High Availability in the Internal Google Key Management System (KMS)

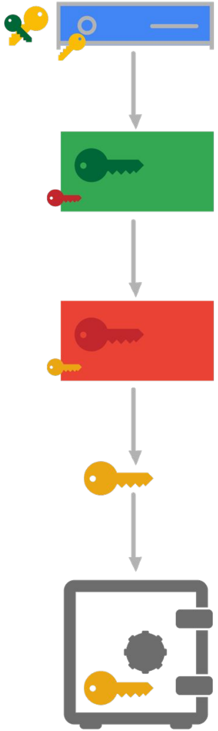
Anand Kanagala, Bodo Möller, Darrell Kindred, Glenn Durfee, Hannes Eder, Maya Kaczorowski, Tim Dierks, Umesh Shankar

Google LLC

Real World Crypto 2018, Zurich, 2018-Jan-10

Not the Google Cloud KMS

Google's key hierarchy



Storage Systems (Millions)
Data encrypted with DEKs, DEKs are encrypted with KEKs

KMS (Tens of Thousands)
KEKs are stored in KMS

Root KMS (Hundreds)
KMS is protected with a KMS master key in Root KMS

Root KMS master key distributor (Hundreds)
Root KMS master key is distributed in memory

Physical safes (a few)
Root KMS master key is backed up on hardware devices

Why use a KMS?

Core motivation: code needs secrets!

Where:

- In code repository?
- On production hard drives?

Alternative:

- Use a KMS!

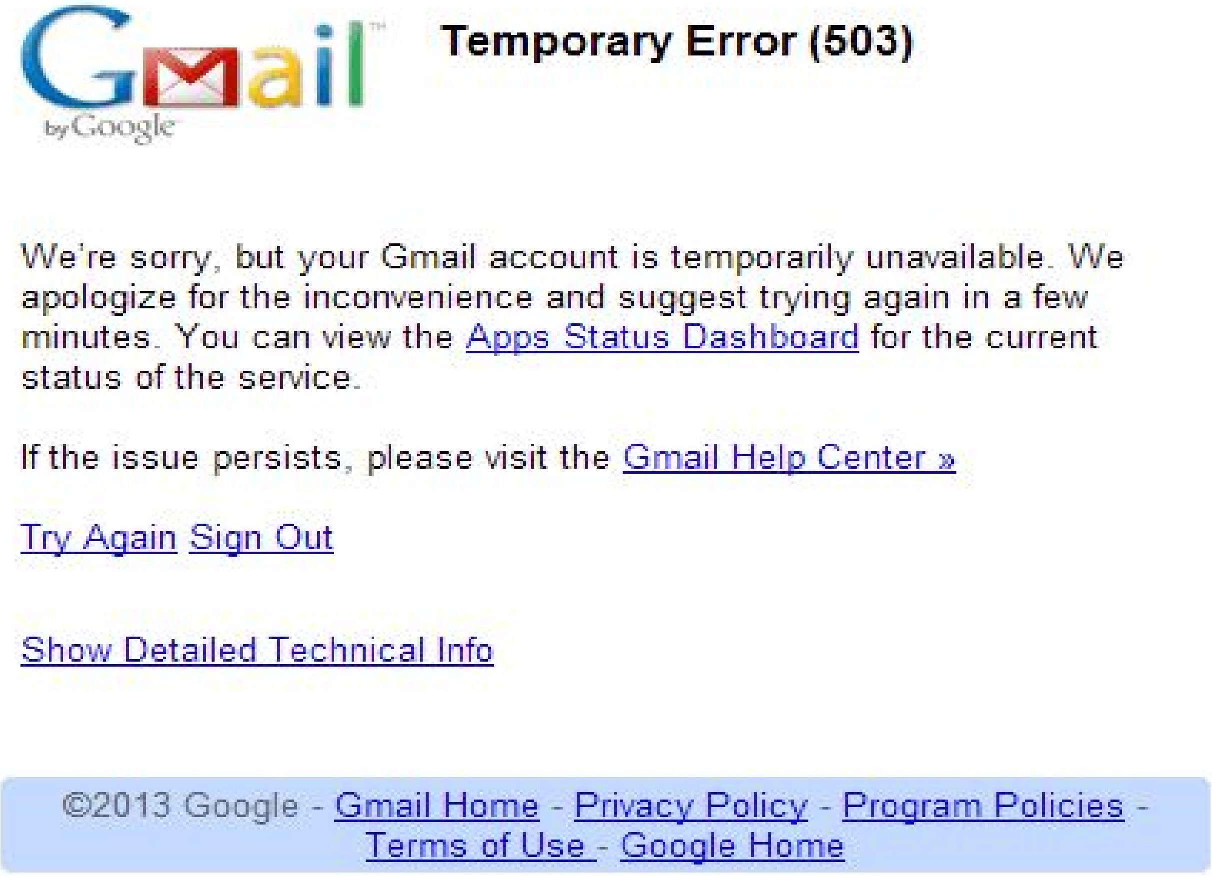
Centralized Key Management

Solves key problems for everybody:

- Access control: who <humans or services?>, what <is the build verifiable?>
- Auditing of cryptographic operations
- Key-handling code management
- Separation of trust

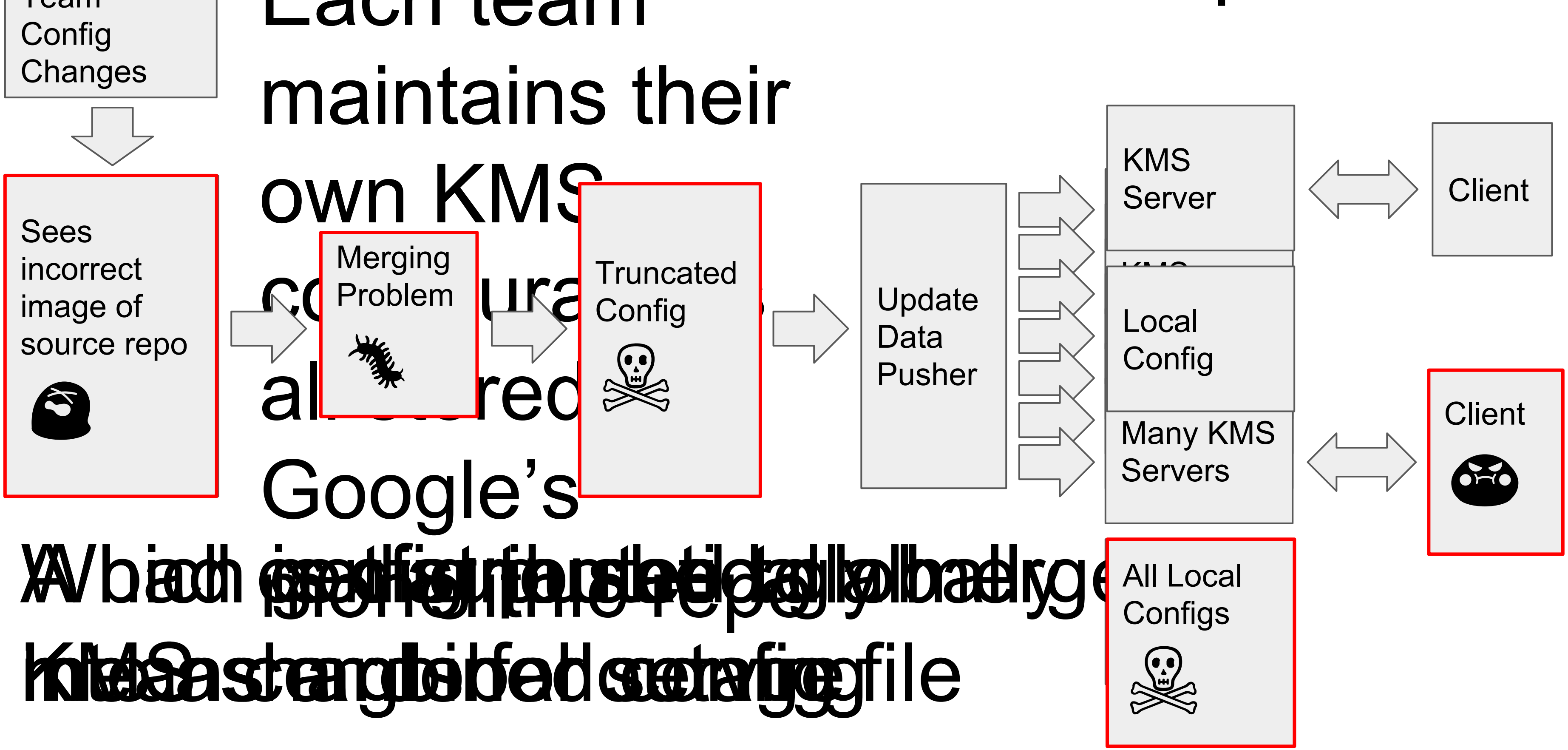
What could go wrong?

The Great Gmail Outage of 2014



https://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html

Normal Operation



Lessons Learned

The KMS had become

- a single point of failure
- **a startup dependency** for services
- often **a runtime dependency**

==> KMS Must Not Fail Globally

KMS Must Not Fail Globally

- Eliminated the global control plane
- Controlled rollout of binaries and configuration
- Minimize dependencies
- Regional failure isolation
- … for the KMS and all dependencies

Google KMS - (some) Requirements

Category	**Requirement**
Availability	> 99.9995% of requests are served
Latency	99% of requests are served < 10 ms
Scalability	All of Google’s Key Management needs
Security	Effortless & foolproof Key Rotation
Efficiency	Requests/Core: As high as possible

Design Choices

- Granularity of Encryption
- Rate of Change
- Position in the trust/key hierarchy

Stateless Serving

Insight: At the KMS layer, key material is not mutable state.

Immutable Key material + Key Wrapping

==> Stateless Server ==> Trivial **Scaling**

Keys in RAM ==> **Low Latency** Serving

Google KMS - What we ended up with

- Infrastructure for managing secrets
- Wraps/unwraps data-encryption-keys(DEK) using keys that never leave the service (KEK)
- Not a traditional database/storage system
- Not a data-encryption service

Google KMS - Requirements Met

Category	Requirement	Actual
Availability	> 99.9995% of requests are served	No downtime since the Gmail outage in 2014 January **>> 99.9999%**
Latency	99% of requests are served < 10 ms	**99.9%** of requests are served **< 200 µs**
Scalability	All of Google’s Key Management needs	~10^7 requests/sec ~10^4 processes & cores
Efficiency	Requests/Core: As high as possible	4-12K requests/sec/core

Why rotate keys?

- Key Compromise
  - *Also requires access to cipher text*
- Broken Ciphers
  - *Access to cipher text is enough*
- Rotating keys limits the window of vulnerability
- *But* Rotating Keys is error prone => data loss

Robust Key Rotation at Scale - 0

Goals

1. KMS clients design with rotation in mind
2. Using multiple key versions is no harder than using a single key
3. Very hard to lose data

Robust Key Rotation at Scale - 1

- Clients choose
  - Frequency of rotation: e.g. every 30 days
  - TTL of cipher text: e.g. 30,90,180 days, 2 years, etc.

- KMS guarantees ‘Safety Condition’
  - All ciphertext produced within the TTL can be deciphered using a keyset in the KMS.

- Tightly integrated with Google's standard cryptographic library
  - Supports **multiple key versions**
  - Each of which can be a different cipher

Robust Key Rotation at Scale - 2

- KMS
  - Derives the number of key versions to retain
  - Adds/Promotes/Demotes/Deletes Key Versions over time
  - Generation/Deletion of key versions completely separate from serving system
  - Rolled out slowly

Time ⇢

	T0	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
K1	A	P	P	A	A	A	SFR				
K2			A	P	P	A	A	A	SFR		
K3					A	P	P	A	A	A	SFR
K4							A	P	P	A	A

A - Active
P - Primary
SFR - Scheduled for Revocation

Mitigating Hardware Faults

- Crypto provides leverage and can amplify errors -
    - A single undetected bit error in a wrapping of a DEK can render large chunks of data unusable.
- Causes of bit errors
    - NICs twiddle bits, Broken CPUs, Cosmic rays flip bits in DRAM.
- Software Mitigations
    - Verify correctness of crypto ops at process start
    - After wrapping DEKs and before responding, we Unwrap
    - Storage services
        - Read back plain text after writing encrypted data blocks
        - Replicate/parity protect at a higher layer

Google KMS - Summary

Implementing encryption at scale required highly available key management.

At Google’s scale this meant 6.5 9s of availability.

To achieve HA and security requirements, we used several strategies:

- Best practices for change management and staged rollouts
- Minimized dependencies and aggressively defend against their unavailability
- Isolated by region & client type
- Combined immutable keys + wrapping to achieve scale
- A declarative API for key rotation
- Defend against hardware issues

Thank You!

Merci! Danke! Grazie!

Further Reading

- Google Cloud Encryption at Rest whitepaper: https://cloud.google.com/security/encryption-at-rest/default-encryption/
- Google Application Layer Transport Security: https://cloud.google.com/security/encryption-in-transit/application-layer-transport-security/
- CrunchyCrypt cryptography and key versioning library: https://github.com/google/crunchy
- Site Reliability Engineering (SRE) handbook: https://landing.google.com/sre/book.html

The End