Payload Compromised: Full Key Recovery in Rocket.Chat E2EE

Speaker: Hayato Kimura

Contributors: Ryoma Ito, Kazuhiko Minematsu and Takanori Isobe

Hayato Kimura

NICT / The University of Osaka, Japan

- Former Application Security Engineer
- Now: Applied Cryptography Researcher
- Focus: Where crypto protocols meet appsec

Our Team

Dr. Ryoma Ito

NICT, Japan

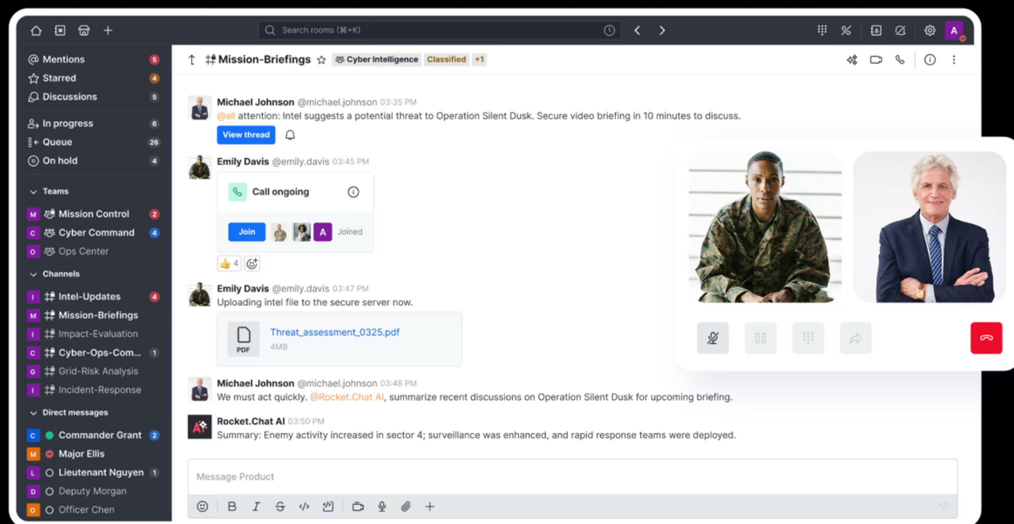
Dr. Kazuhiko Minematsu

NEC, Japan

Prof. Takanori Isobe

The University of Osaka, Japan

What is



- Rocket.Chat
  - Commercial OSS
  - On-premises and SaaS
  - Contrasts:
Slack, MS Teams, Zulip, Mattermost, Element …

Agenda

1. **Rocket.Chat architecture**

    Encryption overview, Core Components, three "Smells"

Agenda

1. **Rocket.Chat Architecture**

   Encryption overview, core components, three "Smells"

2. **Investigation & Attacks**

   Password generator teardown, entropy reduction, key recovery

Agenda

1. **Rocket.Chat architecture**

   Encryption overview, Core Components, three "Smells"

2. **Investigation & Attacks**

   Password generator teardown, entropy reduction, key recovery

3. **What the Attacker Gets**

   Full message decryption, forgery, broken recovery, total compromise

Agenda

1. **Rocket.Chat architecture**

   Encryption overview, Core Components, three "Smells"

2. **Investigation & Attacks**

   Password generator teardown, entropy reduction, key recovery

3. **What the Attacker Gets**

   Full message decryption, forgery, broken recovery, total compromise

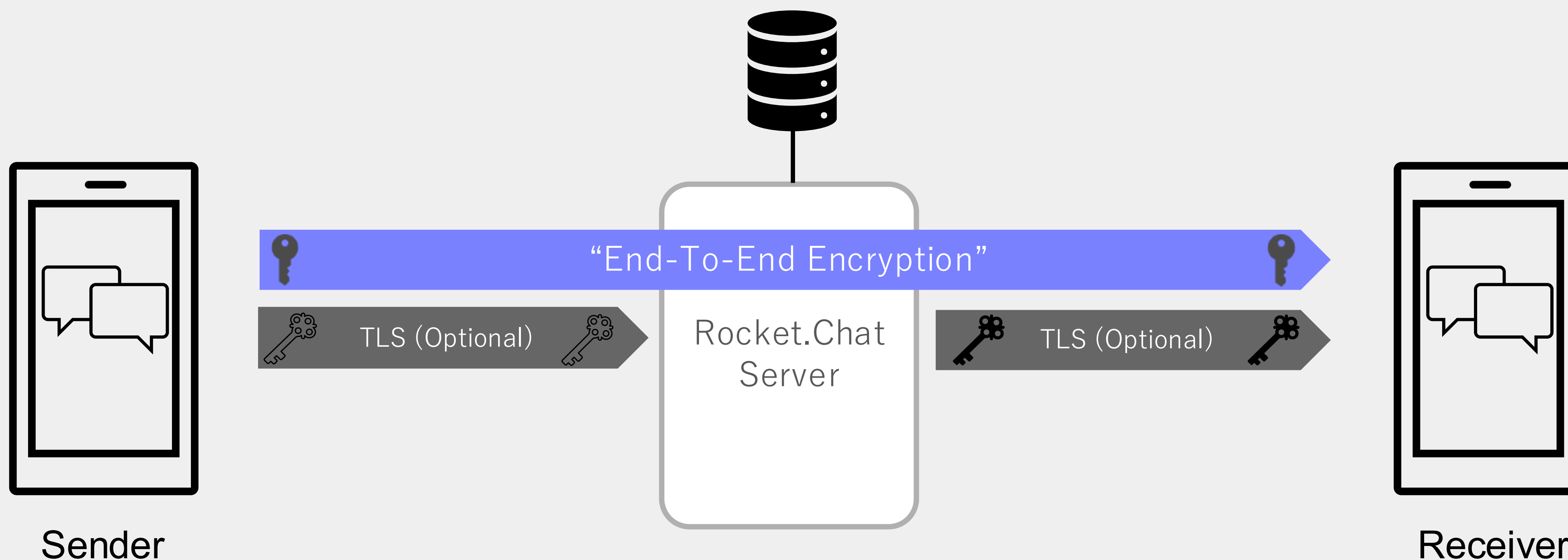
4. **Lessons & Fixes**

   OSINT analysis, disclosure timeline, takeaways

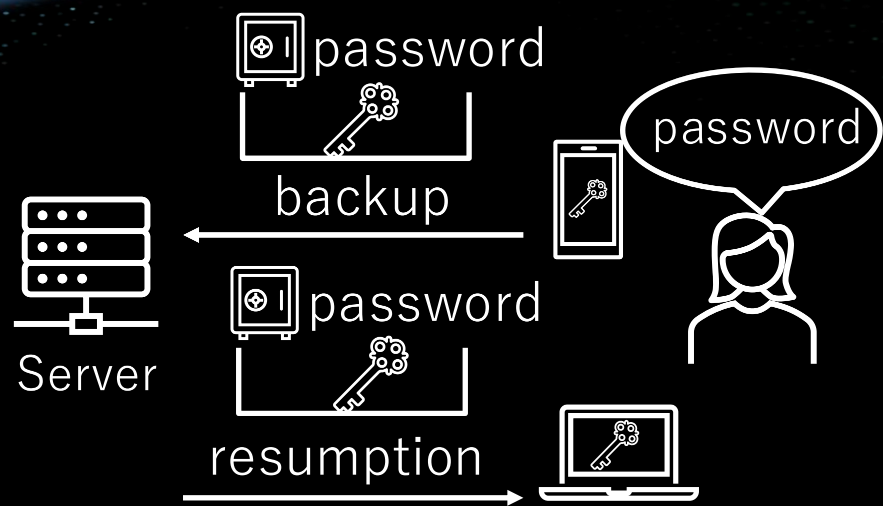
Part 1
Rocket.Chat E2EE architecture

Rocket.Chat encryption

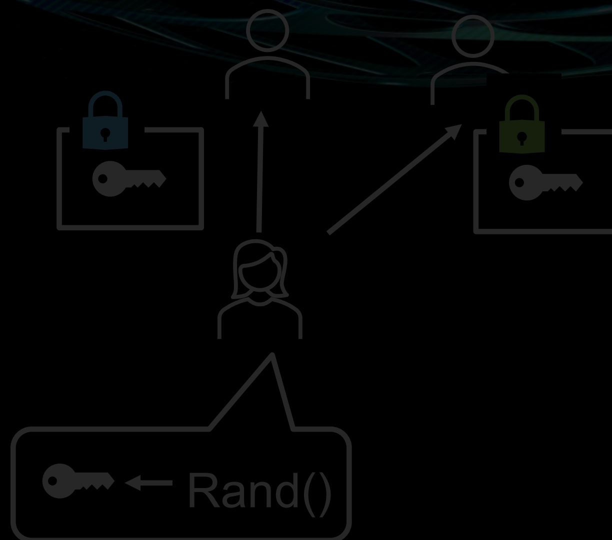
E2EE specification is available, but it lacks detail



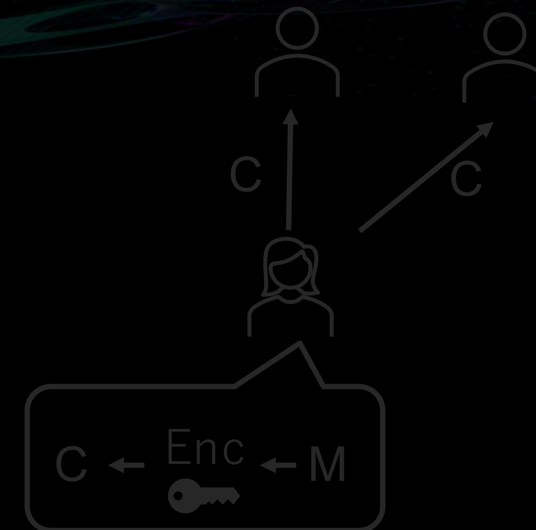
Core Components



RSA Key Backups

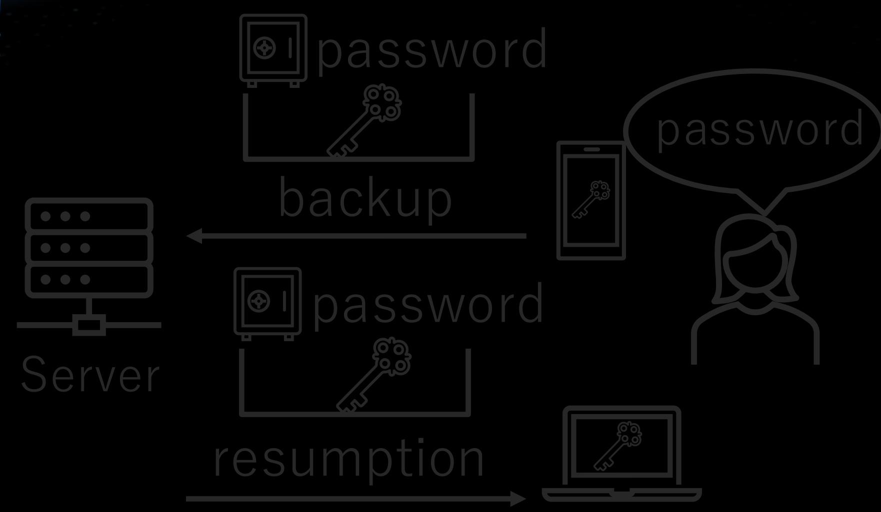


Session Establishment

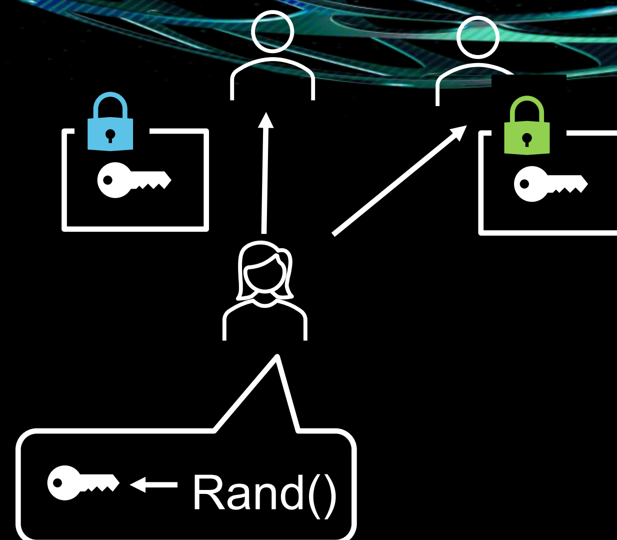


Message Encryption

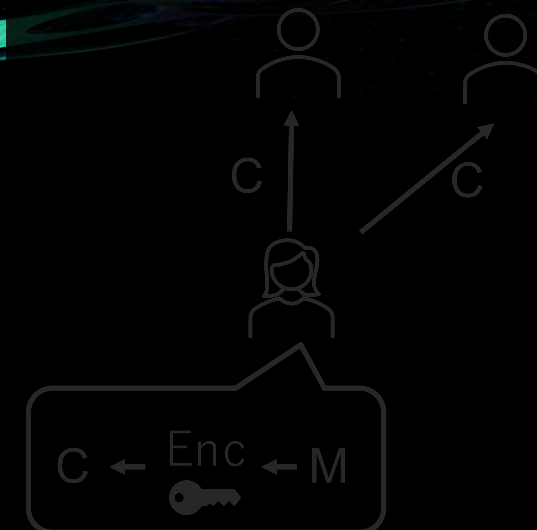
Core Components



RSA Key Backups

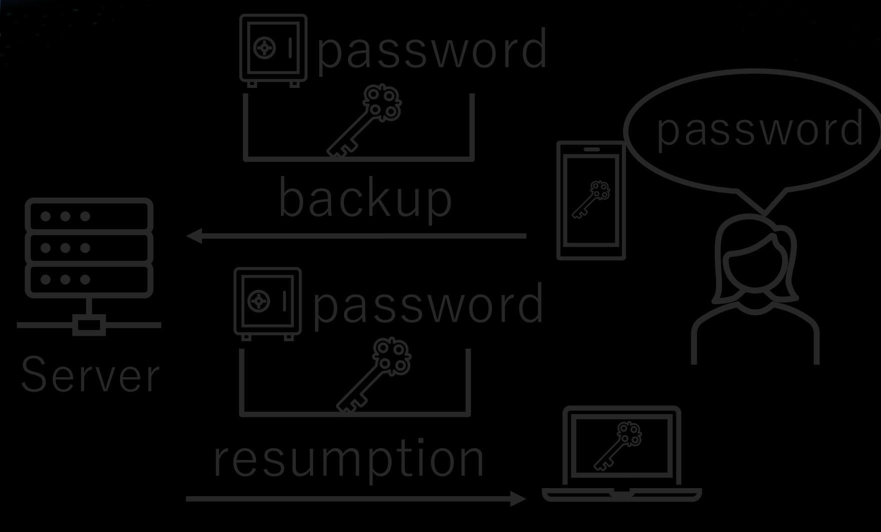


Session Establishment

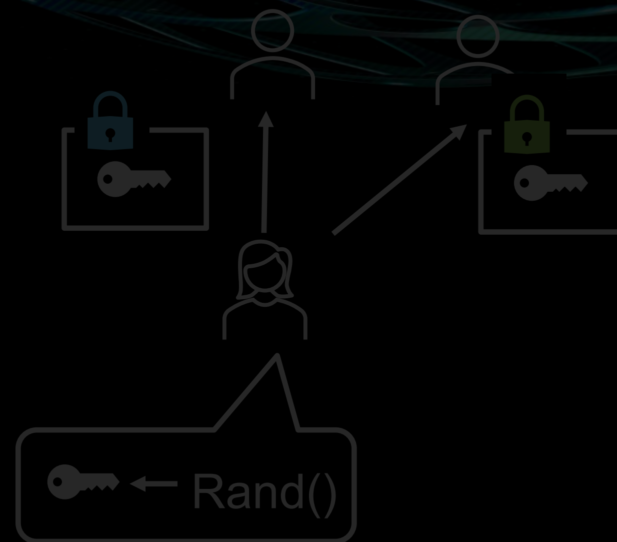


Message Encryption

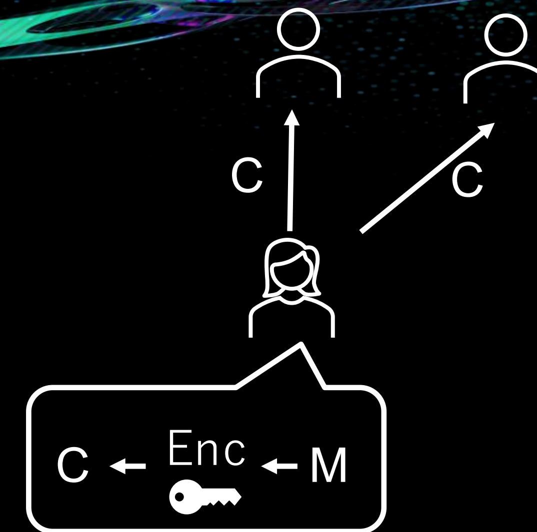
Core Components



RSA Key Backups

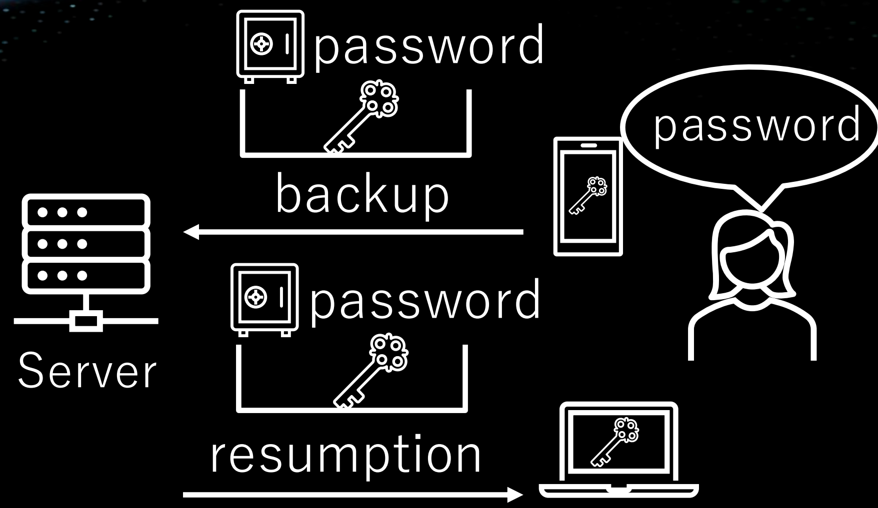


Session Establishment

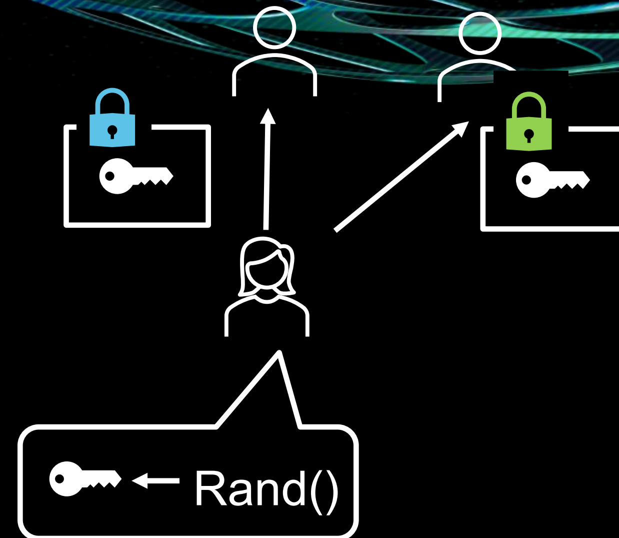


Message Encryption

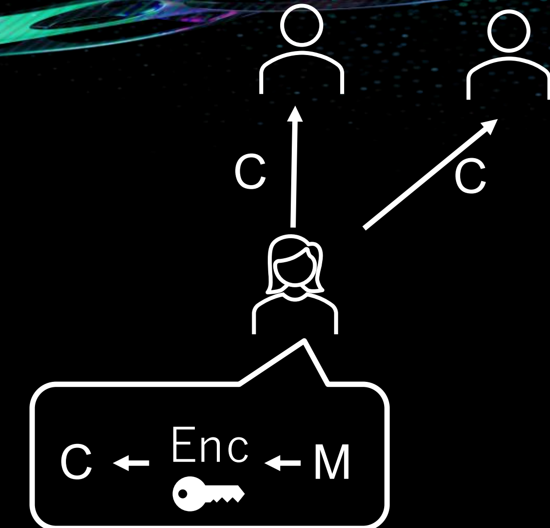
Core Components



RSA Key Backups

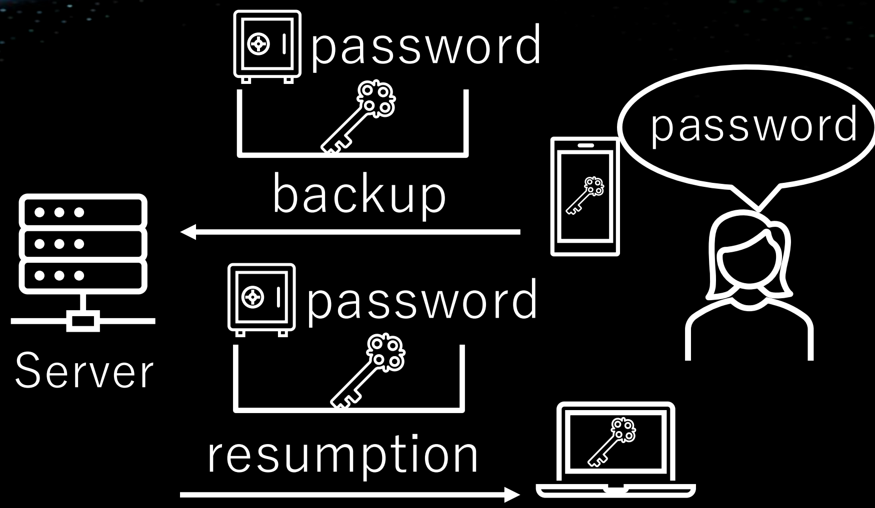


Session Establishment

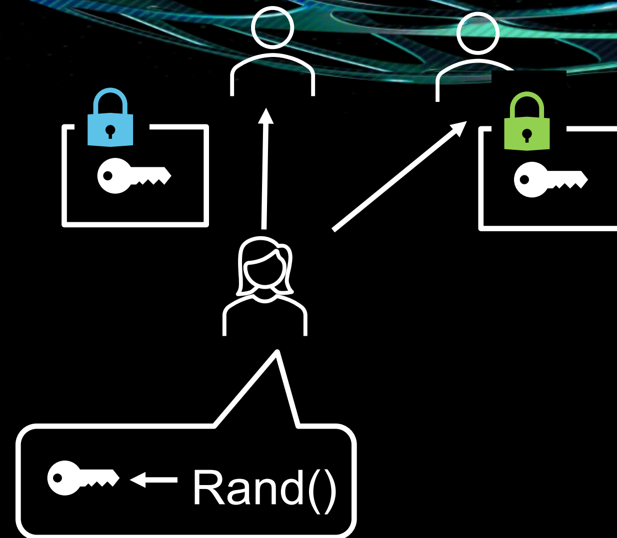


Message Encryption

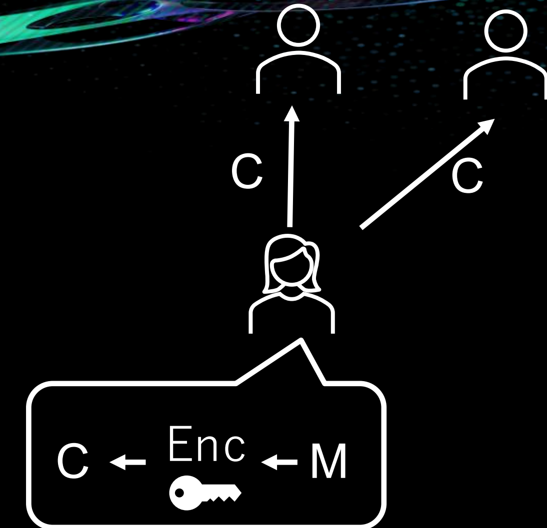
“Smell” of the vulnerabilities



RSA Key Backups

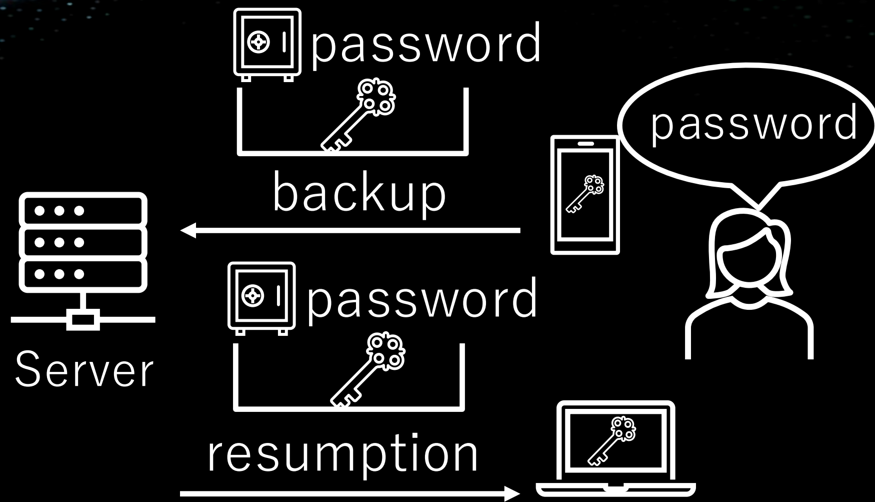


Session Establishment



Message Encryption

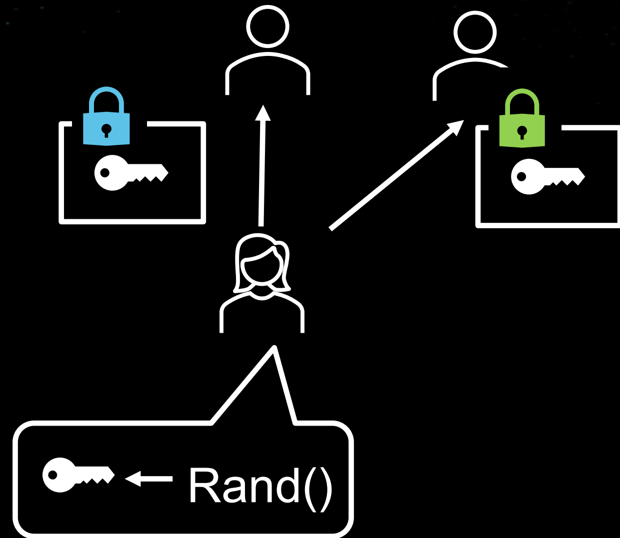
"Smell" of the vulnerabilities



RSA Key Backups
with Password encryption

- Password generation is "automatic".
- But how?

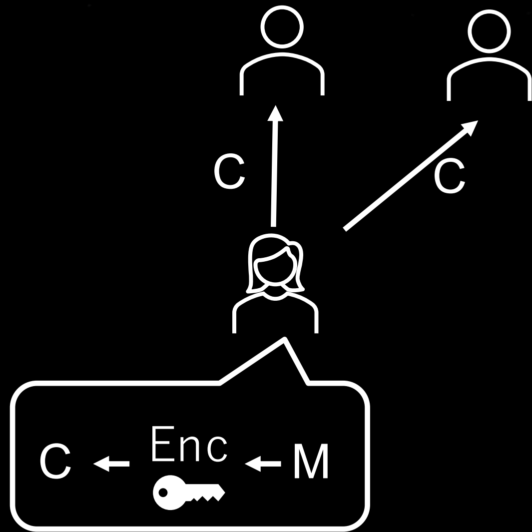
“Smell” of the vulnerabilities



Session Establishment
with RSA encryption

•Legacy key exchange

"Smell" of the vulnerabilities



Message Encryption
with AES-CBC

•No integrity protection

On the RSA Key backup...

- 😀 Docs of E2EE is available
- 😡 But it lacks detail
  (No entropy, no gen-algorithm)

Algorithms Used

Specifically, **E2EE** uses:

```
- client key pair: RSA-OAEP, length 2048
```

```
- master key: AES-CBC, length 256, iterations: 1000
```

```
- session key: AES-CBC, length 128
```

On the RSA Key backup...

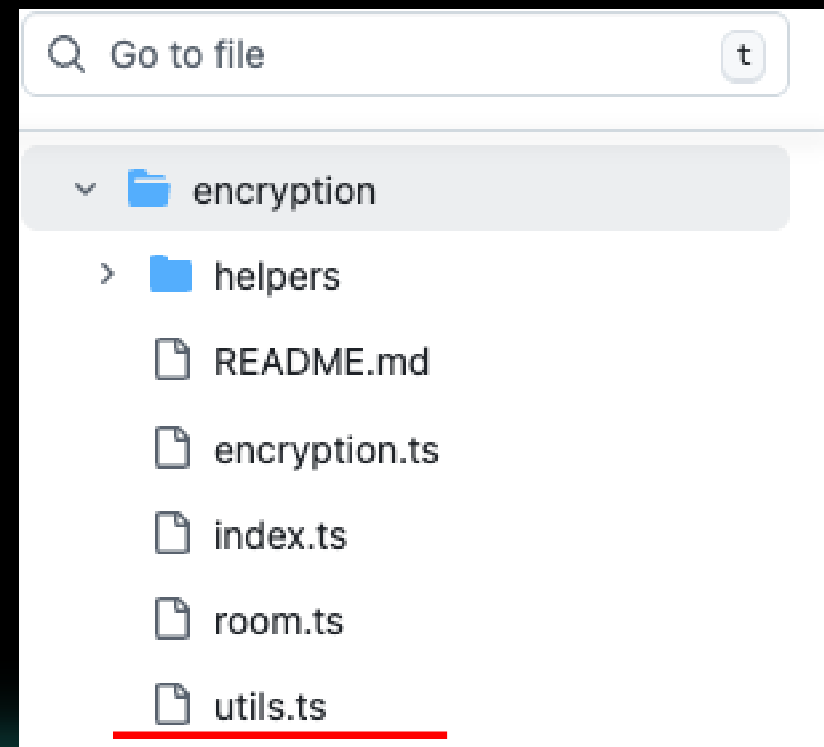
- 😀 Docs of E2EE is available
- 😡 But it lacks detail
  (No entropy, no gen-algorithm)
- 🙂 Rocket.Chat is open source, though

Rocket.Chat Public
The Secure CommsOS™ for mission-critical operations
TypeScript 45,106 13,505 2,392 (8 issues need help)

Rocket.Chat.ReactNative Public
The Secure CommsOS™ for mission-critical operations
TypeScript 2,377 MIT 1,444 225 220

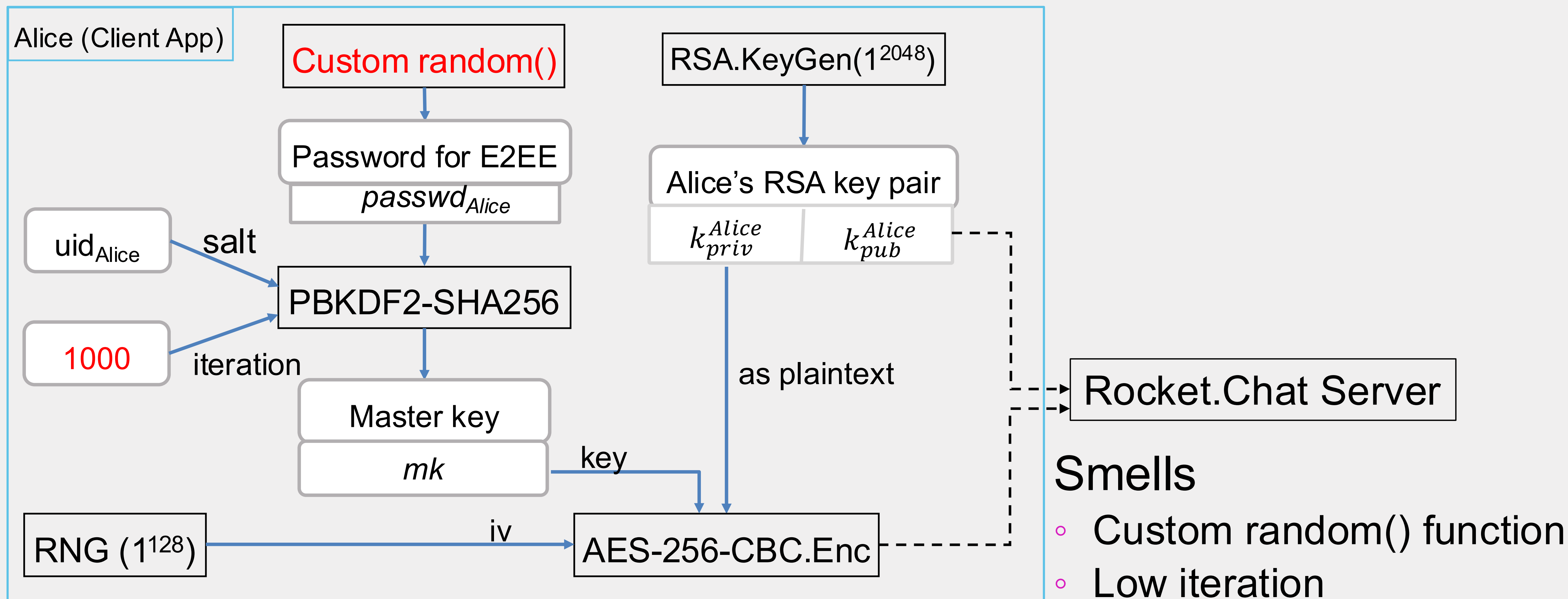
On the RSA Key backup...

- Docs of E2EE is available 😀
- But it lacks detail 😡
  (No entropy, no gen-algorithm)
- Rocket.Chat is open source, though 🙂



Part 2
Investigation & Attacks

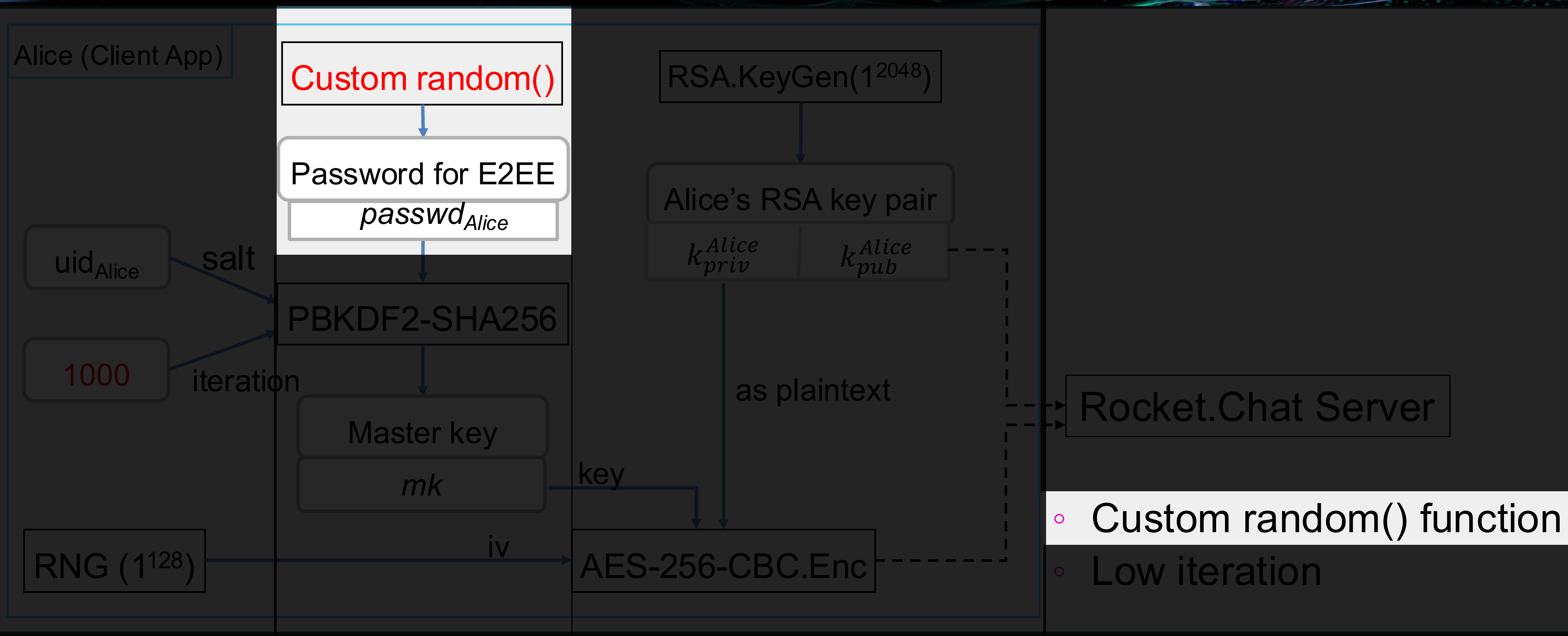
RSA Key backup



Smells

- Custom random() function
- Low iteration

RSA Key backup



- Custom random() function
- Low iteration

RSA Key backup

Alice (Client App)

Custom random()

Password for E2EE

passwd$_{Alice}$

uid$_{Alice}$

salt

PBKDF2-SHA256

1000

iteration

Master key

mk

key

RNG (1^{128})

iv

RSA.KeyGen(1^{2048})

Alice's RSA key pair

k_{priv}^{Alice} k_{pub}^{Alice}

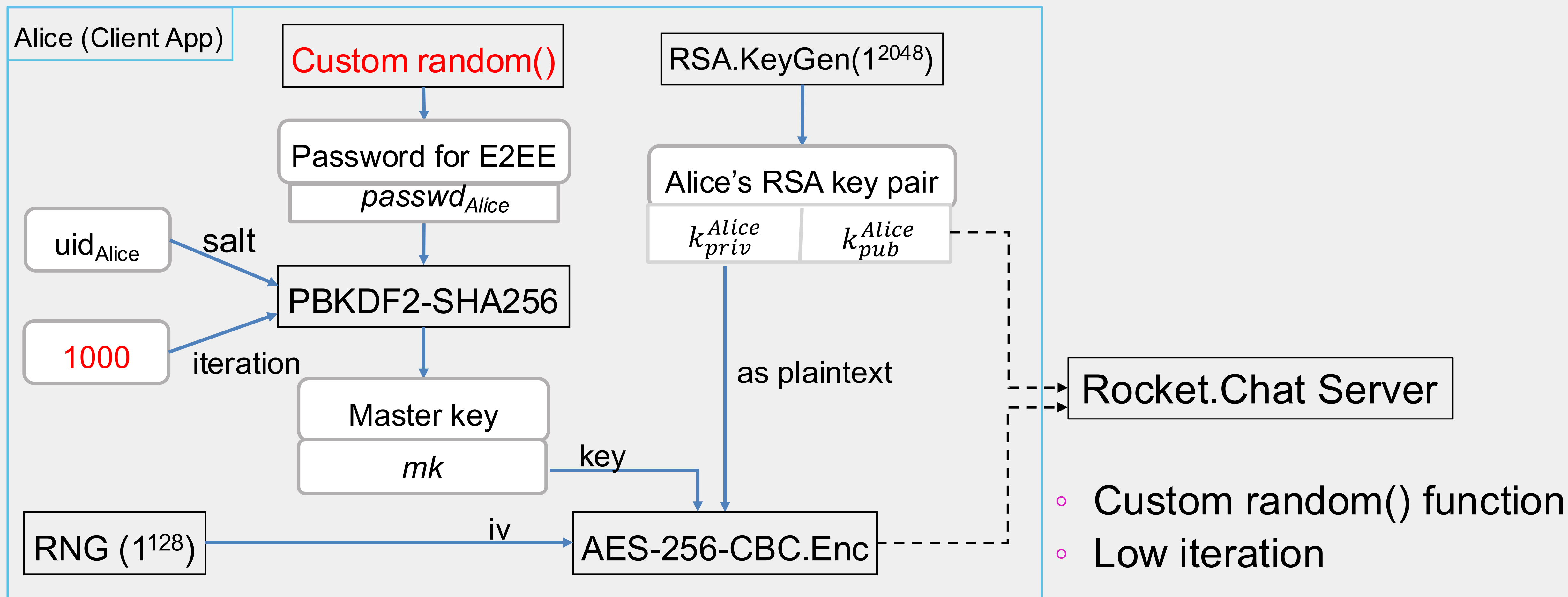
as plaintext

AES-256-CBC.Enc

Rocket.Chat Server

- Custom random() function
- Low iteration

RSA Key backup



- Custom random() function
- Low iteration

RSA Key backup : the password generator

random() function

- Custom random() function 🧐

```
1 ⌄  export function random(length: number): string {
```

RSA Key backup : the password generator

random() function

- Custom random() function 🧐
- 62-character set (lower letter & UPPER letter & numeric)

```
1 export function random(length: number): string {
2         let text = '';
3         const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
```

RSA Key backup : the password generator

random() function

- Custom random() function 🧐
- 62-character set (lower letter & UPPER letter & numeric)

```
1  export function random(length: number): string {
2      let text = '';
3      const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
4      for (let i = 0; i < length; i += 1) {
5          text += possible.charAt(Math.floor(Math.random() * possible.length));
6      }
7      return text;
8  }
```

RSA Key backup : the password generator

random() function

- Custom random() function 🧐
- 62-character set (lower letter & UPPER letter & numeric)

```
1  export function random(length: number): string {
2      let text = '';
3      const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
4      for (let i = 0; i < length; i += 1) {
5          text += possible.charAt(Math.floor(Math.random() * possible.length));
6      }
7      return text;
8  }
```

`Math.random()` is

RSA Key backup : the password generator

random() function

- Custom random() function 🧐
- 62-character set (lower letter & UPPER letter & numeric)

```
1  export function random(length: number): string {
2      let text = '';
3      const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
4      for (let i = 0; i < length; i += 1) {
5          text += possible.charAt(Math.floor(Math.random() * possible.length));
6      }
7      return text;
8  }
```

`Math.random()` is a Mersenne Twister

RSA Key backup : the password generator

random() function

- Custom random() function 🧐
- 62-character set (lower letter & UPPER letter & numeric)
- Not crypto random (Math.random : Mersenne twister) 🤯

```
1   export function random(length: number): string {
2       let text = '';
3       const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
4       for (let i = 0; i < length; i += 1) {
5           text += possible.charAt(Math.floor(Math.random() * possible.length));
6       }
7       return text;
8   }
```

Mersenne Twister is

RSA Key backup : the password generator

random() function

- Custom random() function 🧐
- 62-character set (lower letter & UPPER letter & numeric)
- Not crypto random (Math.random : Mersenne twister) 🤯

```
1  export function random(length: number): string {
2      let text = '';
3      const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
4      for (let i = 0; i < length; i += 1) {
5          text += possible.charAt(Math.floor(Math.random() * possible.length));
6      }
7      return text;
8  }
```

Mersenne Twister is NOT cryptographic random

RSA Key backup : the password generator

Password entropy

- 9 characters password from a 62-character set 🧐

```
randomPassword = (): string => `${random(3)}-${random(3)}-${random(3)}`
```

RSA Key backup : the password generator

<u>Password entropy</u>

- 9 characters password from a 62-character set 🧐

```
randomPassword = (): string => `${random(3)}-${random(3)}-${random(3)}`
```

Too hard to break `Math.random()` from 9 characters...

RSA Key backup : the password generator

<u>Password entropy</u>

- 9 characters password from a 62-character set 🧐

```
randomPassword = (): string => `${random(3)}-${random(3)}-${random(3)}`
```

Too hard to break `Math.random()` from 9 characters...

But don't worry.

RSA Key backup : the password generator

Password entropy

- 9 characters password from a 62-character set 🧐
- toLowerCase() after calling random() 😨

```
randomPassword = (): string => `${random(3)}-${random(3)}-${random(3)}`.toLowerCase();
```

RSA Key backup : the password generator

Password entropy

- 9 characters password from a ~~62~~ ??-character set 🧐
- toLowerCase() after calling random() 😨
  - All uppercase letters (ABC…) in the password are converted to lowercase (abc…).

```
randomPassword = (): string => `${random(3)}-${random(3)}-${random(3)}`.toLowerCase();
```

RSA Key backup : the password generator

Password entropy

- 9 characters password from a ~~62~~ 36-character set 😵‍💫
- toLowerCase() after calling random() 😨
  - All uppercase letters (ABC…) in the password are converted to lowercase (abc…).

```
randomPassword = (): string => `${random(3)}-${random(3)}-${random(3)}`.toLowerCase();
```

This means…

- This reduces the character set from 62 to 36.

RSA Key backup : the password generator

Password entropy

- 9 characters password from a ~~62~~ 36-character set
- toLowerCase() after calling random()
  - All uppercase letters (ABC…) in the password are converted to lowercase (abc…).

```
randomPassword = (): string => `${random(3)}-${random(3)}-${random(3)}`.toLowerCase();
```

This means…

- This reduces the character set from 62 to 36.
- This also introduces strong bias into the password.

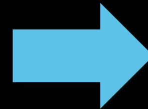
RSA Key backup : strong bias in the password

BEFORE

Lower-letter
abc...xyz

UPPER-letter
ABC...XYZ

Numeric
0...9



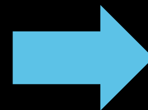
RSA Key backup : strong bias in the password

BEFORE

Lower-letter
abc...xyz

UPPER-letter
ABC...XYZ

Numeric
0...9



AFTER .toLowerCase()

Lower-letter
abc...xyz

Lower-letter (1)
abc...xyz

×2 !!

Numeric
0...9

.toLowerCase() maps UPPER → lower → lower-letter appears **twice as often** as expected

Vuln #1: Weak Custom Password Generators

Mobile Application (explained)

- 9 chars from [A-Z, a-z, 0-9] — but toLowerCase() collapses to 36
- Password space: **$2^{46.53}$** + 32.2% bias toward letters

Vuln #1: Weak Custom Password Generators

Legacy Web Application (version<4.5.0 : 2018 Sept -- 2022 Apr)

- Same toLowerCase() bias as mobile app, but random() is cryptography random
- Password space: $2^{46.53}$ + 32.2% bias toward letters

Web Application (version>=4.5.0 : 2022 May -- Present)

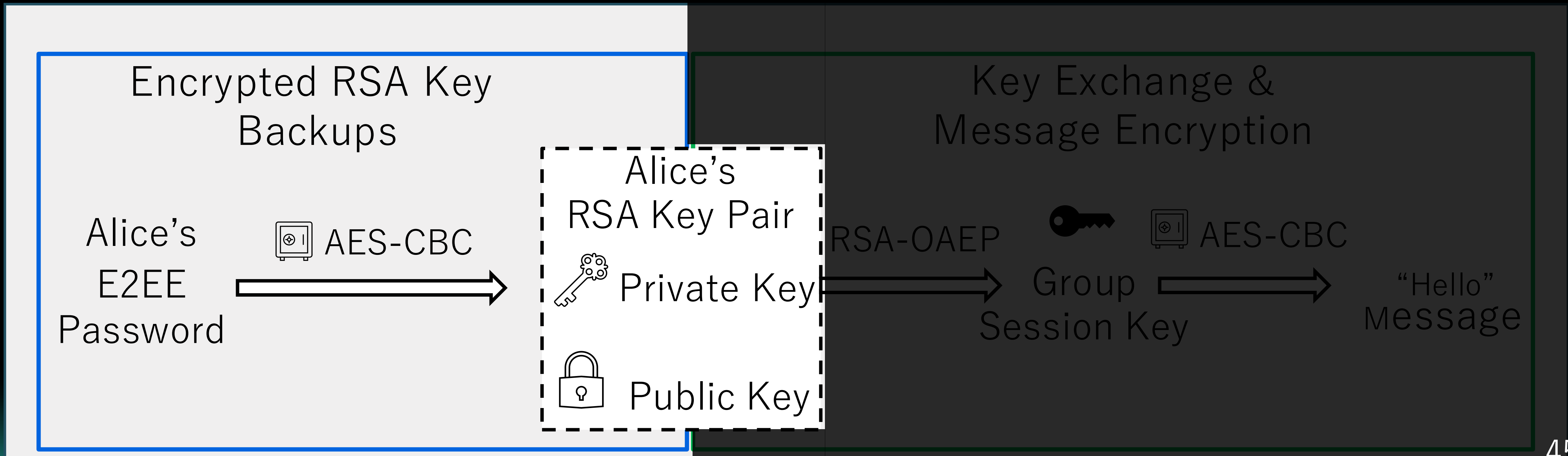
- 5 random words from a list of 1,633 words
- Password space: $2^{53.36}$

Part 3

What the Attacker Gets

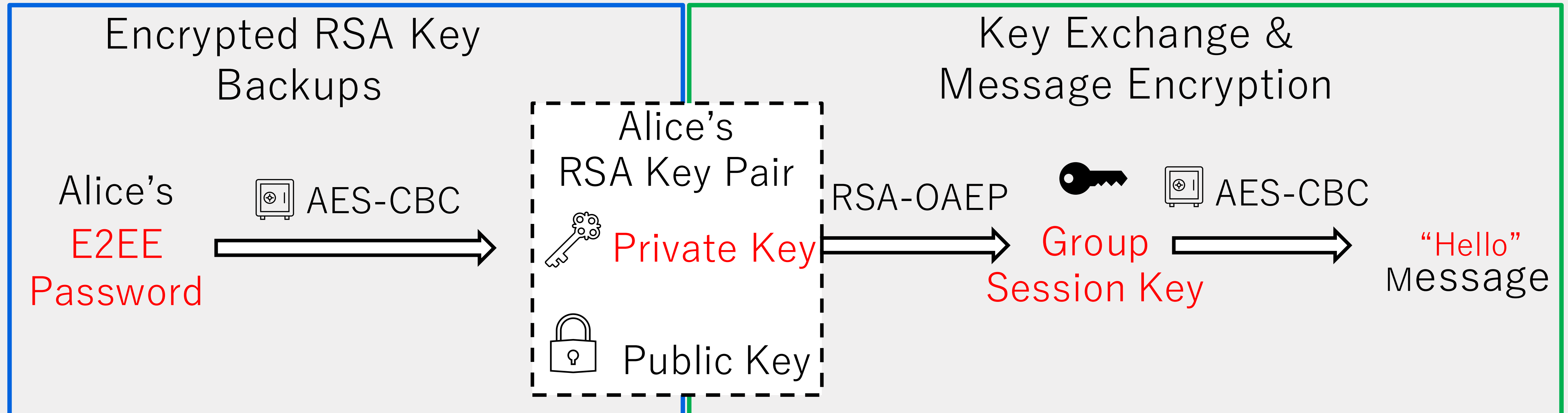
What the Attacker Gets:
Recovering the E2EE password… then what?

- The E2EE password is the "root" of the E2EE key hierarchy



What the Attacker Gets:
Recovering the E2EE password… then what?

- The E2EE password is the “root” of the E2EE key hierarchy
- Compromising all of Rocket.Chat E2EE (Read/Write encrypted message).



Is that all…?

Is that all…? No, there's more.

- Other integrity & confidentiality attacks
  - Message forgery, MITM attack

Is that all…? No, there's more.

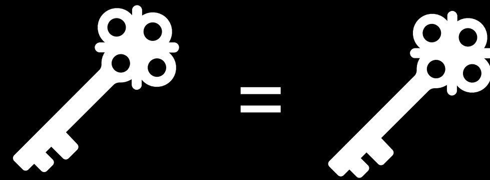
- Other integrity & confidentiality attacks
    - Message forgery, MITM attack
- Flaws in the recovery mechanism
    - The password-reset protocol does not help recover from compromise
    - The compromised-session key is shared again, can be used forever

Summary of Part2 & Part3 : Rocket.Chat E2EE can be completely broken

Full key recovery on key backups

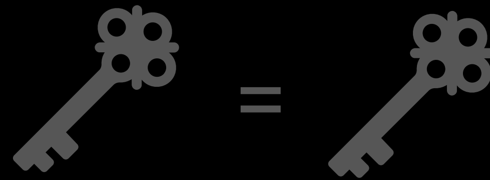
Summary of Part2 & Part3 : Rocket.Chat E2EE can be completely broken

Full key recovery on key backups



Password resets/updates do not rotate session keys

Summary of Part2 & Part3 : Rocket.Chat E2EE can be completely broken



Full key recovery on key backups

Password resets/updates do not rotate session keys

message integrity issue (forgery), key authenticity issue (MITM), and more…

Part 4
Lessons & Fixes

Lesson from OSINT: How the Fragile Design Emerged

- Patching the code is not enough — we need to understand why it happened

Public Development Discussions

- Rocket.Chat released E2EE in Sept 2018
- Opt-in and password-based encryption

Lesson from OSINT:
How the Fragile Design Emerged

- Patching the code is not enough — we need to understand why it happened

Historical Commits

- They planned to implement Signal protocol, but they finally didn't (GSoC 2017)
  - You could see a press release in 2017

Lesson from OSINT: How the Fragile Design Emerged

- Patching the code is not enough — we need to understand why it happened

Historical Commits

- They planned to implement Signal protocol, but they finally didn't (GSoC 2017)
  - You could see a press release in 2017
- The historical reasons behind this design have never been disclosed
  - Yes, this is OSS, but Rocket.Chat manages internal Jira tickets

https://github.com/RocketChat/Rocket.Chat/pull/10094#discussion_r181312128

https://github.com/RocketChat/Rocket.Chat/pull/7181

Lesson from OSINT: How the Fragile Design Emerged

- Patching the code is not enough — we need to understand why it happened

Historical Commits

- They planned to implement Signal protocol, but they finally didn't (GSoC 2017)
  - You could see a press release in 2017
- The historical reasons behind this design have never been disclosed
  - Yes, this is OSS, but Rocket.Chat manages internal Jira tickets
- So, we inferred from GitHub threads why they introduced this legacy cryptography
  - For the password-based encryption, they said:
    "This has been done to ensure that users can use any device"

https://github.com/RocketChat/Rocket.Chat/pull/10094#discussion_r181312128

https://github.com/RocketChat/Rocket.Chat/pull/7181

Disclosure Timeline & Fixes

Reported: May 2024

Fixed: Oct 2024 ~ Dec 2025

- Math.random → CSPRNG (cryptographically random)
- Broken key rotation (reused compromised keys) → proper rotation
- AES-CBC (no integrity) → AES-GCM (authenticated encryption)

Remaining issues

- Public key authenticity: still uses TOFU (Trust-on-first-use) model
  - MITM attacks by malicious server remain possible ; Rocket.Chat treats this as accepted risk

Our Research Story

2.5+ years from first impression to publication

2023

- First impression
- Deep analysis

May 2024

- Responsible Disclosure

Our Research Story

2.5+ years from first impression to publication

2023
- First impression
- Deep analysis

May 2024
- Responsible Disclosure

Oct 2025
- Accepted to academic conf

Jan 2026
- Accepted to Black Hat Asia

Apr 2026
- Today

Our Research Story

2.5+ years from first impression to publication

2023	May 2024	Oct 2025	Jan 2026	Apr 2026
• First impression • Deep analysis	• Responsible Disclosure	• Accepted to academic conf	• Accepted to Black Hat Asia	• Today

Hard Bits

- Investigate undocumented behaviors
- Analyzing all client platforms
- Writing research paper

Our Research Story

2.5+ years from first impression to publication

2023	May 2024	Oct 2025	Jan 2026	Apr 2026
• First impression • Deep analysis	• Responsible Disclosure	• Accepted to academic conf	• Accepted to Black Hat Asia	• Today

Hard Bits

- Investigate undocumented behaviors
- Analyzing all client platforms
- Writing research paper

Cumbersome (but fun) bits

- Read the open specification, open-source code
- Getting responsible disclosure right
  (Many thanks to Julio, Matheus and Diego@Rocket.Chat!)
- Proving exploitability end-to-end

Takeaways for Practitioners

1. Analyzing Real-World E2EE Systems

- Combine formal verification + source code review + OSINT
- Understand not just what is broken, but why it was built that way

Takeaways for Practitioners

1. **Analyzing Real-World E2EE Systems**
    - Combine formal verification + source code review + OSINT
    - Understand not just what is broken, but why it was built that way
2. **Detecting Spec-Implementation Gaps**
    - Documented protocol ≠ actual implementation
    - Undocumented behaviors are where critical bugs hide

Takeaways for Practitioners

1. **Analyzing Real-World E2EE Systems**
   - Combine formal verification + source code review + OSINT
   - Understand not just what is broken, but why it was built that way
2. **Detecting Spec-Implementation Gaps**
   - Documented protocol ≠ actual implementation
   - Undocumented behaviors are where critical bugs hide
3. **Replace Password-Based Architectures**
   - Move to modern key management & multiple device (e.g., OPAQUE, device-bound keys)
   - Legacy password-based encryption is a single point of failure for E2EE

Thank You

Paper: eprint.iacr.org/2025/2300

PoC: github.com/gravity-of-the-situation-rc

Appendix

Threat Model: The Malicious Server

Primary: Malicious Server (SaaS)

- Compromised admin or rogue employee
- Collects encrypted RSA keys and encrypted group keys
- E2EE **should** protect users even if the server is malicious

Secondary: Outsider (Network Attacker)

- Active MITM on network (Dolev-Yao model)
- 50% of servers expose plaintext HTTP (SHODAN scan)

How We Audited the Protocol

1. **Formal Analysis (ProVerif)**
   - Modeled protocol in symbolic logic (Dolev-Yao)
   - Detected MITM, offline attack, downgrade, replay paths
2. **Manual Source Code Review**
   - OSS code + official spec + undocumented behaviors
   - Found CBC malleability, weak password generators, no key rotation
3. **Proof of Concept**
   - Built working exploits for every attack on live test instances

Vuln #2: MITM Key Replacement Attack

Root Cause: No public key authenticity verification

Attack Flow

1. Alice requests Bob's public key from server
2. Server replaces it with adversary's key
3. Alice encrypts group key with adversary's key
4. Server decrypts group key, re-encrypts for Bob
5. Bob decrypts normally — no one detects the attack

No Key Rotation — The Group Key Lives Forever

What happens when a user changes their password?

- Password update: RSA key pair is **NOT regenerated**
- Password reset: new RSA key pair, but group key is **NOT rotated**
- The same group key remains **forever**

Impact

- Attacker with old password can decrypt ALL past **and future** messages
- This was undocumented — we discovered it in the source code

What the Malicious Server Actually Sees

Before our attack — server sees:

- Encrypted blobs it cannot read
- Metadata: who sent what, when, to which room

After our attack — server sees:

- **Every plaintext message — past, present, future**
- **Every user's RSA private key**
- **Ability to forge messages as any user**

Other Vulnerabilities We Found

1. **Downgrade Attack** — metadata not integrity-protected; attacker can strip E2EE flag and inject plaintext
2. **Message Forgery (CBC Malleability)** — AES-CBC without MAC allows 1-block ciphertext forgery using known P/C pair from message ID
3. **Replay Attack** — key reuse in the same room enables message replay

Full details, PoC code, and formal analysis in our paper

Real-World Exposure

SHODAN Scan (Jan–Feb 2025)

- Servers within official support window:
  - **50.2%** plaintext HTTP only
  - 42.6% TLS only
  - 7.3% mixed (TLS + plaintext)

Why this matters

- Outsider attacks (not just malicious server) are viable on half of all deployments

Lessons for Developers

1. **Don't roll your own crypto protocols**
   - Use Signal Protocol or MLS (RFC 9420)
2. **Don't roll your own password generators**
   - Audit entropy. Test for bugs like toLowerCase()
3. **Key rotation must cover all scenarios**
   - Lost device, compromised password, key — all need rotation

The Research Behind This Talk

- **6 vulnerabilities** found, **5 practical attacks** demonstrated
- ~2.5 years of research (2023–2026)
- Formal analysis (ProVerif) + manual code review + PoC
- Published at ACSAC 2025 (IEEE)
- Critical patches deployed by Rocket.Chat
- All PoCs available on GitHub