ZKSECURITY

Audit of Mysten Labs' zkLogin circuits and ceremony

September 11th, 2023

Introduction

On August 16th, 2023, zkSecurity was tasked to audit Mysten Lab's zkLogin application, consisting of Circom circuits, typescript files to generate the public inputs, and ceremony code on top of snarkjs.

Three consultants spent six person-weeks looking for security issues. The code was found to be of high quality, with numerous comments on assumptions made by the various Circom templates, as well as tests for the various components.

No important security issues were found.

Scope

The audit included two distinct components or phases. The first phase included the **Circom** circuits of zkLogin, while the second phase included some ceremony-related patches on top of **snarkjs** and **phase2-bn254**.

Circom circuits. The zkLogin application was conceptually split into three parts:

- An RSA signature verification component, forked from `` `doubleblind-xyz/double-blind` `` and `` `zkp-application/circom-rsa-verify` ``.
- A SHA-256 hasher component, forked from `` `TheFrozenFire/snark-jwt-verify` `` and relying for the most part on **circomlib**.
- The main zkLogin logic which includes parsing the JWT, extracting some claims, verifying the signature, and deriving an address seed (later used to derive the address in the clear) from some of the claims.

Patches on top of snarkJS. Two set of patches were reviewed:

- The first patch consisted of a fix to the CRS trimming logic of `` `kobigurk/phase2-bn254` ``, implemented in a Mysten Lab's fork (https://github.com/MystenLabs/phase2-bn254). We review and explain the fix in the first part of the **ceremony section** of this report.
- The second patch was written on top of the **snarkjs** application in order to improve interoperability with third-party software. The patch was implemented in a Mysten Lab's fork as well (https://github.com/MystenLabs/snarkjs/tree/ceremony). We review and explain the fix in the second part of the **ceremony section**.

As the code changed frequently during the audit, we also focused on reviewing the following commit: `` `8868f13477be63b4417310e934f5c103b7fba19c` `` (from August 16th).

Overview of zkLogin

This section provides a simplified overview of the zkLogin application.

zkLogin is a new way of authenticating users in **Sui**. It is to be integrated in the protocol as an equal authentication mechanism as the existing ones: Pure Ed25519, ECDSA Secp256k1, ECDSA Secp256r1, and Multisig (see **https://docs.sui.io/learn/cryptography/sui-signatures** for more details).

Replacing keys with passwords

The idea behind zkLogin is that users, in general, have a hard time managing cryptographic keys, yet are used to manage passwords and go through multi-factor authentication flows. As such, bridging Single Sign-On (SSO) to the blockchain world, while preserving the privacy of the users, could be a good way to improve the user experience while maintaining a strong level of security.

In more detail, zkLogin allows users to replace a transaction's signature from a public key (or several public keys), with a signature from an ephemeral public key tied to an SSO login.

An SSO login is witnessed as a signed token, which attests that some user (most likely an email address) has logged into some known OpenID "identity provider". (At the time of the audit, the identity providers supported by zkLogin were Google, Facebook, and Twitch).

The zkLogin application uses zero-knowledge technology to hide who the user really is. For example, that the user is **hello@zksecurity.xyz**.

OAuth 2.0 and JSON Web Tokens (JWTs)

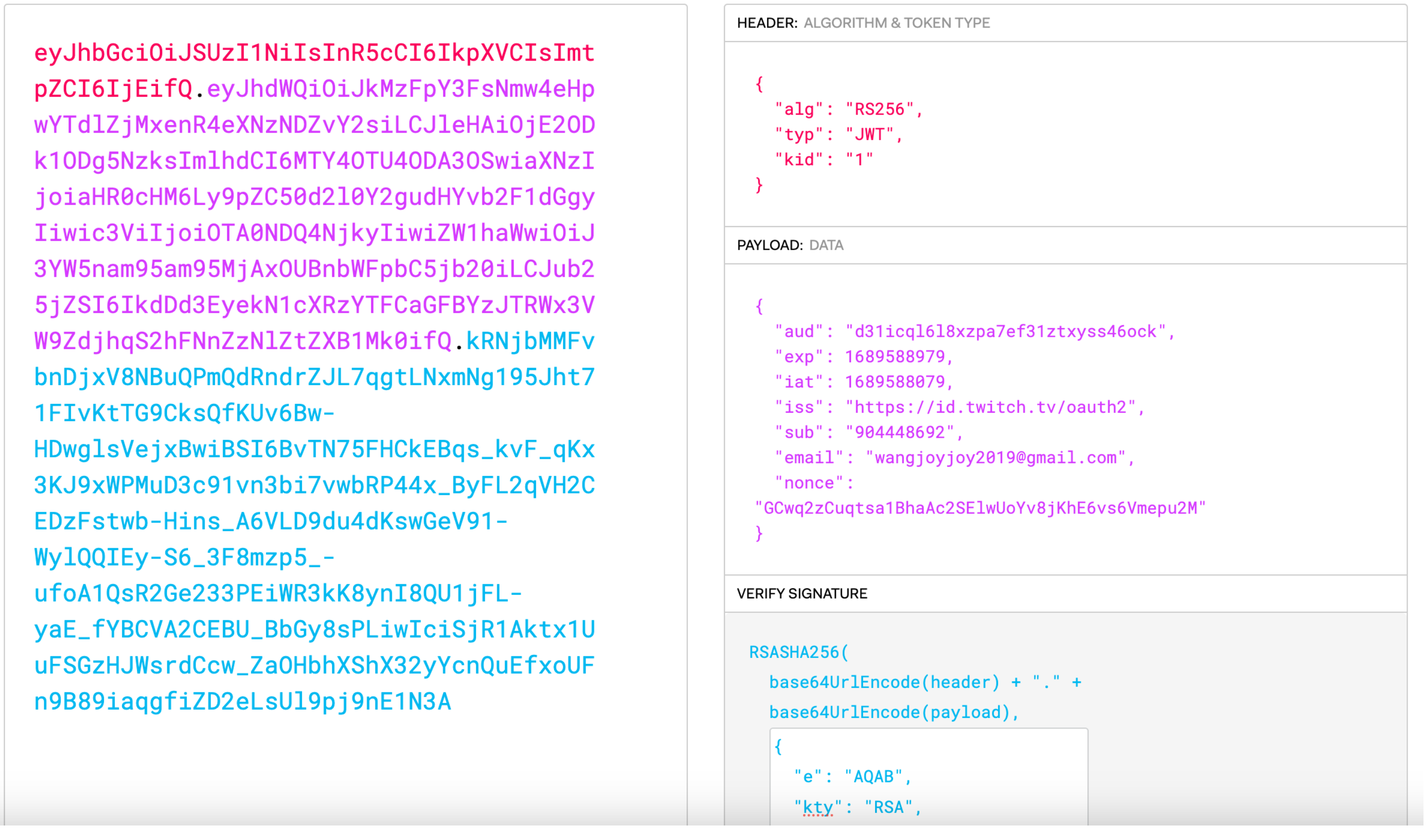
The Single Sign-On protocol supported by zkLogin is **OAuth 2.0**. In which a user can log into a trusted third party (Google, Facebook, etc.) and get a signed token attesting that they logged in in the form of a signed **JSON Web Token** (JWT).

A signed JWT looks like three base64-encoded payloads separated by a dot:

```
eyJhbGciOiJSUzI1NiIsImtpZCI6ImM5YWZkYTM2ODJlYmYwOWViMzA1NWMxYzRiZDM5Yjc1MWZiZjgxOTUiLCJ0eXAiOiJKV1QifQ.e
yJpc3MiOiJodHRwczovL2FjY291bnRzLmdvb2dsZS5jb20iLCJhenAiOiI1NzU1MTkyMDQyMzctbXNvcDllcDQ1dTJ1bzk4aGFwcW1uZ
3Y4ZDg0cWRjOGsuYXBwcy5nb29nbGV1c2VyY29udGVudC5jb20iLCJhdWQiOiI1NzU1MTkyMDQyMzctbXNvcDllcDQ1dTJ1bzk4aGFwc
W1uZ3Y4ZDg0cWRjOGsuYXBwcy5nb29nbGV1c2VyY29udGVudC5jb20iLCJzdWIiOiIxMTA0NjM0NTIxNjczMDM1OTgzODMiLCJlbWFpb
CI6IndhbmdxaWFveWkuam95QGdtYWlsLmNvbSIsImVtYWlsX3ZlcmlmaWVkIjp0cnVlLCJub25jZSI6IkpNaTZjXzNxWG4xSDhVWDVsY
TFQNllEd1Roa041TFp4cWFnVHlqZmlZd1UiLCJpYXQiOjE2ODMzMjMyNjksImV4cCI6MTY4MzMyNjg2OSwianRpIjoiMDEzMzA2YjY1M
mY0Zjg1MjUxZTU1OGVhNGFhOWJkYWI3ZmQxNzk3ZiJ9.TCI2XSbEmFc3KVHn2MoZi4OwCM56l59hiSZBdWwMeaCQWCbcJ87OhqtDTOuJ
MtUclBfkvEDoX_F2VhLJEbUcsFc5XyH_wrPjEqLet3uK1NB8Pqvuk1Q8lMy9nTvSCugGyCRUVhGiOiUfITwq8DhP-
NPQ_2vp0NVb_EmHEUxgRniNA-h5JXK2RRxKb1Sx0-yAnerxAamNcvYCOL679Ig9u0N_G_v2cwUVYEm-
```

```
8XkKpyrUeMv60euxMdO0LDCa1qbOj_l0OmPtMweCMGtVJOCrR3upZ443ttALJ2slsXdXc0Ee9byDoEP9KaPsvMT2ZQX3ZDss_ce3opYD
d0snUf-H8A
```

When decoded, the first payload is a header, the second is the JWT's content itself (called the payload), and the third is the signature. One can use the debugger on **jwt.io** to inspect such JWTs:



```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImt
pZCI6IjEifQ.eyJhdWQiOiJkMzFpY3FsNmw4eHp
wYTdlZjMxenR4eXNzNDZvY2siLCJleHAiOjE2OD
k1ODg5NzksImlhdCI6MTY4OTU4ODA3OSwiaXNzI
joiaHR0cHM6Ly9pZC50d2l0Y2gudHYvb2F1dGgy
Iiwic3ViIjoiOTA0NDQ4NjkyIiwiZW1haWwiOiJ
3YW5nam95am95MjAxOUBnbWFpbC5jb20iLCJub2
5jZSI6IkdDd3EyekN1cXRzYTFCaGFBYzJTRWx3V
W9ZdjhqS2hFNnZzNlZtZXB1Mk0ifQ.kRNjbMMFv
bnDjxV8NBuQPmQdRndrZJL7qgtLNxmNg195Jht7
1FIvKtTG9CksQfKUv6Bw-
HDwglsVejxBwiBSI6BvTN75FHCkEBqs_kvF_qKx
3KJ9xWPMuD3c91vn3bi7vwbRP44x_ByFL2qVH2C
EDzFstwb-Hins_A6VLD9du4dKswGeV91-
WylQQIEy-S6_3F8mzp5_-
ufoA1QsR2Ge233PEiWR3kK8ynI8QU1jFL-
yaE_fYBCVA2CEBU_BbGy8sPLiwIciSjR1Aktx1U
uFSGzHJWsrdCcw_ZaOHbhXShX32yYcnQuEfxoUF
n9B89iaqgfiZD2eLsUl9pj9nE1N3A
```

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "1"
}
```

PAYLOAD: DATA

```
{
  "aud": "d31icql6l8xzpa7ef31ztxyss46ock",
  "exp": 1689588979,
  "iat": 1689588079,
  "iss": "https://id.twitch.tv/oauth2",
  "sub": "904448692",
  "email": "wangjoyjoy2019@gmail.com",
  "nonce":
"GCwq2zCuqtsa1BhaAc2SElwUoYv8jKhE6vs6Vmepu2M"
}
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  {
    "e": "AQAB",
    "kty": "RSA",
```

There are a number of important fields to notice in the payload:

- the issuer `iss` field, indicates who issued/signed the JWT.
- the audience `aud` field, indicates who the JWT was meant for.
- the subject `sub` field, represents a unique user ID (from the point of view of the issuer) who the JWT is authenticating.
- the nonce `nonce` field, reflects a user nonce for application to prevent replay attacks.

Verifying JWTs

To verify a JWT, one needs to verify the signature over the JWT. To verify a signature over a JWT, one must know the public key of the issuer of the JWT.

Issuers all have a published JSON Web Key Set (JWKS). For example, Facebook's JWKS can be downloaded from https://www.facebook.com/.well-known/oauth/openid/jwks and looks like the following at the time of this writing:

```
{
  "keys": [
    {
            "kid": "5931701331165f07f550ac5f0194942d54f9c249",
            "kty": "RSA",
            "alg": "RS256",
            "use": "sig",
            "n": "-GuAIboTsRYNprJQOkdmuKXRx8ARnKXOC9Pajg4KxHHPt3OY8rXRmVeDxTj1-m9TfW6V-wJa_8ncBbbFE-aV-
eBi_XeuIToBBvLZp1-
UPIjitS8WCDrUhHiJnbvkIZf1B1YBIq_Ua81fzxhtjQ0jDftV2m5aavmJG4_94VG3Md7noQjjUKzxJyUNl4v_joMA6pIRCeeamvfIZor
jcR4wVf-wR8NiZjjRbcjKBpc7ztc7Gm778h34RSe9-
DLH6uicTROSYNa99pUwhn3XVfAv4hTFpLIcgHYadLZjsHfUvivr76uiYbxDZx6UTkK5jmi51b87u1b6iYmijDIMztzrIQ",
            "e": "AQAB"
        },
        {
            "kid": "a378585d826a933cc207ce31cad63c019a53095c",
            "kty": "RSA",
            "alg": "RS256",
            "use": "sig",
            "n": "1aLDAmRq-
QeOr1b8WbtpmD5D4CpE5S0YrNklM5BrRjuZ6FTG8AhqvyUUnAb7Dd1gCZgARbuk2yHOOca78JWX2ocAId9R4OV2PUoIYljDZ5gQJBaL6
liMpolQjlqovmd7IpF8XZWudWU6Rfhoh-j6dd-
8IHeJjBKMYij0CuA6HZ1L98vBW1ehEdnBZPfTe28H57hySzucnC1q1340h2E2cpCfLZ-vNoYQ4Qe-
CZKpUAKOoOlC4tWCt2rLcsV_vXvmNlLv_UYGbJEFKS-
I1tEwtlD71bvn9WWluE7L4pWlIolgzNyIz4yxe7G7V4jlvSSwsu1ZtIQzt5AtTC--5HEAyQ",
            "e": "AQAB"
        }
    ]
}
```

As you can see, a JWKS contains several Json Web Keys (JWKs) identified by their key id **`kid`**. Several keys are often displayed to provide support for key rotation (keeping the old one around for people to have the time to rotate their JWTs).

Because this information is external to the JWT, the network must know who the issuer is, and specifically what key ID **`kid`** was used to issue the JWT.

Note that As the issuer of a JWT is contained in the payload, not in the header, the zkLogin circuit must extract this value and witness it in its public input.

The zkLogin circuit

Finally, we can talk about what the zkLogin circuits fulfill at a high level.

Given the following public input:

- the issuer **`iss`** field that we expect the JWT to have (the user needs to pass that information along in the clear)
- the RSA public key of the issuer (for now only the RSA signature algorithm is supported, which is what seems to be the most widely used configuration in OAuth 2.0)

It extracts the following as public output:

- the ephemeral public key contained in the `nonce` field of the JWT, as well as some expiration information
- an "address seed", which is later used in the clear to derive the address of the user (see later on how this address seed is computed)
- the header of the JWT (which the network needs to validate, and also contains the key ID used by the issuer)
- the audience `aud` field of the JWT

The circuit, in addition to extracting the previously-discussed outputs, performs the following checks:

1. It inserts the actual JWT in the circuit as a private witness.
2. It checks that the issuer passed as public input is indeed the one contained in the JWT.
3. It hashes the JWT with SHA-256 and then verifies the signature (passed as private input) over the obtained digest using the issuer's public key (passed as public input).
4. It derives the address seed deterministically using the Poseidon hash function and the user identifier (e.g. an email) as well as some user randomness.

Note that the signature is verified in-circuit to avoid issuers from being able to track users on-chain via the signatures and digests.

The idea at this point is for the network to make sure that, besides the validity of the zero-knowledge proof, the address is strongly correlated to the user. To do that, validators deterministically derive the user's address based on the *address seed* output by the circuit, as well as the issuer and audience fields of the JWT. This information should be enough to uniquely identify the user.

Attack methodologies

Circom as a language prevents good abstractions and the creation of safe-to-use objects. Writing secure Circom code is not an easy task. Implicit contracts are frequently created between a template and its callers, leading to assumptions and preconditions that are not always explicit and have to be carefully audited throughout the callstack. Documentation is primordial for a Circom application to be secure.

During the engagement, it was found that the code was of high quality, with numerous comments on suppositions made by the various Circom templates, as well as tests for the various components.

This section describes the various attack methodologies that were used during the audit.

Attempt 1: Underconstrained RSA signature verification. Our first attempt at attacking the application was to look for underconstrain issues in the implementation of non-native arithmetic modulo an arbitrary RSA modulus. We could not find any issues there, and documented our review of the critical operations in the RSA's non-native arithmetic in zkLogin section.

Attempt 2: Underconstrained SHA-256 digest. The next idea is to see if we can decorrelate what's given to the hash function from what's obtained after hashing. As such, we could still get a valid signature (on a correct hash), but the JWT used in all the other checks would be different. We could not find any issues there as well.

Attempt 3: Faking the hash input. This is similar to the previous attack, but attempts to decorrelate the JWT payload that is being used in the application from the input that's being passed to the hash function. If this worked, we could use a valid JWT to hash and verify the signature, but use a malicious one to go through the main parsing logic of zkLogin. As with other attempts, this one failed as well.

Attempt 4: Decorrelate the JWT from what's being checked. If we could not fake the verification of the JWT, what about using a valid JWT but redirecting the checks to malicious data? The JWT payload is inserted in the circuit within a fixed-sized vector `` `padded_unsigned_jwt` `` that should be large enough for a wide range of JWTs (including some padding for the hash function). This means that there might be some room at the very end of the buffer which could contain arbitrary data that we could point to (instead of pointing to the actual JWT payload). This did not work as the `` `sha2wrapper()` `` template enforces that the end of the buffer is filled with zeros and can't be used to load arbitrary data.

Attempt 5: Attacks on the JWT parsing. The next step was to attempt pure attacks on the parsing logic. One idea was to try to break procedures that have tricky bit or byte-alignment offset edge-cases. For example, this shows up in both base64 decoding and efficient slicing with groups. However, the zkLogin codebase handles offsets properly with vector programming techniques which we have documented in the Vector Programming for offsets in zkLogin section of the report.

Attempt 6: Attacks on the public input packing. The public input of the circuit packs all the public inputs of the circuit by hashing them (with the Poseidon hash) and witnessing a single public digest. Some of the inputs are themselves (Poseidon) hashes of longer input vectors (for example, `` `modulus_F` `` is a hash of the whole modulus so as to fit it in a single field element). Hashing without padding, or with a bad padding, could lead to collisions which would have allowed us to provide malicious inputs in the circuit (instead of the ones that were packed). For example, providing a bigger or smaller modulus could introduce more factors which would break RSA. This approach did not work overall because of all the hashing being done on fixed-size inputs.

Attempt 7: General misuse of libraries. As stated in the introduction to this section, Circom lacks good abstractions which makes it hard to write safe-to-use libraries. Circomlib, the main standard library of Circom, is no exception. We reviewed for general issues that can arise from not following the implicit contracts of the library's templates. For example, the `LessThan` template of the library is only safe to use when the input is bound to be less than half of the prime modulus of the underlying field. No issues were found.

RSA's non-native arithmetic in zkLogin

zkLogin's circuit implementation of the RSA signature verification must perform modular arithmetic with large numbers. Specifically, a modular multiplication is implemented to support a large modulus of 2048 bits. This section reviews the algorithm used by the implementation.

Problem statement

The goal of the circuit is to compute $a \cdot b \mod p$ but in a different field native (where native is the native field).

In the non-negative integers, we know that there exists q and $r < p$ such that:

$$ab = pq + r$$

When $p < \text{native}$ we have to ensure that $pq + r$ doesn't overflow modulo native. In our case though, we have that $p > \text{native}$, so we have to split any element modulo p in k limbs of n bits (and enforce things in this representation) so that they can fit in our circuit.

Computing the multiplication

We can interpret the k limbs of n-bit of a as the following polynomials:

$$a(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_{k-1}x^{k-1}$$

and similarly for $b(x)$. Note that the value a is the polynomial evaluated with the basis: $a = a(2^n)$ (same for b).

Notice that the polynomial $ab(x) = a(x) \cdot b(x)$ represents the "unreduced" multiplication $a \cdot b$, where coefficients (limbs) can reach $2n + \epsilon$ bits (instead of n bits due to the coefficient of x^{k-1} that looks like $a_{k-1}b_0 + a_{k-2}b_1 + \cdots + a_1b_{k-2} + a_0b_{k-1}$).

Note also that this polynomial is of degree $2(k-1)$ (this will be important later).

Computing the modular reduction

To do the modular reduction with the non-native field we witness r and q as k limbs of n bits as well.

We then look at the polynomial $pqr(x) = p(x) \cdot q(x) + r(x)$ where (remember) p is fixed as the non-native field. Note also that this polynomial is of degree $2(k-1)$.

Now we need to prove that this $pqr(x)$ polynomial is equivalent to the polynomial $ab(x)$. We can do this by showing that they match on enough points. That is, that $t(x) = ab(x) - pqr(x)$ is 0 on enough points. (Note that if we had access to randomness, we could have used the Schwartz-Zippel lemma to test this equality with a single random evaluation.)

Enough point is $d + 1$ points, where d is the max degree of $pqr(x)$ and $ab(x)$ (which is $2(k-1)$ as we've noted above). So we need $2(k-1)+1 = 2k-1$ evaluation points. The implementation does this for this many `x`s, taken in the range $[[0, 2k-2]]$:

```
// create enough evaluations for pqr(x)
signal v_pq_r[2*k-1];
for (var x = 0; x < 2*k-1; x++) {
    var v_p = poly_eval(k, p, x);
    var v_q = poly_eval(k, q, x);
    var v_r = poly_eval(k, r, x);
    v_pq_r[x] <== v_p * v_q + v_r;
}

// create enough evaluations for t(x) = ab(x) - pqr(x)
signal v_t[2*k-1];
for (var x = 0; x < 2*k-1; x++) {
    v_t[x] <== v_ab[x] - v_pq_r[x];
}
```

Checking that t represents zero

At this point, the circuit interpolates the polynomial t from its values.

But limbs of the encoded value $a \cdot b$ (coefficients of $ab(x)$) and $pq + r$ (coefficients of $pqr(x)$) are not necessarily the same (since their coefficients can overflow).

In addition, we don't necessarily have that $ab_i > pqr_i$. Due to this, the subtraction might underflow. The following shows that this doesn't matter as long as the bigint representation is 0 (i.e. $\sum_i t_i \cdot 2^{n \cdot i} = 0$)

First, if $ab(x)$ and $pqr(x)$ are indeed the same polynomials (mas o menos the overflows) then we should have that the first limbs agree in the first n bits:

$$t_0 = ab_0 - pqr_0 = 0 + 2^n(c_{ab,i} - c_{pqr,i}) = 2^n c_{t,i}$$

We now have proven that the next carry is $c_{t,i} = c_{ab,i} - c_{pqr,i}$ (the overflow of both limbs) for the base case. Remains to show that it is true for i to get induction. Let's assume that $c_{t,i-1} = c_{ab,i-1} - c_{pqr,i-1}$, then we have that the next carry is also correctly constructed (if the limbs agree on the first 64 bits):

$$\begin{aligned} t_i &= ab_i - pqr_i + c_{t,i-1} \\ &= ab_i - pqr_i + c_{ab,i-1} - c_{pqr,i-1} \\ &= (ab_i + c_{ab,i-1}) - (pqr_i - c_{pqr,i-1}) \\ &= (0 + 2^n c_{ab,i}) - (0 + 2^n c_{pqr,i}) \\ &= 0 + 2^n(c_{ab,i} - c_{pqr,i}) \end{aligned}$$

Induction follows, and at the very end, we should have that if the final carry is zero, then $t(2^n) = 0$.

The previous equations should have no soundness issues as long as what we've shown to be true in the integers is also true modulo our circuit field.

For that to happen, we want to enforce that for each limb $t_i = 0 + 2^n c_i$ we have that c_i is correctly bound not to wrap around in our circuit field. That is, we know that for some l:

$$c_i \in [-l, l] \iff l + c_i \in [0, 2l]$$

so the bit-constrain that they perform on the carry is enough if l is computed correctly.

Vector Programming for offsets in zkLogin

Handling offsets is tricky and a common source of bugs even in a traditional codebase. But zk-circuit programming is truly circuit-like: Arbitrary jumps are inefficient (with circom against the Groth16 backend anyway). This incentivizes working with a vector programming-mindset.

Vector programming is a paradigm where operations are applied to whole arrays or vectors, rather than element-by-element. This style of programming draws heavily from the principles of linear algebra, with the intent of providing optimized and parallelizable implementations. These optimizations traditionally were leveraged since they are effective for data-parallel problems, where the same operation must be performed on all elements in a data set. However, working in this paradigm also ends up saving constraints most of the time, ultimately leading to more efficient zk circuits.

Below we show two examples that use vectorization to handle edge-cases and offsets within parsing code to illustrate what offsets show up and how zkLogin handles these edge cases.

Vector Offset Example 1: Searching for ASCII Substrings inside a Base64-encoded string

Throughout the zkLogin algorithm, information must be extracted from a base64-encoded string presented within the JWT. While this extraction is fairly straight-forward outside of a zkSNARK circuit, the mechanism used within the zkLogin codebase within the zk circuit can be broken down into checking if an ASCII substring exists within a Base64-encoded string and then slicing such string into chunks. The ASCII-substring-within-base64 existence is quite interesting and is highlighted here for informational purposes.

To follow along, the **`` `template ASCIISubstrExistsInB64` ``** is defined in **`` `circuits/strings.circom:269` ``**.

Before we dig into precisely what the circom component is doing, let's first define ASCII and Base64-encodings in more detail. Note that we are considering the URL-safe base64 variant for the purposes of this analysis.

ASCII Encoding. At its core, ASCII (American Standard Code for Information Interchange) encoding is a scheme designed to map 128 specific characters, including control and printable characters, onto 7-bit integers. The characters range from 'a' to 'z', 'A' to 'Z', '0' to '9', along with various punctuation symbols, control characters, and other characters like space. In a computational context, the ASCII character set is often extended to 8 bits, allowing for 256 characters. This is not strictly ASCII, but rather an extended ASCII set, providing additional characters such as accented letters used in various languages.

The ASCII Table of Mapping. The ASCII table provides a critical mapping from 7-bit integers to characters in the ASCII character set. The table is limited to 128 unique mappings because ASCII uses 7 bits to represent each character.

Decimal	Char	Binary	Decimal	Char	Binary
32	' '	0100000	65	'A'	1000001
33	'!'	0100001	66	'B'	1000010

Decimal	Char	Binary	Decimal	Char	Binary
34	'"'	0100010	67	'C'	1000011
35	'#'	0100011	68	'D'	1000100
36	'$'	0100100	69	'E'	1000101
37	'%'	0100101	70	'F'	1000110
38	'&'	0100110	71	'G'	1000111
39	'''	0100111	72	'H'	1001000
40	'('	0101000	73	'I'	1001001
41	')'	0101001	74	'J'	1001010
42	'*'	0101010	75	'K'	1001011
43	'+'	0101011	76	'L'	1001100
44	','	0101100	77	'M'	1001101
45	'-'	0101101	78	'N'	1001110
46	'.'	0101110	79	'O'	1001111

... and so on, up to 127.

As mentioned above, we can treat ASCII characters as raw 8-bit bytes, and we'll do so in this algorithm for simplicity.

Base64 Encoding. In essence, base64 encoding is a mechanism to map binary data, generally arbitrarily sized, onto a specific set of 64 characters (`` `A-Z` ``, `` `a-z` ``, `` `0-9` ``, `` `-` ``, `` `_` ``).

The Base64 Table of Mapping. At the core of base64 encoding is the lookup table. It maps 6-bit integers to characters in the base64 alphabet. As one might anticipate, the set of characters for this purpose is restricted to 64 in number.

Decimal	Char	Decimal	Char	Decimal	Char	Decimal	Char
0	`` `A` ``	16	`` `Q` ``	32	`` `g` ``	48	`` `w` ``
1	`` `B` ``	17	`` `R` ``	33	`` `h` ``	49	`` `x` ``
2	`` `C` ``	18	`` `S` ``	34	`` `i` ``	50	`` `y` ``
3	`` `D` ``	19	`` `T` ``	35	`` `j` ``	51	`` `z` ``
4	`` `E` ``	20	`` `U` ``	36	`` `k` ``	52	`` `0` ``
5	`` `F` ``	21	`` `V` ``	37	`` `l` ``	53	`` `1` ``
6	`` `G` ``	22	`` `W` ``	38	`` `m` ``	54	`` `2` ``

Decimal	Char	Decimal	Char	Decimal	Char	Decimal	Char
7	`H`	23	`X`	39	`n`	55	`3`
8	`I`	24	`Y`	40	`o`	56	`4`
9	`J`	25	`Z`	41	`p`	57	`5`
10	`K`	26	`a`	42	`q`	58	`6`
11	`L`	27	`b`	43	`r`	59	`7`
12	`M`	28	`c`	44	`s`	60	`8`
13	`N`	29	`d`	45	`t`	61	`9`
14	`O`	30	`e`	46	`u`	62	`-`
15	`P`	31	`f`	47	`v`	63	`_`

Base64 Semantics. The base64 encoding operation can be decomposed into several logical steps:

1. Firstly, bytes from the binary data are grouped into chunks of three bytes (or 24 bits).
2. Each 24-bit chunk is divided into four 6-bit integers.
3. Each 6-bit integer is mapped to its corresponding base64 alphabet character. The mapping is governed by the table above.
4. Finally, the resulting characters are concatenated to form the output base64 encoded string.

Note that the length (`L`) of the encoded base64 string should be approximately $(\frac{4}{3})$ the length of the original binary data, rounded up to the nearest multiple of 4 due to padding.

Algorithm. At a high-level, the algorithm used within this circom component is:

1. Break apart the relevant chunk of base64 data into bits.
2. Break apart the ASCII needle into bits.
3. Verify that the bits match taking care to handle offsets, see below for more details.

ASCII bit decoding is straightforward.

Base64 bit decoding, delegated to `template DecodeBase64URLChar` at `circuits/base64.circom:37`, computes the base64 character in the prover, and then constrains the result to different contiguous sections of characters (`A-Z`, `a-z`, `0-9`, `-`, `_`).

Offsets. Since base64-encoded characters map to 6-bit values and are packed tightly, but ASCII values map to 8-bit ones. It's possible that data doesn't line up perfectly, and so it must be offset.

There are 3 possibilities which cycle between each other as the more data is packed together.

6-bit groups	8-bit groups	Offset
2n	n	4

6-bit groups	8-bit groups	Offset
3n	2n	2
4n	3n	0

Handling all offsets simultaneously. Ultimately, `` `ASCIISubstrExistsInB64` `` computes the bit-offset of the data (ensuring it's one of the valid cases 0-bit, 2-bit, 4-bit) and then encodes potential constraints to verify that the bits match at *all the offsets*, and uses the correct bit-offset as a mask to only selectively create the real constraints (using `` `AssertEqualIfEnabled` ``).

Conclusion. We have illustrated how base64 decoding works in general and how the zkLogin code uses vector programming to handle the bit alignment offset that arises. Moreover, we argued that the implementation is correct due to handling all possible offsets.

Vector Offset Example 2: Efficient Slicing With Groups

Within the zkLogin circuits, it's possible to reuse the same circuit component, a Slice component, to interrogate parts of SHA2 padding, remove the header from a JWT, and several times to extract claim strings from a JWT (both in and out of a base64 context).

Basic Slice. The Slice operation is straightforward; a simple direct implementation of `` `template Slice` `` can be found in `` `circuits/strings.circom:58` `` that extracts a possibly variably-length subarray starting at an index within a bigger array. The circom component does a few bounds checks around the index and the length as well.

A direct implementation of slice has costs dominated by an `` `n*m` `` term for `` `n=subarray_length` `` and `` `m=bigarray_length` ``.

Grouped Slice. In addition to special casing for a slice at the start of the bigarray which zkLogin does at `` `template SliceFromStart` `` which is also much cheaper, it's also possible to speed up the general Slice for even smaller subarrays and moderately sized bigger arrays.

In `` `template SliceGroup` ``, the circuit first packs together elements of both the larger and subarray (in practice it packs an array of bytes into 16 bytes into each field element).

Then it computes adjusted start and end indices due to the packing. This can be done with an efficient bitshift when the grouping is a power two as the adjustment is just a bitshift. Then the elements are ungrouped.

Finally, offsets are handled which will be described in more detail below.

Assuming a grouping of 16, the implementation's cost is instead dominated by an `` `18m + (n*m)/32` `` term, in practice much cheaper!

Offsets. Say we group 16 elements at a time, in that case there are 16 possible offsets to handle, 0-15 inclusive. Here we again create *all solutions simultaneously*! Specifically, we produce a 2D array of elements where one dimension is one of the offsets, and the other is the result shifted by the offset.

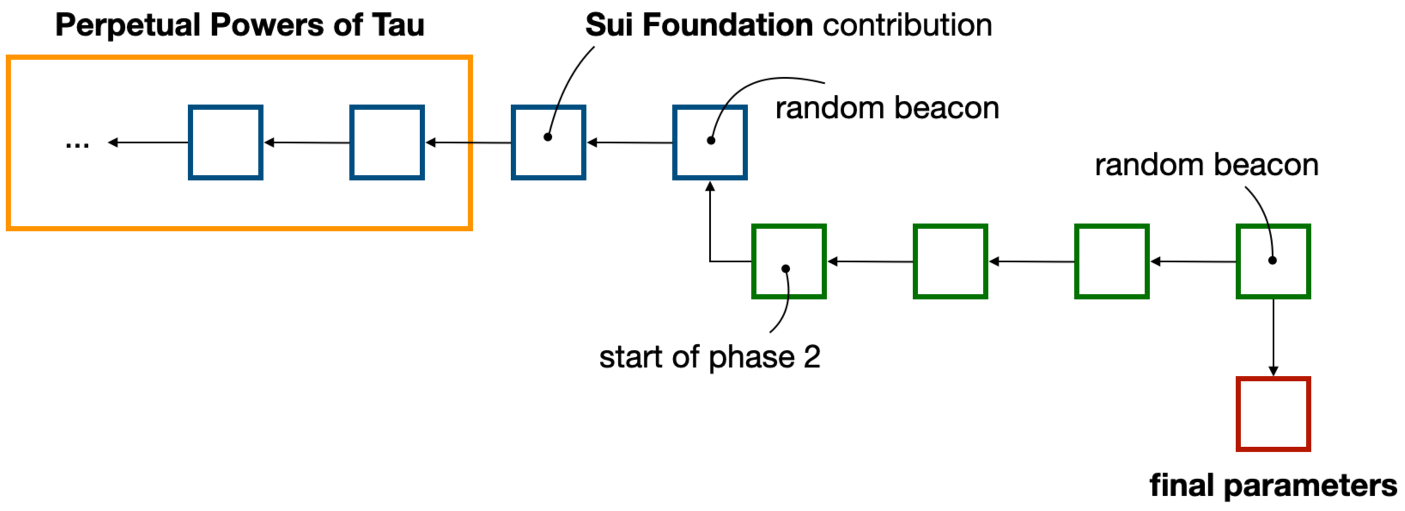
The correct offset for this case is selected using a standard circom multiplexer component by examining the remainder after dividing the start index by the number of elements in a group.

Conclusion. We discussed slicing in general and the group slicing efficiency change. Group slicing is a safe optimization that zkLogin performs and there are no offsets unhandled. Vectorization ensures that the operation is performed effectively within the circuit.

zkLogin Ceremony

As zkLogin is built on top of the Groth16 proof system, it relies on a set of parameters produced as part of a trusted setup. To provide assurance that the trusted setup was performed in a secure manner, best practice is to decentralize that process with a multi-party computation.

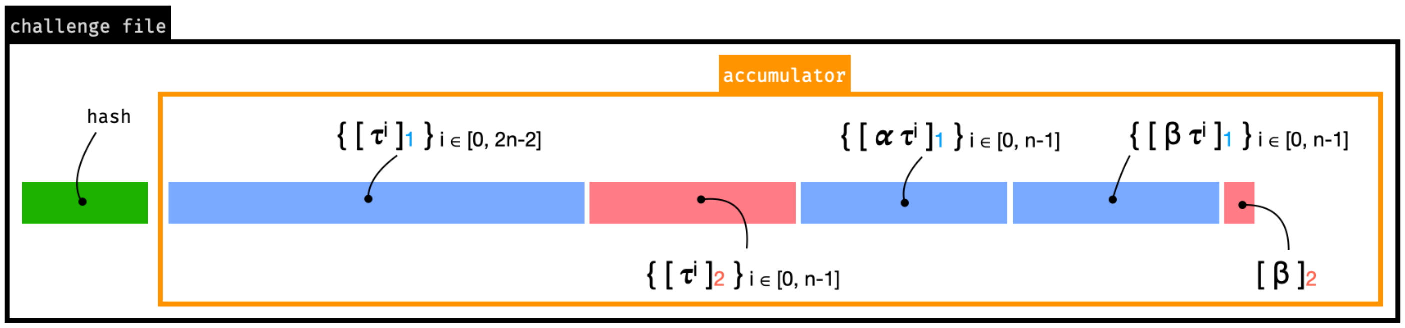
zkLogin follows this pattern, by reusing the well-known Perpetual Powers of Tau ceremony. The Perpetual Powers of Tau ceremony is a linear multi-party computation, instantiating the MMORPG paper, where participants can continuously join the protocol (one-by-one) to add their own randomness. As long as a single participant was honest in following the protocol (i.e. by destroying their own randomness after the fact), the protocol is secure.



The usual way to make use of these public parameters is to join the chain of contribution (so-called phase 1), and to then fork it to specialize it on specific circuits (in a phase 2). This is because Groth16 is not a universal proof system, and as such the public parameters offered by the Perpertual Powers of Tau can't be used as is.

phase2-bn254's serialization

The parameters of the first phase are configured to work for circuits up to 2^{28} constraints. The figure below shows how the parameters are serialized.


challenge file
accumulator
hash
$\{[\tau^i]_1\}_{i \in [0, 2n-2]}$
$\{[\alpha\tau^i]_1\}_{i \in [0, n-1]}$
$\{[\beta\tau^i]_1\}_{i \in [0, n-1]}$
$\{[\tau^i]_2\}_{i \in [0, n-1]}$
$[\beta]_2$

Smaller circuits can "trim" the parameters as they don't need its full length. To do that, they can simply trim each of the sections in the figure above.

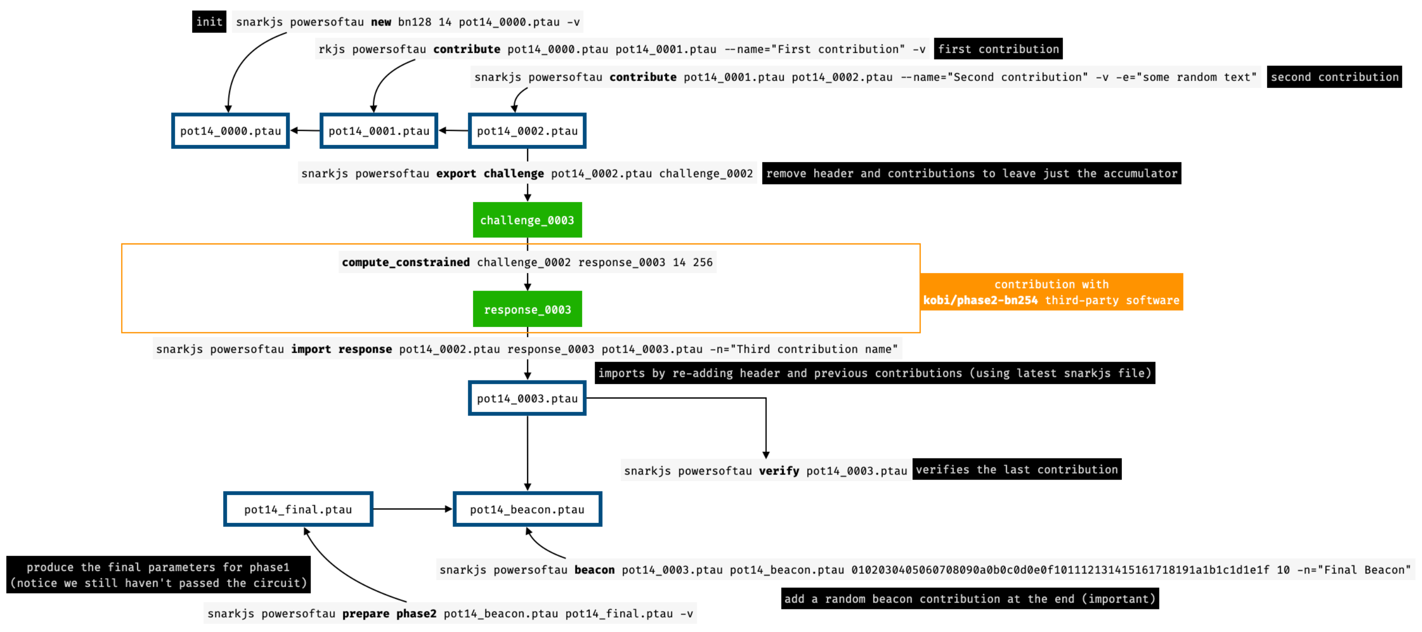
The `kobigurk/phase2-bn254` tool was patched in https://github.com/MystenLabs/phase2-bn254 to support this trimming (which was broken before the patch). We reviewed the patch and found it to be correct. Essentially, the issue was that the code was deserializing the full-length parameters (for `original_circuit_power`) using a trimmed description of the parameters (`reduced_circuit_power`). The important line that fixed the deserialization was:

```
-    let parameters = CeremonyParams::<Bn256>::new(reduced_circuit_power, batch_size);
+    let parameters = CeremonyParams::<Bn256>::new(original_circuit_power, batch_size);
```

Re-importing response files in snarkjs

Snarkjs allows exporting the latest state so that other tools (like `kobigurk/phase2-bn254`) can make use of it (for example, to contribute or to trim the parameters as discussed previously).

The typical flow is summarized in the figure below.

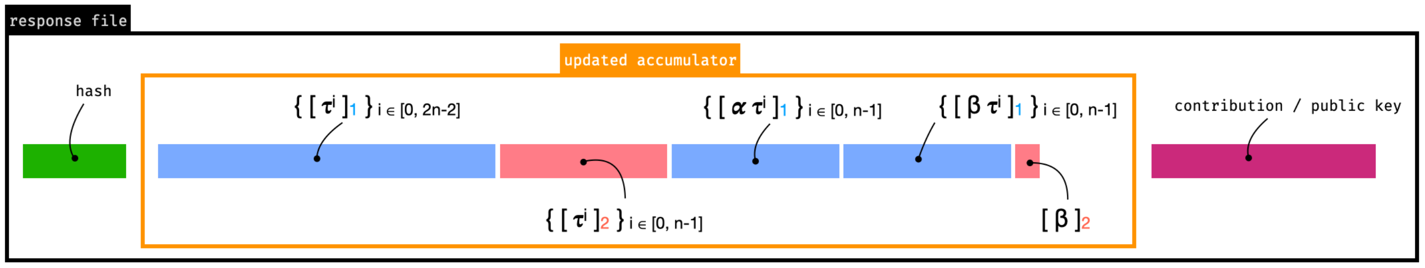


```
snarkjs powersoftau new bn128 14 pot14_0000.ptau -v
rkjs powersoftau contribute pot14_0000.ptau pot14_0001.ptau --name="First contribution" -v
snarkjs powersoftau contribute pot14_0001.ptau pot14_0002.ptau --name="Second contribution" -v -e="some random text"
snarkjs powersoftau export challenge pot14_0002.ptau challenge_0002
compute_constrained challenge_0002 response_0003 14 256
snarkjs powersoftau import response pot14_0002.ptau response_0003 pot14_0003.ptau -n="Third contribution name"
snarkjs powersoftau verify pot14_0003.ptau
snarkjs powersoftau beacon pot14_0003.ptau pot14_beacon.ptau 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f 10 -n="Final Beacon"
snarkjs powersoftau prepare phase2 pot14_beacon.ptau pot14_final.ptau -v
```

init
first contribution
second contribution
pot14_0000.ptau
pot14_0001.ptau
pot14_0002.ptau
remove header and contributions to leave just the accumulator
challenge_0003
contribution with kobi/phase2-bn254 third-party software
response_0003
imports by re-adding header and previous contributions (using latest snarkjs file)
pot14_0003.ptau
verifies the last contribution
pot14_final.ptau
pot14_beacon.ptau
produce the final parameters for phase1 (notice we still haven't passed the circuit)
add a random beacon contribution at the end (important)

Exporting with snarkjs works by removing the header as well as the history of contributions. What is left is a hash and the latest state of the ceremony (called accumulator in the figure above). The exported file is called a "challenge".

Importing with snarkjs works by taking a "response" file, which is similar to a challenge file, but with the latest contribution appended to it (see figure below). As snarkjs keeps track of the history of contributions, the latest snarkjs state (on which the last export was performed) is also used to recover that history.



response file
updated accumulator
hash
$\{ [\tau^i]_1 \}_{i \in [0, 2n-2]}$
$\{ [\alpha \tau^i]_1 \}_{i \in [0, n-1]}$
$\{ [\beta \tau^i]_1 \}_{i \in [0, n-1]}$
contribution / public key
$\{ [\tau^i]_2 \}_{i \in [0, n-1]}$
$[\beta]_2$

The problem fixed by Mysten Lab's patch is that snarkjs verifies consistency of the prepended hash when reimporting, which doesn't always interoperate nicely with external tools.

Specifically, `` `kobigurk/phase2-bn254` `` computes that prepended hash as `` `HASH(challenge)` ``, which when called several times becomes a digest based on a challenge unknown to snarkjs.

Mysten Lab's fix is to remove this check. We found it to be an acceptable fix as the check is there for sanity check, and a well-coordinated ceremony should not be affected by this change.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
1	circuits/helpers/bigint_func.circom	log_ceil() produces incorrect results	Low
2	circuits/helpers/jwtchecks.circom	Underconstrained bit decomposition of nonce	Low
3	circuits/helpers/rsa/bigint.circom	Incorrect bounds for RSA signature verification	Low
5	test/*.test.js	Use standard JS tooling to improve code hygiene	Informational
9	circuits/helpers/hasher.circom	Redundant bit decomposition	Informational
12	circuits/*	Too many arguments	Informational

# 1 - log_ceil() produces incorrect results

- circuits/helpers/bigint_func.circom

Low

Description. The `` `log_ceil()` `` helper function is used to determine the maximum number of bits that certain expressions can have. Expressions arise in the bigint multiplication code and are of the form $S = \sum_{i=1}^{k} a_i$ where a_i are known to be n-bit integers. To determine the max number of bits of S, we want to estimate

$$S = \sum_{i=1}^{k} a_i \leq k \cdot (2^n - 1) < 2^m$$

Finding the smallest m on the right hand side will give us a precise upper bound on the number of bits that S needs. Assuming we knew that $k \leq 2^K$, we could infer

$$k \cdot (2^n - 1) < 2^{n+K}$$

so we find $m = n + K$. Choosing the minimal K will make m minimal as well. Note here that the inequality on k is not strict -- it's fine if $k = 2^K$, thanks to the -1 in the other factor.

Taking logs, we get the condition

$$\log_2(k) \leq K$$

and so the minimal K can be written as $K = \lceil \log_2(k) \rceil$. The maximum number of bits of S is precisely $n + \lceil \log_2(k) \rceil$. This motivates the use of a `` `log_ceil()` `` function for this problem.

The `` `log_ceil()` `` function at the time of the audit was implementing something slightly different, namely, it was computing the *number of bits* of k.

Here are some examples:

Input k	**Input (Binary)**	**Expected output `` `log_ceil(k)` ``**	**Actual output = number of bits**
1	1	0	1
2	10	1	2
3	11	2	2
4	100	2	3

The difference is for k a power of two, when the actual implementation returns something too large by 1.

Recommendations. Because of the specific use as an upper bound, returning something too large is no security problem, it's more conservative than necessary. However, we recommend fixing **`` `log_ceil()` ``**, first because off-by-one issues could become a problem when used in other circumstances, and second because using more bits than necessary implies using more constraints than necessary.

Furthermore, at the time of audit **`` `log_ceil()` ``** defaults to 254 for any input larger than 254 bits. Again this is fine because inputs that large are not used in practice. We recommend adding an assertion preventing this case from being exercised.

# 2 - Underconstrained bit decomposition of nonce

● circuits/helpers/jwtchecks.circom

Low

Description. The `NonceChecker` template takes two input values and checks that they represent the same nonce:

- `expected_nonce` is a field element that was derived in the circuit as a Poseidon hash of ephemeral public key, max epoch and randomness.
- `actual_nonce` is the base64 nonce value which was extracted from the JWT

This check is where the public key and max epoch are linked to the signed JWT. The goal is to check that the least significant 160 bits of `expected_nonce` are equal to the base64-decoded `actual_nonce`.

The template first decomposes `expected_nonce` into 256 bits:

```
signal input expected_nonce;
signal expected_bits[256] <== Num2BitsBE(256)(expected_nonce);
```

Under the hood, `Num2BitsBE` establishes that the `expected_nonce` n is related to the `expected_bits` $(\mathrm{b_i})$ by asserting that

$$\mathrm{n} == \sum_{\mathrm{i}=0}^{255} \mathrm{b_i} 2^{\mathrm{i}} \tag{1}$$

The issue is that this check doesn't fully constrain the `expected_bits` $\mathrm{b_i}$ to unique values.

- First, note that the zk proof is over the BN254 curve and the native field size p has 254 bits (more precisely, $\mathrm{p} \approx 1.5 \cdot 2^{253}$).
- Second, the assertion (1) is an equality modulo p. Therefore, if we let $(\mathrm{b_i})$ be the bits of $\mathrm{n} + \mathrm{kp}$ instead of n, for any $\mathrm{k} > 0$ such that $\mathrm{n} + \mathrm{kp}$ fits in 256 bits, the check still succeeds. This works for $\mathrm{k} = 0, \ldots, 4$ and possibly $\mathrm{k} = 5$, depending on how large $\mathrm{n} \in [0, \mathrm{p})$ is.

Recommendations. The implication of underconstraining the nonce bits is that the prover gains the freedom to use either of the 6 values $\mathrm{n} + \mathrm{kp}$ (truncated to 160 bits) as `actual_nonce`. They can put this modified nonce in the JWT when it is signed by the OAuth provider. When viewed in the context of the entire protocol, this is deemed not a security risk.

Nevertheless, we recommend to adapt the bit decomposition so that the nonce value is fully constrained to follow the original protocol.

A possible remedy is decomposing into fewer bits. However, anything less than 254 bits would destroy completeness, since the nonce can be any field element. With exactly 254 bits there is still the possibility that $\mathrm{n} + \mathrm{p} < 2^{254}$ and the bits aren't unique.

The recommendation is to use 254 bits and also add a dedicated check which makes sure that the bit-decomposed value is smaller than p.

# 3 - Incorrect bounds for RSA signature verification

● circuits/helpers/rsa/bigint.circom

Low

Description. Following the RSA's non-native arithmetic in zkLogin section, it is important that the non-native arithmetic circuit correctly constrains the carry of each limb of the t polynomial. If the check is too large, there could be soudness issues, if too small, there could be completeness issues.

We found that the range is a bit too large, which is most likely not a big problem as it should still be enough to prevent our equations from wrapping around in our circuit field.

In the code, carries are constrained with an $l = 2^{m+\epsilon-n-1}$ as can be seen:

```
carryRangeChecks[i] = Num2Bits(m + EPSILON - n);

// TRUNCATED...

carryRangeChecks[i].in <== carry[i] + ( 1<< (m + EPSILON - n - 1));
```

Combined together, the snippet enforces that for all i (except for the last limb) we have $c_i + l \in [0, 2l[$.

Specifically, for the parameters used by zkLogin (`` `n = 64, EPSILON = 3, k = 32, m = n + n + log_ceil(k) + 2 = 135` ``) we have that $l = 2^{73}$.

This value is too large. To see why, let's compute l manually.

Notice that if all coefficients take their maximum potential value, the biggest coefficient will look like $\sum_{i=0}^{k-1}(2^n - 1)(2^n - 1) = k(2^{2n} - 2^{n+1} + 2)$. If we write $k = 2^{\lfloor \log(k) \rfloor} + r_k$ then we have that the sum is upperbounded by $2^{2n+\lceil \log(k) \rceil}$, which is $133 = 64 + 69$ if $n = 64$ and $k = 32$. So the carry should be 69 bits at most.

We can also verify this with code, using SageMath we also get that the closest power of 2 above the value of l is 2^{69}:

```
# params
k = 32
n = 64
maxval = 2^n - 1
R = PolynomialRing(ZZ, 'x')

# find largest coefficient in ab_poly and return its bit-size
def upperbound(poly):
    ab_poly_coeffs = ab_poly.coefficients(sparse=False)
    ab_poly_coeffs.sort()
    ab_poly_coeffs.reverse()
    res = bin(ab_poly_coeffs[0] >> 64)[2:] # remove "0b" prefix
    print(res)
    return len(res)

# calculate upperbound (since we do ab - pqr, then upperbound of carry is determined on largest limb
that has the pqr carry set to 0)
a_poly = R([maxval] * k)
b_poly = R([maxval] * k)
ab_poly = a_poly * b_poly
print("log2 upperbound:", upperbound(ab_poly))

# caculate lowerbound (ab = 0, pqr is the highest value)
# highest value of pqr is? p * q + r
p = R([maxval] * k)
q = R([maxval] * k)
r = R([maxval] * k)
pqr_poly = p * q + r
print("log2 lowerbound:", upperbound(pqr_poly))
```

Recommendations. Document the calculation involved in the bit-constraints of the carries. Ensure that the right number is used.

# 5 - Use standard JS tooling to improve code hygiene

● test/*.test.js

Informational

Description. The JS code in `test/*.test.js` uses code patterns which are usually discouraged, such as assignment of variables that were never declared with `var` or `let` (which assigns the variable to a property on the global object, and is forbidden in "strict" mode).

We recommend using a linter such as `eslint` to warn about issues like this. We also recommend the `eslint-plugin-mocha` plugin to catch common errors specific to using mocha (the test framework):

```
// package.json
"devDependencies": {
    // ...
+   "eslint": "^8.48.0",
+   "eslint-plugin-mocha": "^10.1.0",
  }
```

This is a linter config we found to be a good starting point:

```
// .eslintrc.js
module.exports = {
  env: { node: true, commonjs: true, es2021: true },
  extends: ["eslint:recommended", "plugin:mocha/recommended"],
  parserOptions: { ecmaVersion: "latest" },
  rules: {
    "mocha/no-mocha-arrows": "off",
    "no-unused-vars": "warn",
  },
};
```

Using this config, numerous issues are reported by the linter. We found none that made the code as used incorrect, but some that could easily lead to errors in the future. For example, one of the tests passes an `async` function to mocha's `describe` test runner:

```
describe(provider, async function () {
  // ...
});
```

The mocha linter plugin treats this as an error, because `describe` blocks are executed synchronously to collect all tests before running them. So, as soon as an actual promise would be awaited in that code block, none of the tests after the `await` would be collected, but this would easily stay undetected as the test would continue to report success even if 0 tests were run.

Apart from using a linter, we also recommend installing an auto-formatter like `prettier`, simply to make the code more readable and consistent with what JS developers are used to.

# 9 - Redundant bit decomposition

● circuits/helpers/hasher.circom

Informational

Description. In `HashBytesToField` (`hasher.circom`, line 107), we take a byte array (the input) and pack it into an array of 248-bit chunks, for the purpose of efficiently hashing with Poseidon. This conversion is performed by `ConvertBase`, which first expands / decomposes the inputs from 8-bit to 8x 1-bit, and then compresses / packs the resulting bit array by summing 248 bits each. Unpacking has to constrain each bit to 0 or 1 which in total takes `total_number_of_bits = inCount * 8` constraints. For example, this is applied to the header which has a max size of 500 bytes, so in this instance `ConvertBase` uses about $500 \cdot 8 = 4000$ constraints.

Using the knowledge that both input and output widths are multiples of 8, we can save most of those constraints. Instead of unpacking into bits and packing into 248-bit values, we can directly pack the input bytes into 31-byte field elements. This only takes about `outCount = ceil(inCount / 31)` constraints. In the header example, this amounts to as few as $\lceil 500/31 \rceil = 17$ constraints.

It is safe to apply this change when the input bytes are already constrained to fit within 8 bits elsewhere. This is, in fact, the case for all places where `HashBytesToField` is used, as we will now explain.

First, let's look at the JWT header:

```
// zkLoginMain.circom, line 90
var header_length = payload_start_index - 1;
signal header[maxHeaderLen] <== SliceFromStart(inCount, maxHeaderLen)(
    padded_unsigned_jwt, header_length
);
signal header_F <== HashBytesToField(maxHeaderLen)(header);
```

As we can see, `header` is obtained by slicing from the start of the main JWT string `padded_unsigned_jwt`, before being passed into `HashBytesToField`. The elements of `padded_unsigned_jwt` are constrained to fit in 8 bits inside `Sha2_wrapper`:

```
// sha256.circom, line 144
sha256_blocks[b][s] = Num2BitsBE(inWidth);   // inWidth = 8
sha256_blocks[b][s].in <== in[payloadIndex]; // in = padded_unsigned_jwt
```

`SliceFromStart` preserves the proof that elements fit in 8 bits, because the elements of the slicing result must be either an element of the input (`padded_unsigned_jwt`), or zero:

```
// SliceFromStart, strings.circom, line 116
out[i] <== in[i] * lts[i];
```

This proves that the elements of `` `header` `` must fit within 8 bits, even without constraining them so in `` `ConvertBase` ``.

The same reasoning applies to `` `iss_b64` `` which is passed to `` `HashBytesToField` `` after being sliced from the JWT.

The three remaining inputs of `` `HashBytesToField` `` are `` `kc_name` ``, `` `kc_value` `` and `` `aud_value` ``, all of which undergo the following transformations:

- They originate as witnesses of the form `` `"key":"value",` `` (extended key-value pair)
- These are given to `` `ExtClaimOps` ``, which passes them on to `` `ASCIISubstrExistsInB64` `` as the `` `A` `` input
- `` `ASCIISubstrExistsInB64` `` constrains all elements of the input `` `A` `` to 8 bits
- Subsequently, `` `ExtClaimOps` `` passes extended key-value pairs to `` `ExtendedClaimParser` `` which slices them into a key and value
- In the end, keys and values are passed to `` `QuoteRemover` `` where quotes are sliced off.

Since slicing preserves the 8-bit range check, these values don't need the extra bit constraints in `` `HashBytesToField` ``. In conclusion, the proposed optimization of `` `HashBytesToField` `` can be applied without other modifications to the code base.

With the max length parameters used at the time of writing, this finding will save about $7392 = 8 \cdot 934$ constraints, because

`` `maxHeaderLen` `` + `` `maxKCNameLen` `` + `` `maxKCValueLen` `` + `` `maxAudValueLen` `` + `` `maxExtIssLength_b64` ``$= 500 + 40 + 100 + 150 + 4 \cdot (1 + (100/3)) = 934$

# 12 - Too many arguments

● circuits/*

Informational

Description. Many templates in Circom have many inputs, and calling them using Circom's syntactic sugar for template input assignment makes it very hard to trace the arguments that are assigned to each input parameter.

Here's an example from `` `jwtchecks.circom:25` ``:

```
template ExtClaimOps(inCount, maxExtClaimLen, maxClaimNameLen, maxClaimValueLen, maxWhiteSpaceLen) {
    assert(maxExtClaimLen == maxClaimNameLen + maxClaimValueLen + maxWhiteSpaceLen + 2); // 4 for
quotes. 2 for colon, comma.

    signal input content[inCount];
    signal input index_b64;
    signal input length_b64;

    signal input ext_claim[maxExtClaimLen];
    signal input ext_claim_length;
    signal input name_length; // with quotes
    signal input colon_index;
    signal input value_index;
    signal input value_length; // with quotes

    signal input payload_start_index;
    //...
    signal output claim_name[maxClaimNameLen];
    signal output claim_value[maxClaimValueLen];
    //...
}
```

Calling such templates leads to code that looks like this (from `` `zkLoginMain.circom:291` ``):

```
    (aud_name_with_quotes, aud_value_with_quotes) <== ExtClaimOps(
        inCount, maxExtAudLength, aud_name_length, maxAudValueLenWithQuotes, maxWhiteSpaceLen
    )(
        padded_unsigned_jwt,
        aud_index_b64, aud_length_b64,
        ext_aud, ext_aud_length,
        aud_name_length, aud_colon_index, aud_value_index, aud_value_length,
        payload_start_index
    );
```

Recommendations. Use the longer syntax with named arguments when calling such functions. While more verbose, the logic becomes much clearer to the reader and less error-prone.

For example, we could fix the above template instantiation with:

```
component AudExtClaim = ExtClaimOps(
        inCount, maxExtAudLength, aud_name_length, maxAudValueLenWithQuotes, maxWhiteSpaceLen
    );

AudExtClaim.content <== padded_unsigned_jwt;
AudExtClaim.index_b64 <== aud_index_b64;
AudExtClaim.length_b64 <== aud_length_b64;

AudExtClaim.ext_claim <== ext_aud;
AudExtClaim.ext_claim_length <== ext_aud_length;
AudExtClaim.name_length <== aud_name_length;
AudExtClaim.colon_index <== aud_colon_index;
AudExtClaim.value_index <== aud_value_index;
AudExtClaim.value_length <== aud_value_length;

AudExtClaim.payload_start_index <== payload_start_index;

aud_name_with_quotes <== AudExtClaim.claim_name;
aud_value_with_quotes <== AudExtClaim.claim_value;
```