomposition, and application, against the specifications as described in [M03]. No issues were identified, with the exception of incorrect behavior that might occur from unintended use of input parameters (Suggestion 2).

Batched Trait Implementations

Additionally, our team compared the various implementations of the `BatchGroupArithmetic` trait against the specification as described in the Specification for ZEXE Algorithmic/Performance Optimisations (including `batch_scalar_mul_in_place()`), which included the batched inversion operations and the various batch addition operations. During the audit, the cLabs team improved the documentation around the bucketed batch addition operation, which significantly aided our team in verifying the correctness of the respective function. In this instance, we did not identify security issues, except for similar unintended behavior as noted above (Suggestion 2).

Assembly Code Implementation

Our team checked the Assembly code implementation of the underlying field arithmetics of the BW6-768 curve. The multiplication implements the Selective Operand Scanning (SOS) algorithm given in [KAK96]. We verified that the implementation corresponds to the algorithm described in the paper and analyzed and checked the correctness of the subtraction and addition functions. The Assembly code was tested for byte slices that represent actual field elements and no unexpected behaviour occurred. The functions adhere to standard C calling-conventions, as used in Rust when “extern C” functions are used. Furthermore, our team checked the safe integration of the Assembly functions into the Rust code (i.e. safe handling of pointers) and found the Assembly code to be constant time, which we discuss further in the System Design section.

Code Quality

The code base is well organized, which facilitated our ability to review and comprehend the implementation of the new curve. Our team found the test coverage to be sufficient. We recommend some improvements to further aid the ability to maintain and review the code (Suggestion 2; Suggestion 3; Suggestion 4). In particular, multi-precision (`bigint`) arithmetic is implemented for a small set of limb counts in order to improve the potential for optimizations at compile time, which extends to the types representing fields and groups that internally use multi-precision arithmetic. At times, this leads to difficulty in understanding macro structures, even though they are not extraordinarily complex. In addition, modules have a tendency to re-publish a considerable amount of values from modules they use themselves. As a result, one piece in the source code is identified by several different names, thus increasing cognitive overhead by the reviewer or maintainer of the code.

The code operates on a very low level, thus relying on only a small number of external dependencies, which reduces the surface area for potential vulnerabilities. According to the RustSec Security Advisory Database, no known vulnerabilities exist in the dependency versions used in the Plumo code and GitHub will automatically warn the cLabs team of newly published security advisories, unless the cLabs team chooses to opt out of the Dependabot feature.

Documentation

Our team found the documentation to be broad and comprehensive and found the Specification for ZEXE Algorithmic/Performance Optimisations and code comments to be a particularly helpful source in understanding most of the code. In addition, the accompanying papers, [GLV01], [HG20], [M03] and [KAK96], were used for reference. Thorough documentation is critical as the implementation uses highly non-trivial optimization of elliptic curve operations and we commend the cLabs team for their comprehensive documentation.

The Specification for ZEXE Algorithmic/Performance Optimisations documentation is well written and valuable in providing an understanding of most of the complex aspects of code. In areas where our team required further insight, the cLabs team responded quickly and thoroughly to requests for further documentation and clarification. One area of improvement that benefited from further clarification were the proofs in section GLV of the Specification for ZEXE Algorithmic/Performance Optimisations documentation.

Although most of the complex components of the implementation are well commented, we suggest additional code comments or clear references to the appropriate sections of the Specification for ZEXE Algorithmic/Performance Optimisations documentation (Suggestion 4). In particular, there are some instances where low-level functions crash for some of the possible inputs, and there is an absence of comments that define valid inputs. Important variables from research papers and literature were named accordingly, however, code comments that reference the relevant papers and literature would be helpful for users and reviewers of the code in order to avoid confusion which could cause misuse issues. For example, in some instances, single letters such as R have various meanings in different reference materials. The cLabs team significantly improved the code comments in the bucketed batch addition function upon request, which helped us understand and verify the correctness of the addition trees used therein. We recommend clearly defining these variables and potential specific properties, which will provide clarity for users and reviewers of the code.

System Design

Underlying ZEXE Implementation

The design and structure of the Plumo code is mostly predetermined by the underlying ZEXE code base, which the cLabs team has further optimized. The structure of ZEXE matches algebraic structures like fields, groups, and curves. The system design requirements that ZEXE fulfills are well-defined and are also used in the reference research papers, thus making Plumo's optimizations relatively straightforward from a design perspective.

The design is theoretically sound and practical. However, we recommend that the cLabs team consider timing side-channel and constant time algorithms (Suggestion 5). With the exception of the Assembly code, functions that operate on potentially secure values are not constant-time and this could be used by an attacker who can execute processes on the same system to gain knowledge about the secret values. For operations like batch-inversion, there are recent publications on constant-time algorithms which could be considered [BY19].

Assembly Functions

Plumo implements the prime field arithmetics in the underlying base field of the curve BW6-761 in Assembly. The optimized Assembly functions for the modular multiplication, addition, and subtraction follow the standard calling-conventions. The optimized multiplication follows [KAK96] and all three Assembly functions take constant time, so execution time does not depend on the input, thus avoiding the potential for side-channel attacks.

Input Validity Checks in Internal Functions

Many functions do not check whether the input values they were called with are inputs the functions are supposed to process. For many low-level functions, we identified several inputs that crashed the test function or gave incorrect results. Our team did not identify a method in the existing implementation to make the public functions call those functions with such arguments. At a minimum, we recommend including a comment that describes what makes a valid input for functions that crash for certain inputs. However, it would be optimal to have a check to determine whether the inputs are valid, if only during debug builds (Suggestion 2), or to ensure that internal helper functions can not be called publicly (Suggestion 1).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

SUGGESTION	STATUS
Suggestion 1: Adhere to a Clear Distinction Between Private and Public Functions	Resolved
Suggestion 2: Document Assumptions Made by Functions	Resolved
Suggestion 3: Clean up Namespaces	Unresolved
Suggestion 4: Improve Published Documentation	Resolved
Suggestion 5: Consider Side-Channel Attacks	Unresolved

Suggestion 1: Adhere to a Clear Distinction Between Private and Public Functions

Location

Trait Batch Group Arithmetic in https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra-core/src/curves/batch_arith.rs

Synopsis

Several functions have higher visibility than is necessary in order to accomplish the intended behavior. This is particularly concerning when the functions are internal helper functions that do not return the correct result for all inputs, or even crash in some instances.

Mitigation

Emulate private functions and macros in public traits and eventually implement publicly callable wrappers that check for unintended arguments around the (faster) internal functions. While previous Rust versions allowed for private and public parts in traits by splitting a trait accordingly, this functionality has been removed. We recommend splitting the trait into a private and a public part, wherein the private part must be represented by a public trait within a private module.

Status

The cLabs team issued a pull request which clarifies the distinction between private and public visibility of their interfaces. In particular, they restricted helper functions and constants to only be visible inside the current crate and only exposed functions that are safe and intended for use by external developers.

In addition, they emulated functions in public traits, which should only be visible in the current crate by representing those functions in a public trait inside a `pub(crate)` module, as recommended.

Verification

Resolved.

Suggestion 2: Document Assumptions Made by Functions

Location

For example, functions in the `BatchGroupArithmetic` trait defined in
https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra-core/src/curves/batch_arith.rs

This example should not be considered to be exhaustive.

Synopsis

For several functions throughout the code base, it is easy to find arguments that make the function crash or for which it returns wrong results. For example, `batch_wnaf_opcode_recoding()` crashes on input `w=0` and `batch_add_in_place()` gives incorrect results when the first elements of the tuples in the argument array occur multiple times, as w.g. with `index = [(0,1), (0,2), (0,3)]`.

While these functions are not called with such invalid arguments internally, it is best practice to document the assumptions or preconditions of functions. This allows reviewers and maintainers of the code immediate understanding of what they should and should not pass into the function. Furthermore, it allows for easier verification of correctness.

Mitigation

Clarify assumptions made by functions on their arguments and state within the comments of the function definition what the requirements are on the inputs such that the function behaves correctly. While it should be specific and correct, it does not need to be machine readable.

Additionally, it should be checked whether the assumptions hold in debug builds. In this case, the assumption needs to be encoded in a way that the computer can automatically check if it holds.

While we have identified this issue with the functions noted above, this example should not be considered to be exhaustive. We recommend that the cLabs team further identify and document functions that make assumptions without checking them.

Status

The cLabs team issued a pull request that adds comments that specify the classes of inputs for which the aforementioned functions fail. Automatic verification would be desirable in order to prevent regressions in the future. However,at present, no such tooling is yet implemented in the Rust ecosystem and the cLabs team is unable to make use of automatic verification until that effort has been completed.

Verification

Resolved.

Suggestion 3: Clean up Namespaces

Location

https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra-core/src/lib.rs

https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra-core/src/curves/mod.rs https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra-core/src/fields/mod.rs

https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra-core/src/fields/models/mod.rs

https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra/src/lib.rs

https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra/src/bw6_761/mod.rs

https://github.com/celo-org/zexe/tree/2d389ddfd84a6b7c632700050c473e5edb0a74e8/algebra/src/bw6_761/fields/mod.rs

This list of locations should not be considered to be exhaustive.

Synopsis

Throughout the code base, types and values from submodules are frequently re-exported, which has led to them having several valid identifiers. For example, consider the files `mod.rs` and `sub.rs`:

```
// file mod.rs
pub mod sub; pub mod other; pub mod yet_another;
pub use sub::*;
pub use other::*;
pub use yet_another::*;

// file sub.rs
pub const PI = 3;
```

In this example, it is difficult to see in `mod.rs` where `PI` is defined, since it may be present in any of the re-published modules.

Note that in this case, both `PI` and `sub::PI` are valid identifiers for the constant. There are repeated instances of this in the in-scope code, such that a single item has a large number of names it acquired through re-publishing. This makes the code more difficult to read and maintain.

Mitigation

The following two approaches are recommended to be implemented in conjunction with each other:

- First, avoid blanket re-publishing of modules in the style of `pub use some_mod::*`. Instead, consider publishing the module itself, which avoids unnecessary ambiguity. However, importing whole modules into the local namespace is not an issue.
- Second, only re-publish an item from a used module if that module is considered private. This avoids unnecessary namespace blowup and reduces the effort required from readers and maintainers of the code.

Status

The cLabs team has responded with the decision to address this suggestion at a later point in time, and provided the following insights:

The ZEXE code base that the cLabs team initially forked has been abandoned and restructured into a new project (arkworks). In part, this decision was made in order to resolve some of the issues outlined in this suggestion. The cLabs team has responded that they intend to port Plumo to arkworks at a future point in time, which would help resolve this suggestion.

Furthermore, the cLabs team does not consider this to be an immediate security issue. Our team does not regard this to be a security critical suggestion, however, confusing and unclear code may result in the introduction of errors and mistakes as development continues. We urge the cLabs team to consider implementing the suggested mitigation at the earliest convenience.

Verification

Unresolved.

Suggestion 4: Improve Published Documentation

Synopsis

Central ideas of the optimizations applied in the code were explained and derived in the accompanying reference (Specification for ZEXE Algorithmic/Performance Optimisations), which is currently an external source and not part of the code base or referenced in the repository. Since understanding certain optimizations without that reference is difficult, it is important that this information remains accessible and conveniently discoverable. In addition, in some instances, single letters such as R have various meanings in different reference materials.

Mitigation

In order to make it possible for reviewers to verify the correctness of the code, we recommend including the Specification for ZEXE Algorithmic/Performance Optimisations into the repository and to expand all relevant code comments by referencing the relevant sections in the document accordingly. Moreover additional code comments should clarify the intended use of variable names.

Status

The cLabs team has updated the documentation in the repository to include all relevant information, presented in an understandable format, as recommended.

Verification

Resolved.

Suggestion 5: Consider Side-Channel Attacks

Synopsis

Side-channel attacks utilize both the explicit input and output values of a system and metadata, such as how long the computation took or how much power was consumed during the computation. If an attacker is able to learn the values and infer information about secret data, it can be considered that they have access to a side-channel. For networked applications, the most relevant form of side-channel is the timing side-channel and the most well-known timing side-channel attacks are variants of Bleichenbacher's attack [Ronen et. al. 19].

In order to avoid vulnerabilities to this class of attacks, many cryptographic protocols are diligent in their use of constant-time algorithms for cryptographic computations. We note that the arithmetic for the new curve does not run in constant time.

For the scope of this audit, this is not an issue, since we are looking at the correctness of an implementation of arithmetic computations, not the security of a distributed system. However, the fact that many computations are not constant-time means that the implementation needs to be used with caution in interactive protocols.

We acknowledge that the ZEXE code base (and by extension Plumo) does not have constant-time behavior as a design goal, and that achieving that property is a significant undertaking. Additionally, unlike the setting of the Bleichenbacher variants, variable-time computations in ZEXE/Plumo are not immediately performed on attacker input, leaving open the question of how such an attack could even be mounted. However, this is a long-term security goal that is worth considering.

Mitigation

We recommend investigating constant-time alternatives to the current algorithms and to evaluate their performance and efficacy for interactive use-cases. Possible starting points could be the Montgomery ladder [M87] and the greatest common divisor algorithm proposed in [BY19].

Status

The cLabs team has responded that this suggestion will remain unaddressed since the ZEXE code base from which Plumo is forked would require significant redesign in order to achieve this property.

Additionally, the cLabs team has noted that the performance costs of achieving side-channel attack resistance are not acceptable for them.

Finally, the cLabs team makes the argument that the optimizations are used for the setup and proof generation. The setup is secure as long as one involved party is honest (i.e. not under a successful attack). Using the optimizations for proof generation is secure, because Plumo does not use ZEXE for hiding data, only for verifying the integrity of transactions. Therefore, there is no relevant secret data that could be extracted from them. Our team agrees with this assessment for the Plumo use case.

Nonetheless, we recommend that the cLabs team continue to assess the risk of side-channel attacks in future development efforts and consider further addressing them if the above-mentioned circumstances change.

Verification

Unresolved.

Recommendations

We recommend that the unresolved *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

We also recommend that the cLabs team clean up namespaces in order to facilitate better readability and comprehension of the implementation.

Finally, we commend the cLabs team for publishing the Specification for ZEXE Algorithmic/Performance Optimisations, which is now linked to the public repository and easily accessible by maintainers and reviewers of the code. Furthermore, the cLabs team has integrated relevant technical details into the code comments by documenting assumptions made by functions.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.