PYTHIA SERVICE WHITEPAPER

BY VIRGIL SECURITY

May 21, 2018

CONTENTS

Introduction

We all want to move beyond passwords. Unfortunately, it's the most prevalent form of authentication. The large majority of online services, devices, laptops, and phones require passwords. And worst, the way that most services are storing and protecting passwords is fundamentally flawed.

Criminals break into online services and steal password databases. Once they have these password databases, they can run cracking software to recover most passwords. The simplest form of recovery is to use rainbow tables, which is a large dictionary file of hashes mapped to passwords. Attackers simply look up a hash and find what password it was generated from.

Another way to recover stolen password databases is brute-force/dictionary attacks. On a single laptop an attacker can try 1 trillion different passwords in 2 weeks (2^20 passwords/second * 2^20 seconds/2 weeks = 2^40 = 1T). Serious attackers can throw a few thousand dollars of hardware--graphics processing units (GPUs)--and that number goes up to 10 quadrillion. That's insane. On the dark markets, passwords sell in the range of $1-30 dollars per account [1], which is a big incentive for criminals.

The current industry-recommended best practice is for services that store passwords to use "hard functions" to process these passwords. Simply, the tools scrypt, bcrypt, and iterated hashing make password processing slower (or use more memory) and so the burden is also applied to the password attackers. Unfortunately, these techniques put the same burden on the good guys--the service also has to pay this performance penalty on every legitimate login. These techniques don't give any leverage to the defenders. (Unlike, say, encryption, where the legitimate key holder can encrypt and decrypt quickly, but that attacker has an exponentially harder problem trying to crack the key.)

[1] On the Economics of Offline Password Cracking - Purdue CS

How does Pythia solve these problems?

Pythia provides multiple innovations over state-of-the-art:

- Cryptographic leverage for the defender (by eliminating offline password cracking attacks),
- Detection for online attacks, and key rotation to recover from stolen password databases,
- Designed for performance and scalability,
- No impact to end-user experience.

Are there any other solutions?

2-factor authentication: in theory, 2FA helps avoid an account takeover. While this may be a potential solution, 2FA is after-the-fact (i.e. the attacker already reverse-engineered the users' passwords and they can use them elsewhere). Also, most users don't enable 2FA because of inconvenience.

Salted hash: salting is when you introduce random characters into passwords before hashing them. While salting randomizes hashes, they have one major flaw: if you ever built a hashed-salted user login function, you know that you'll need to retrieve the salt to re-generate the user's password hash to compare it to the salted hash you store in your user database. I.e. you have to store the salt with the hashed password in your user database: so, it's available to attackers too.

Super-strong passwords: this is your best option today. Require super-strong passwords from every single one of your users and make sure that they don't reuse that same password elsewhere. Unfortunately, this is practically impossible.

What is Pythia?

Simply put, Pythia is a service that:

1. Replaces hashing & salts with public and private keys, making cracking a lot more expensive,
2. Enables the separation of your password database from the private key that's protected by Pythia.
3. Enables the immediate rotation of the private key, rendering a stolen password database unusable.

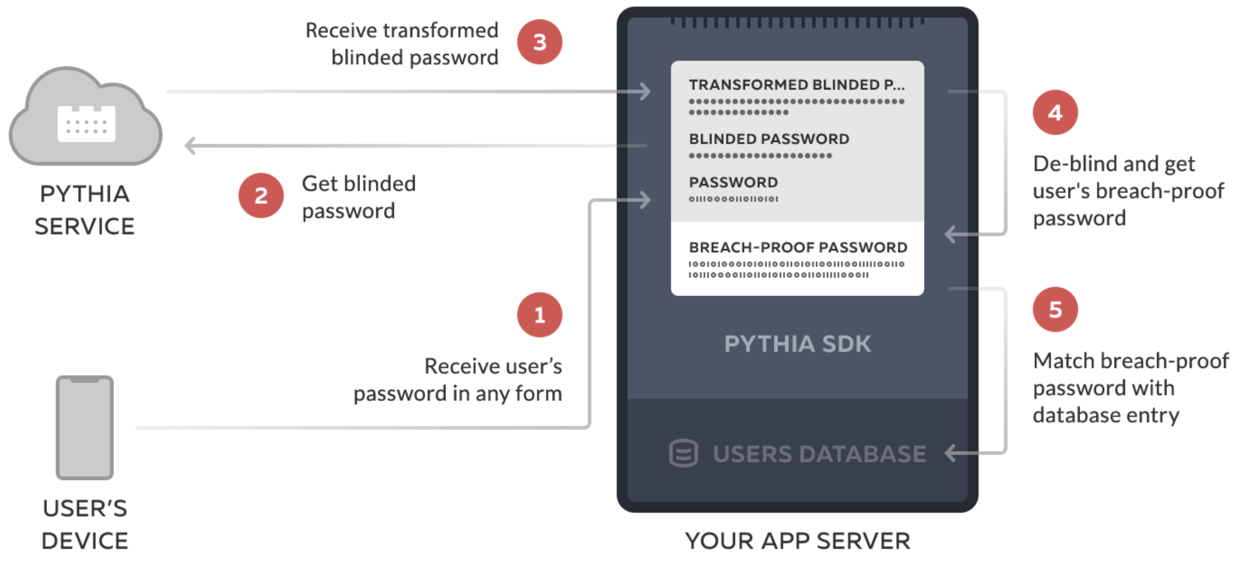
These 3 properties will keep your user passwords are safe from today's password breach techniques, even if your password database has gotten into the wrong hands.

How does it work?

The typical Pythia implementation consists of the following components:

- Your web server where you run your user auth code,
- Your database where you store your users' usernames and hashed passwords today,
- A cloud-based Pythia service account that your web server uses to generate breach-proof-passwords, which will replace your current password hashing practice.

User login flow


Receive transformed blinded password
3
PYTHIA SERVICE
2
Get blinded password
1
Receive user's password in any form
USER'S DEVICE
TRANSFORMED BLINDED P...
BLINDED PASSWORD
PASSWORD
BREACH-PROOF PASSWORD
PYTHIA SDK
USERS DATABASE
YOUR APP SERVER
4
De-blind and get user's breach-proof password
5
Match breach-proof password with database entry

1. User enters password which goes up to your web server.
2. Your web server hashes the password and encrypts it using a Blind() function from the Pythia-lib. Blinding makes sure that the Pythia service has no knowledge of the password.
3. The Pythia service transforms the blinded password using a secret that only Pythia knows.
4. Using the blinding secret from Step 2, your web server de-blinds the response from Pythia. The end result is the breach-proof-password that replaces your current password hash field value in your user table.
5. Every time users log in, you run steps 1-4 and compare if the login-time breach-proof-password matches with the registration-time breach-proof-password. If they do, the password is correct.

Performance: faster than bcrypt & scrypt

Depending on the link between your web backend and the Pythia server, your user login may be faster than the existing techniques of iterated hashing, bcrypt, and scrypt. Instead of slowing down attackers (and defenders) by running harder and harder operations, Pythia uses cryptography to give defenders back the advantage. Pythia was designed with the goal of scaling up to protecting every password in the world.

Protection against password DB breach & online guessing attacks

Pythia has built-in support for key rotations and detecting online guessing attacks. A website/app operator can request a key rotation from the Pythia server, update all the stored passwords, and, then request that the original key be destroyed. Once it is, any stolen passwords are useless. No one can recover them without the key.

Pythia also builds in online attack detection. If a password database is stolen it can't be cracked offline anymore, but an attacker can send requests to the crypto server. Pythia was designed so that all requests require an unforgeable user ID and client ID provided in plaintext (not blinded like passwords). This allows the Pythia service to detect attacks. Detection is very powerful and often overlooked as a defensive mechanism--Pythia can lock accounts, throttle (slow down) requests, and send out alerts to your web backend.

Open source

Pythia's originators are: Adam Everspaugh and Rahul Chaterjee, University of Wisconsin—Madison; Samuel Scott, University of London; Ari Juels and Thomas Ristenpart, Cornell Tech.

Virgil Security implemented the service for developers and published it on GitHub at https://github.com/VirgilSecurity/pythia under the AGPL-3.0 license: https://github.com/VirgilSecurity/pythia/blob/master/LICENSE

The math

From hashes to numbers

Today, passwords are protected by being hashed (hashing theoretically is a one-way function). A hash is nothing more than an array of bytes which uniquely identifies the password.

Another form which can represent an array is a number. The bigger the array - the bigger the number. So in Pythia, instead of hashing the password as most auths do, we convert the password to a big (256-bit) number.

Elliptic curve BLS12-381: also used by Zcash cryptocurrency

Pythia works with elliptic curve “BLS12-381”: the same curve that’s used in Zcash, which the Zcash team claims to be “the first open, permissionless cryptocurrency that can fully protect the privacy of transactions using zero-knowledge cryptography”.

If you’re new to elliptic curves, this is how they work: we start with a point with two coordinates of **x** and **y**.

1. We generate a 256-bit number from the user’s password and hash it into the **x** coordinate of the elliptic curve equation,
2. By solving the equation, we get **y** and check if it fits the chosen elliptic curve.
3. If not, we increment **x** and try again. We repeat this until **y** fits on the curve.

Blinding on the client makes Pythia not see the password

After we get a valid elliptic curve point (**x** and **y** now fit on the curve), we generate a big random number **r** and multiply the (**x**, **y**) point by **r**. This process is called the blinding and we do it because we don’t want the Pythia service to see the number we generated from the user's password (**x**). The result is another elliptic curve point: **X**.

Transformation of blinded value with Pythia’s secret number on the Pythia sever

X together with the user’s unique **ID** is sent to the Pythia service where this big number is stored. Then the user’s **ID** is also converted to the elliptic curve point the same way we did with the password: let's call this **T**.

X and **T** are put into a function called Bilinerar Pairing which produces a single point out of these two. The resulting point is multiplied by a **secret number** stored at the Pythia server and returned to the client. Let's call it **Y**.

Deblinding on the client

The client calculates an inverse of **r** and multiplies **Y** by that inverse to eliminate the random factor used for Blinding.

The result is (user's password hash + User ID) multiplied by the Pythia service's secret. Because of the discrete logarithm problem, there is no way to tell what the server's secret is or what the client password's hash is only by looking at the result of the computation.

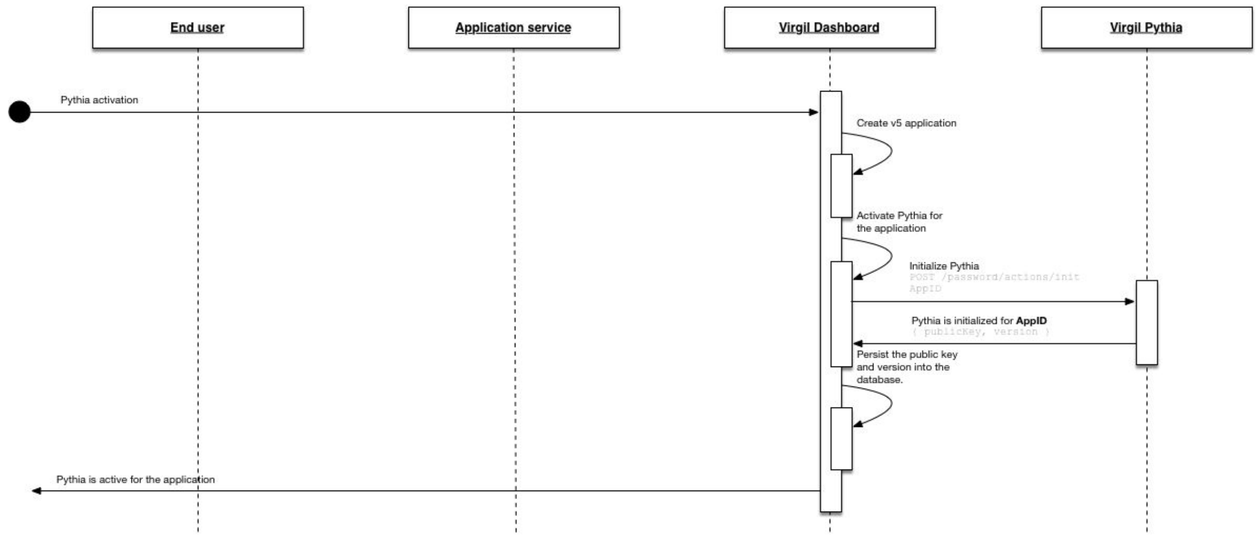
Crypto primitives

Crypto primitive	**What we use it for?**
Curve BLS12-381	Blinding, transforming, ZKP (zero-knowledge proof)
SHA-384	Blinding, transforming, ZKP (zero-knowledge proof)
HMAC	ZKP

Basic flows

Pythia initialization for the application

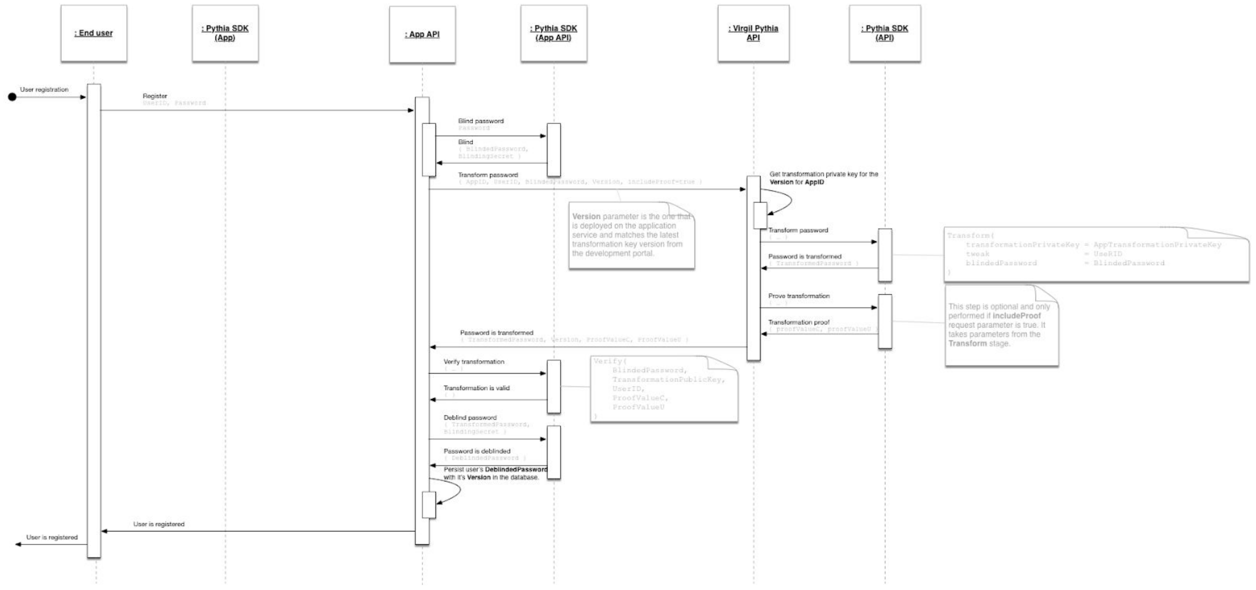
This is this first action that enables Virgil's Pythia service for all further calls. To make Pythia available for the application developer, they must at first register on the Virgil development portal at https://dashboard.virgilsecurity.com/login and create a new Pythia app.


End user
Application service
Virgil Dashboard
Virgil Pythia
Pythia activation
Create v5 application
Activate Pythia for the application
Initialize Pythia
POST /password/actions/init
AppID
Pythia is initialized for AppID
(publicKey, version)
Persist the public key and version into the database.
Pythia is active for the application

Application developer is supposed to copy the **Proof Key** returned returned after Pythia initialization and deploy them on his application service. **Version** value is used as a request parameter in password transformations to fetch proper transformation private Key by the Pythia service, and **Public Key** is used by the application service to verify transformations performed by the Pythia service.

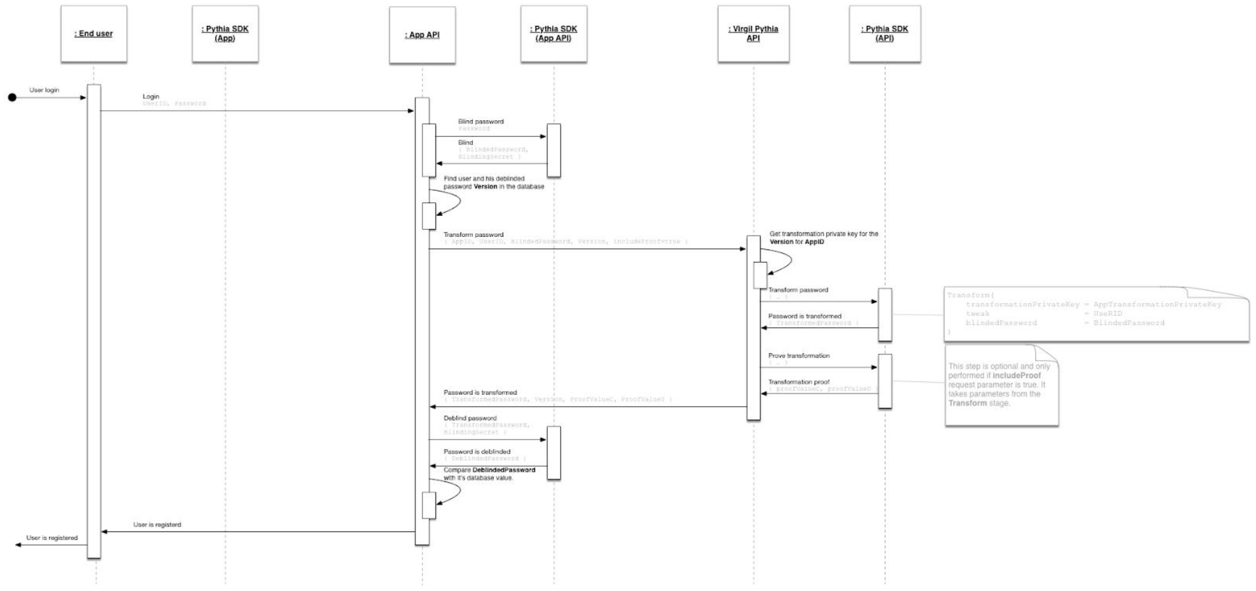
End user registration on the application service

Password blinding can take place on a client or on a web server, it which case one is supposed to provide a **blindingSecret** alongside with a transformed password value to make the **Deblind()** operation possible after the Pythia server transformed the password.


: End user
: Pythia SDK (App)
: App API
: Pythia SDK (App API)
: Virgil Pythia API
: Pythia SDK (API)
User registration
Register
UserID, Password
Blind password
Password
Blind
(BlindedPassword, BlindingSecret)
Transform password
(AppID, UserID, BlindedPassword, Version, includeProof=true)
Get transformation private key for the Version for AppID
Version parameter is the one that is deployed on the application service and matches the latest transformation key version from the development portal.
Transform password
(…)
Password is transformed
(TransformedPassword)
Transform(transformationPrivateKey = AppTransformationPrivateKey tweak = UseRID blindedPassword = BlindedPassword)
Prove transformation
(…)
Transformation proof
(proofValueC, proofValueU)
This step is optional and only performed if includeProof request parameter is true. It takes parameters from the Transform stage.
Password is transformed
(TransformedPassword, Version, ProofValueC, ProofValueU)
Verify transformation
(…)
Transformation is valid
( )
Verify(BlindedPassword, TransformationPublicKey, UserID, ProofValueC, ProofValueU)
Deblind password
(TransformedPassword, BlindingSecret)
Password is deblinded
(DeblindedPassword)
Persist user's DeblindedPassword with it's Version in the database.
User is registered
User is registered

The proof parameter is supposed to be true for new user registrations (initial password transformation). This allows validation that Pythia's response is correct.

End user password validation

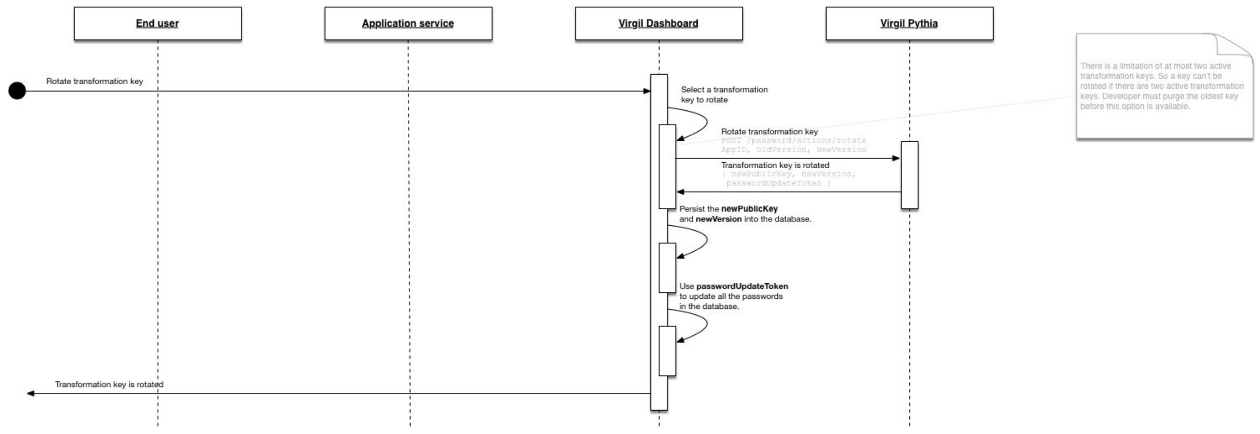

: End user
: Pythia SDK (App)
: App API
: Pythia SDK (App API)
: Virgil Pythia API
: Pythia SDK (API)
User login
Login
UserID, Password
Blind password
Password
Blind
(BlindedPassword, BlindingSecret)
Find user and his deblinded password Version in the database
Transform password
(AppID, UserID, BlindedPassword, Version, includeProof=true)
Get transformation private key for the Version for AppID
Transform password
(…)
Password is transformed
(TransformedPassword)
Transform(transformationPrivateKey = AppTransformationPrivateKey tweak = UseRID blindedPassword = BlindedPassword)
Prove transformation
(…)
Transformation proof
(proofValueC, proofValueU)
This step is optional and only performed if includeProof request parameter is true. It takes parameters from the Transform stage.
Password is transformed
(TransformedPassword, Version, ProofValueC, ProofValueU)
Deblind password
(TransformedPassword, BlindingSecret)
Password is deblinded
(DeblindedPassword)
Compare DeblindedPassword with it's database value.
User is registered
User is registered

Include proof is false here because the password transformation was already verified in the user registration stage.

Rotate transformation key / issue new password update token

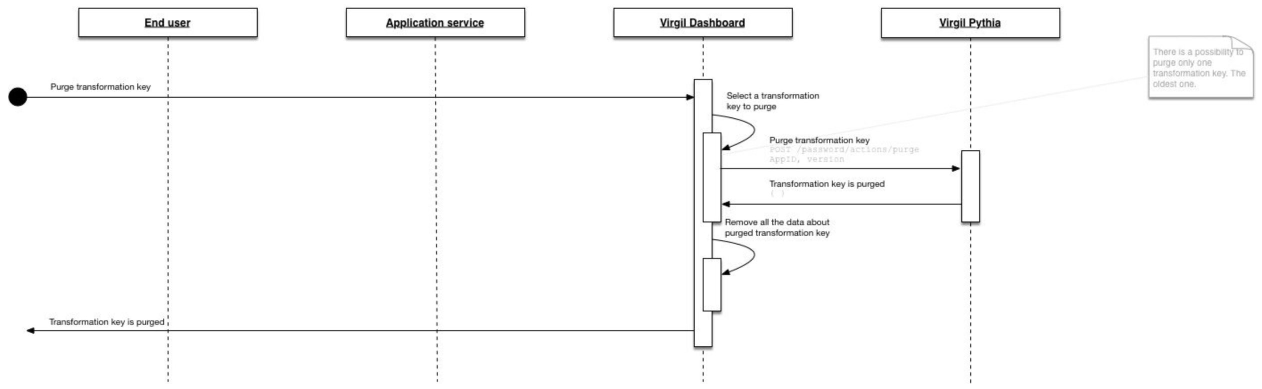
This scenario gets activated when it's necessary to rotate the application transformation key (i.e. the password database has been breached). This action involves new transformation key issuing along with updating all deblinded password stored in the application database. In this scenario application developer logs into his/her Virgil Dashboard account and activates application transformation key rotation. Beware that only two active transformation keys may be active, so this process is unavailable until all legacy transformation keys are purged.

The Virgil Dashboard requests a transformation key rotation, which means that a completely new key will be generated and the key's version will be incremented. Developer receives a new transformation public key (with its version) which is supposed to be saved for further transformation validations, and password update token which must be applied in **UpdateBreachProofPassword()** operation for all the passwords stored in the database.


End user
Application service
Virgil Dashboard
Virgil Pythia
There is a limitation of at most two active transformation keys. So a key can't be rotated if there are two active transformation keys. Developer must purge the oldest key before this option is available.
Rotate transformation key
Select a transformation key to rotate
Rotate transformation key
POST /password/actions/rotate
AppID, OldVersion, NewVersion
Transformation key is rotated
{ newPublicKey, newVersion, passwordUpdateToken }
Persist the newPublicKey and newVersion into the database.
Use passwordUpdateToken to update all the passwords in the database.
Transformation key is rotated

Purge transformation key

This action is performed on the Virgil Development portal and removes the oldest transformation key by its version. The newest (HEAD) transformation key cannot be purged.


End user
Application service
Virgil Dashboard
Virgil Pythia
Purge transformation key
Select a transformation key to purge
There is a possibility to purge only one transformation key. The oldest one.
Purge transformation key
POST /password/actions/purge
AppID, version
Transformation key is purged
{ }
Remove all the data about purged transformation key
Transformation key is purged