Caveat Implementor!
Key Recovery Attacks on MEGA

Martin R. Albrecht[1] Miro Haller[2,3] *Lenka Mareková*[4] Kenneth G. Paterson[3]

[1]King's College London
martin.albrecht@kcl.ac.uk

[2]UC San Diego
mhaller@ucsd.edu

[3]Applied Cryptography Group, ETH Zürich
kenny.paterson@inf.ethz.ch

[4]Information Security Group, Royal Holloway, University of London
lenka.marekova.2018@rhul.ac.uk

24 April 2023

Introduction

- MEGA – E2EE cloud storage and communication platform with 280M registered users

Introduction

- MEGA – E2EE cloud storage and communication platform with 280M registered users
- Previous work by Backendal, Haller and Paterson [BHP23] gave 5 attacks, the first two completely breaking confidentiality of user files

Introduction

- MEGA – E2EE cloud storage and communication platform with 280M registered users
- Previous work by Backendal, Haller and Paterson [BHP23] gave 5 attacks, the first two completely breaking confidentiality of user files
- MEGA did not implement suggested countermeasures, instead relying on validation of plaintext payloads

Introduction

- MEGA – E2EE cloud storage and communication platform with 280M registered users
- Previous work by Backendal, Haller and Paterson [BHP23] gave 5 attacks, the first two completely breaking confidentiality of user files
- MEGA did not implement suggested countermeasures, instead relying on validation of plaintext payloads
- These checks were sufficient to prevent the specific attacks, but (as we will show) not sufficient in general

Background

Our new attacks are enabled by

Background

Our new attacks are enabled by

- the continued lack of key separation and integrity protection

Background

Our new attacks are enabled by

- the continued lack of key separation and integrity protection
- an ECB encryption oracle present in MEGAdrop, a feature unrelated to the protocol under attack

Background

Our new attacks are enabled by

- the continued lack of key separation and integrity protection
- an ECB encryption oracle present in MEGAdrop, a feature unrelated to the protocol under attack
- detailed reporting of errors by the client to the server, added shortly after the patch for the attacks of [BHP23]

Background

Our new attacks are enabled by

- the continued lack of key separation and integrity protection
- an ECB encryption oracle present in MEGAdrop, a feature unrelated to the protocol under attack
- detailed reporting of errors by the client to the server, added shortly after the patch for the attacks of [BHP23]

and inspired by

- the small-order subgroup attacks on DH [vW96, LL97]

Background

Our new attacks are enabled by

- the continued lack of key separation and integrity protection
- an ECB encryption oracle present in MEGAdrop, a feature unrelated to the protocol under attack
- detailed reporting of errors by the client to the server, added shortly after the patch for the attacks of [BHP23]

and inspired by

- the small-order subgroup attacks on DH [vW96, LL97]
- the key overwriting attacks on OpenPGP [KR02, BPH22]

Bird's-eye view of the MEGA key hierarchy

Bird's-eye view of the MEGA key hierarchy

- Each user has

Bird's-eye view of the MEGA key hierarchy

- Each user has
  - a 128-bit *encryption key* $\mathtt{k_e}$ derived from their password

Bird's-eye view of the MEGA key hierarchy

- Each user has
  - a 128-bit *encryption key* $\mathtt{k}_{\mathtt{e}}$ derived from their password
  - a 128-bit *master key* $\mathtt{k}_{\mathtt{M}}$

Bird's-eye view of the MEGA key hierarchy

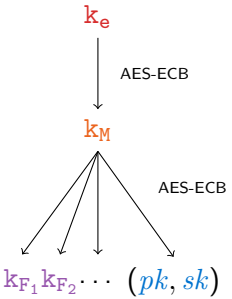
- Each user has
  - a 128-bit *encryption key* $\mathtt{k_e}$ derived from their password
  - a 128-bit *master key* $\mathtt{k_M}$
  - a 2048-bit RSA keypair (pk, sk)

Bird's-eye view of the MEGA key hierarchy

- Each user has
  - a 128-bit *encryption key* $\mathtt{k_e}$ derived from their password
  - a 128-bit *master key* $\mathtt{k_M}$
  - a 2048-bit RSA keypair (pk, sk)
  - *file encryption keys* $\mathtt{k_{F_1}}$, $\mathtt{k_{F_2}}$, ...

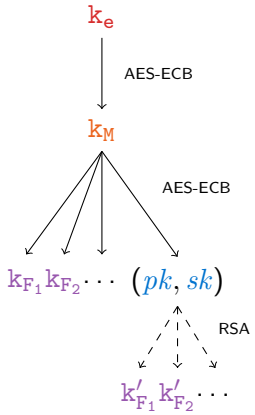
Bird's-eye view of the MEGA key hierarchy

- Each user has
  - a 128-bit *encryption key* $\mathtt{k_e}$ derived from their password
  - a 128-bit *master key* $\mathtt{k_M}$
  - a 2048-bit RSA keypair (pk, sk)
  - *file encryption keys* $\mathtt{k_{F_1}}$, $\mathtt{k_{F_2}}$, ...
- The keys are encrypted using AES-ECB as
  - $[\mathtt{k_M}]_{\mathtt{k_e}}$
  - $[\mathtt{k_F}]_{\mathtt{k_M}}$, $[sk]_{\mathtt{k_M}}$

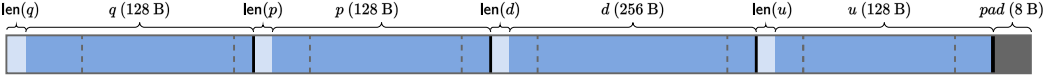


Bird's-eye view of the MEGA key hierarchy

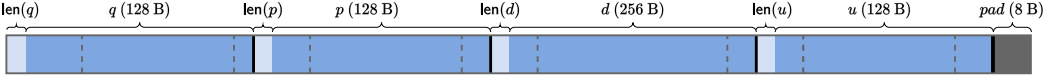
- Each user has
  - a 128-bit *encryption key* $\mathtt{k_e}$ derived from their password
  - a 128-bit *master key* $\mathtt{k_M}$
  - a 2048-bit RSA keypair (pk, sk)
  - *file encryption keys* $\mathtt{k_{F_1}}$, $\mathtt{k_{F_2}}$, ...
- The keys are encrypted using AES-ECB as
  - $[\mathtt{k_M}]_{\mathtt{k_e}}$
  - $[\mathtt{k_F}]_{\mathtt{k_M}}$, $[sk]_{\mathtt{k_M}}$
- *Shared-file encryption keys* $\mathtt{k'_F}$ are encrypted under pk using RSA



Format of sk

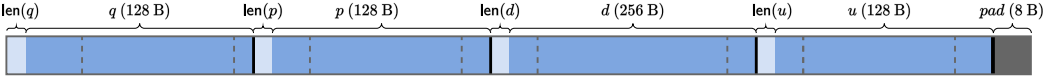


Format of sk



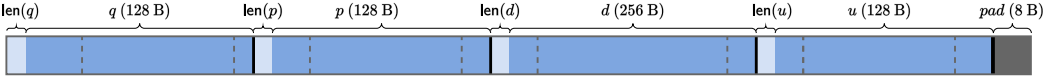
- Custom encoding of sk for RSA-CRT decryption, referred to as `privk`

Format of sk



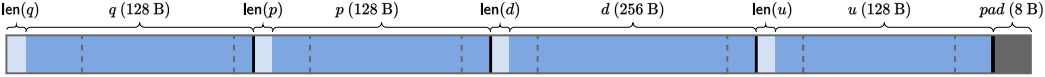
- Custom encoding of sk for RSA-CRT decryption, referred to as `privk`
  - the prime factors p, q of the RSA modulus
  - the secret exponent d
  - the value $u = q^{-1} \bmod p$

Format of sk



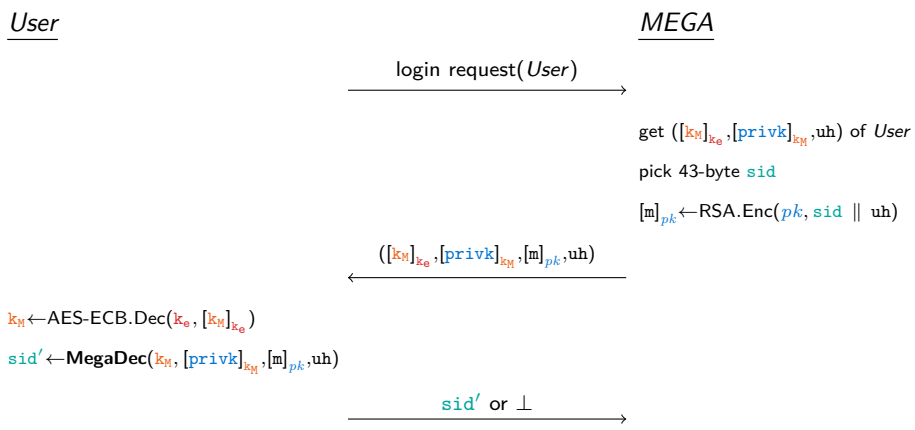
- Custom encoding of sk for RSA-CRT decryption, referred to as `privk`
  - the prime factors p, q of the RSA modulus
  - the secret exponent d
  - the value $u = q^{-1} \bmod p$
- Each value is prefixed with a 2-byte length field

Format of sk



- Custom encoding of sk for RSA-CRT decryption, referred to as `privk`
  - the prime factors p, q of the RSA modulus
  - the secret exponent d
  - the value $u = q^{-1} \bmod p$
- Each value is prefixed with a 2-byte length field
- Split into 16-byte blocks for AES-ECB

MEGA login procedure



Client decryption and parsing

MegaDec($\mathtt{k_M}$, $[\mathtt{privk}]_{\mathtt{k_M}}$, $[\mathtt{m}]_{pk}$, $\mathtt{uh}$):

1. $sk \leftarrow \mathsf{DecryptPrivk}(\mathtt{k_M}, [\mathtt{privk}]_{\mathtt{k_M}})$ // AES-ECB
2. $\mathtt{sid}' \leftarrow \mathsf{DecryptSid}(sk, [\mathtt{m}]_{pk})$ // RSA-CRT
3. Return $\mathtt{sid}'$

Client decryption and parsing

MegaDec($\mathtt{k_M}$, $[\mathtt{privk}]_{\mathtt{k_M}}$, $[\mathtt{m}]_{pk}$, $\mathtt{uh}$):

1. $sk \leftarrow \mathsf{DecryptPrivk}(\mathtt{k_M}, [\mathtt{privk}]_{\mathtt{k_M}})$ // AES-ECB
2. $\mathtt{sid}' \leftarrow \mathsf{DecryptSid}(sk, [\mathtt{m}]_{pk})$ // RSA-CRT
3. Return $\mathtt{sid}'$

Both steps rely on validity checking of the decrypted values and return distinguishable errors to the server!

Oracles from error reporting

Oracles from error reporting

Explicit errors due to validity checking:

- In DecryptSid($sk, \cdot$), a length check on the plaintext together with a legacy padding check reveal if the second byte of `m` is 0
  $\implies$ attack based on small subgroups (#2)

Oracles from error reporting

Explicit errors due to validity checking:

- In DecryptSid($sk, \cdot$), a length check on the plaintext together with a legacy padding check reveal if the second byte of $\mathtt{m}$ is 0
  $\implies$ attack based on small subgroups (#2)

Implicit errors due to bugs in the low-level library:

- In DecryptPrivk($\mathtt{k_M}, \cdot$), failure in recomputing $u' \leftarrow q^{-1} \bmod p$ reveals if $\gcd(p, q) \neq 1$
  $\implies$ attack based on modular inverses (#1)

ECB encryption oracle from MEGAdrop

ECB encryption oracle from MEGAdrop

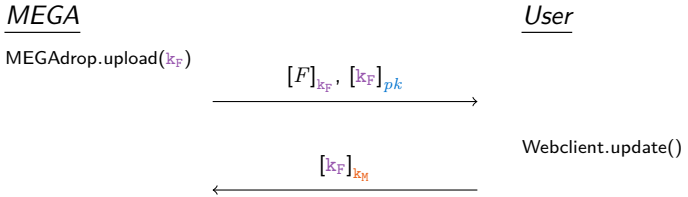
- **MEGAdrop** lets anyone upload files to a folder in the cloud storage of the recipient

ECB encryption oracle from MEGAdrop

- **MEGAdrop** lets anyone upload files to a folder in the cloud storage of the recipient
- Clients automatically re-encrypt the received shared-file keys

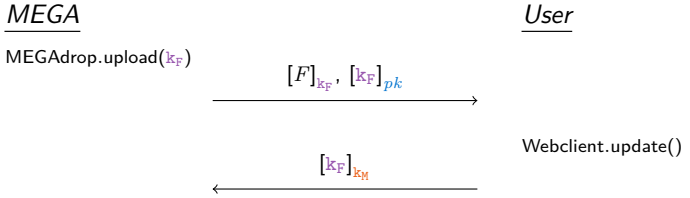
ECB encryption oracle from MEGAdrop

- **MEGAdrop** lets anyone upload files to a folder in the cloud storage of the recipient
- Clients automatically re-encrypt the received shared-file keys



ECB encryption oracle from MEGAdrop

- **MEGAdrop** lets anyone upload files to a folder in the cloud storage of the recipient
- Clients automatically re-encrypt the received shared-file keys



A malicious provider can construct an ECB encryption oracle *without user interaction* and *without leaving traces*

Attacks

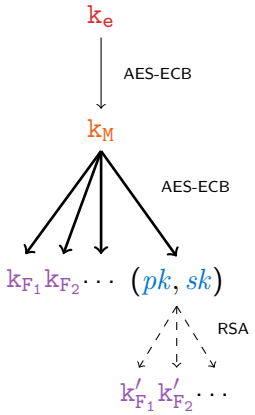
Attacks

Setting: a malicious service provider

Attacks

Setting: a malicious service provider

Goal: obtain ECB decryption ability under $\mathtt{k_M}$
$\implies$ recover sk (or any $\mathtt{k_F}$)

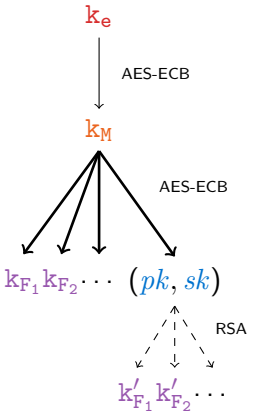


Attacks

Setting: a malicious service provider

Goal: obtain ECB decryption ability under $\mathtt{k_M}$
$\implies$ recover sk (or any $\mathtt{k_F}$)

Cost measured mainly in the number of login attempts



Attack based on modular inverses

Attack based on modular inverses

Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block, $\mathrm{OECB}_{\mathtt{k_M}}$ be the ECB encryption oracle, and $\perp_{\mathsf{inv}}$ be the error output by **MegaDec** if $\gcd(p, q) \neq 1$

Attack based on modular inverses

Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block, $\mathrm{OECB}_{\mathtt{k_M}}$ be the ECB encryption oracle, and $\perp_{\mathsf{inv}}$ be the error output by **MegaDec** if $\gcd(p, q) \neq 1$

Main idea is to

- Construct $[\mathtt{privk}^*]_{\mathtt{k_M}}$ (using $\mathrm{OECB}_{\mathtt{k_M}}$ and $[B]_{\mathtt{k_M}}$) such that

Attack based on modular inverses

Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block, $\mathrm{OECB}_{\mathtt{k_M}}$ be the ECB encryption oracle, and $\perp_{\mathsf{inv}}$ be the error output by **MegaDec** if $\gcd(p, q) \neq 1$

Main idea is to

- Construct $[\mathtt{privk}^*]_{\mathtt{k_M}}$ (using $\mathrm{OECB}_{\mathtt{k_M}}$ and $[B]_{\mathtt{k_M}}$) such that
  - $p \bmod r = 0$ for small prime r, and

Attack based on modular inverses

Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block, $\mathrm{OECB}_{\mathtt{k_M}}$ be the ECB encryption oracle, and $\perp_{\mathsf{inv}}$ be the error output by **MegaDec** if $\gcd(p, q) \neq 1$

Main idea is to

- Construct $[\mathtt{privk}^*]_{\mathtt{k_M}}$ (using $\mathrm{OECB}_{\mathtt{k_M}}$ and $[B]_{\mathtt{k_M}}$) such that
  - $p \bmod r = 0$ for small prime r, and
  - q contains B in the least-significant position

Attack based on modular inverses

Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block, $\mathrm{OECB}_{\mathtt{k_M}}$ be the ECB encryption oracle, and $\perp_{\mathsf{inv}}$ be the error output by **MegaDec** if $\gcd(p, q) \neq 1$

Main idea is to

- Construct $[\mathtt{privk}^*]_{\mathtt{k_M}}$ (using $\mathrm{OECB}_{\mathtt{k_M}}$ and $[B]_{\mathtt{k_M}}$) such that
  - $p \bmod r = 0$ for small prime r, and
  - q contains B in the least-significant position

  so that $\perp_{\mathsf{inv}} \iff q \bmod r = 0$, so by adjusting q we can learn $B \bmod r$

Attack based on modular inverses

Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block, $\mathrm{OECB}_{\mathtt{k_M}}$ be the ECB encryption oracle, and $\perp_{\mathsf{inv}}$ be the error output by **MegaDec** if $\gcd(p, q) \neq 1$

Main idea is to

- Construct $[\mathtt{privk}^*]_{\mathtt{k_M}}$ (using $\mathrm{OECB}_{\mathtt{k_M}}$ and $[B]_{\mathtt{k_M}}$) such that
  - $p \bmod r = 0$ for small prime r, and
  - q contains B in the least-significant position

  so that $\perp_{\mathsf{inv}} \iff q \bmod r = 0$, so by adjusting q we can learn $B \bmod r$
- Repeat for a set of primes r_i such that their product R is 128 bits, so we can learn $B \bmod R$ via CRT

Attack based on modular inverses

Let $[B]_{k_M}$ be the target ciphertext block, OECB_{k_M} be the ECB encryption oracle, and $\perp_{\text{inv}}$ be the error output by **MegaDec** if $\gcd(p, q) \neq 1$

Main idea is to

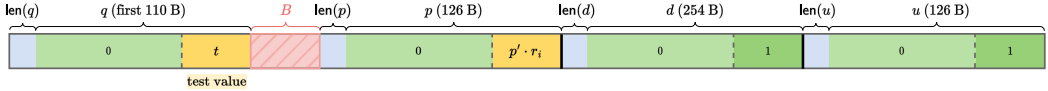
- Construct $[\texttt{privk}^*]_{k_M}$ (using OECB_{k_M} and $[B]_{k_M}$) such that
  - $p \bmod r = 0$ for small prime r, and
  - q contains B in the least-significant position

  so that $\perp_{\text{inv}} \iff q \bmod r = 0$, so by adjusting q we can learn $B \bmod r$
- Repeat for a set of primes r_i such that their product R is 128 bits, so we can learn $B \bmod R$ via CRT

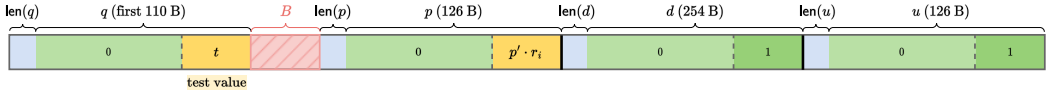
Average cost: 627 login attempts and 66-91 OECB_{k_M} queries

Attack based on modular inverses (two versions)

Attack based on modular inverses (two versions)

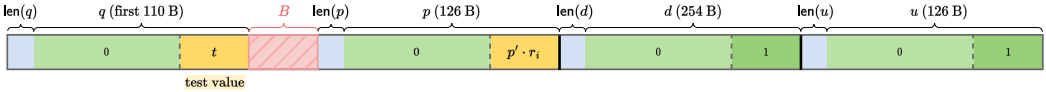


Attack based on modular inverses (two versions)

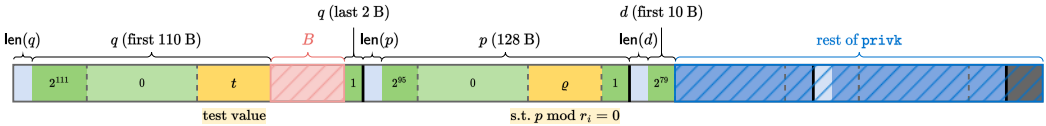


Simple version could be prevented by extra validation checks
$\Longrightarrow$ full version using more valid-looking values and parts of the original $[\texttt{privk}]_{\texttt{K}_\texttt{M}}$

Attack based on modular inverses (two versions)



Simple version could be prevented by extra validation checks
$\Longrightarrow$ full version using more valid-looking values and parts of the original $[\texttt{privk}]_{\texttt{K}_\texttt{M}}$



Attack based on small subgroups

Attack based on small subgroups

Exploit the behaviour of DecryptSid($sk, [\mathtt{m}]_{pk}$):

Attack based on small subgroups

Exploit the behaviour of DecryptSid($sk, [\mathtt{m}]_{pk}$):

```
...
m ← RSA-CRT(sk, [m]_pk)
if m[1] ≠ 00 then m' ← 00 || m
else m' ← m
m' ← m'[2 : bytelen(m')]
if bytelen(m') ≠ 255 then
    return (⊥_00, bytelen(m'))
...
```

Attack based on small subgroups

Exploit the behaviour of DecryptSid($sk, [\mathtt{m}]_{pk}$):

```
...
m ← RSA-CRT(sk, [m]_pk)
if m[1] ≠ 00 then m' ← 00 || m
else m' ← m
m' ← m'[2 : bytelen(m')]
if bytelen(m') ≠ 255 then
    return (⊥_00, bytelen(m'))
...
```

for $\mathsf{bytelen}(\mathtt{m}) = 256$,
this means $\perp_{00} \iff \mathtt{m}[1] = 00$

Attack based on small subgroups (cont'd)

Main idea is to

- Construct $[\texttt{privk}^*]_{\texttt{k}_\texttt{M}}$ such that

Attack based on small subgroups (cont'd)

Main idea is to

- Construct $[\texttt{privk}^*]_{\texttt{k}_\texttt{M}}$ such that
  - d contains B in the least-significant position

Attack based on small subgroups (cont'd)

Main idea is to

- Construct $[\texttt{privk}^*]_{\texttt{k}_\texttt{M}}$ such that
  - d contains B in the least-significant position
  - p, q are such that $(p-1)(q-1)$ has small prime factors r_i

Attack based on small subgroups (cont'd)

Main idea is to

- Construct $[\texttt{privk}^*]_{\texttt{k}_\texttt{M}}$ such that
  - d contains B in the least-significant position
  - p, q are such that $(p-1)(q-1)$ has small prime factors r_i
- Replace $[\texttt{m}]_{pk}$ with $c = g^x \bmod pq$ where

Attack based on small subgroups (cont'd)

Main idea is to

- Construct $[\texttt{privk}^*]_{\texttt{k}_\texttt{M}}$ such that
  - d contains B in the least-significant position
  - p, q are such that $(p-1)(q-1)$ has small prime factors r_i
- Replace $[\texttt{m}]_{pk}$ with $c = g^x \bmod pq$ where
  - g has order r_i and $\exists t \in \{1, \ldots, r_i - 1\}\colon g^t \bmod pq$ has second byte 00

Attack based on small subgroups (cont'd)

Main idea is to

- Construct $[\texttt{privk}^*]_{\texttt{k}_\texttt{M}}$ such that
  - d contains B in the least-significant position
  - p, q are such that $(p-1)(q-1)$ has small prime factors r_i
- Replace $[\texttt{m}]_{pk}$ with $c = g^x \bmod pq$ where
  - g has order r_i and $\exists t \in \{1, \ldots, r_i - 1\}\colon g^t \bmod pq$ has second byte 00
  - $x \in \{1, \ldots, r_i - 1\}$ is a tested value so that $\perp_{00}$ reveals $x \cdot d \equiv t \pmod{r_i}$, and we learn $B \bmod r_i$

Attack based on small subgroups (cont'd)

Main idea is to

- Construct $[\texttt{privk}^*]_{\texttt{k}_\texttt{M}}$ such that
  - d contains B in the least-significant position
  - p, q are such that $(p-1)(q-1)$ has small prime factors r_i
- Replace $[\texttt{m}]_{pk}$ with $c = g^x \bmod pq$ where
  - g has order r_i and $\exists t \in \{1, \ldots, r_i - 1\}\colon g^t \bmod pq$ has second byte 00
  - $x \in \{1, \ldots, r_i - 1\}$ is a tested value so that $\perp_{00}$ reveals $x \cdot d \equiv t \pmod{r_i}$, and we learn $B \bmod r_i$

Requires precomputation, $\approx$ 3200 logins and 15 $\text{OECB}_{\texttt{k}_\texttt{M}}$ calls

Attack based on small subgroups (cont'd)

Main idea is to

- Construct $[\texttt{privk}^*]_{\texttt{k}_\texttt{M}}$ such that
  - d contains B in the least-significant position
  - p, q are such that $(p-1)(q-1)$ has small prime factors r_i
- Replace $[\texttt{m}]_{pk}$ with $c = g^x \bmod pq$ where
  - g has order r_i and $\exists t \in \{1, \ldots, r_i - 1\}\colon g^t \bmod pq$ has second byte 00
  - $x \in \{1, \ldots, r_i - 1\}$ is a tested value so that $\perp_{00}$ reveals $x \cdot d \equiv t \pmod{r_i}$, and we learn $B \bmod r_i$

Requires precomputation, $\approx$ 3200 logins and 15 $\text{OECB}_{\texttt{k}_\texttt{M}}$ calls

The variety of errors used demonstrates the fragility of the system

Recovering the full key

Naive use of previous attacks would require up to 9 blocks to recover sk, so instead

Recovering the full key

Naive use of previous attacks would require up to 9 blocks to recover sk, so instead

1. Recover 4 blocks of q

Recovering the full key

Naive use of previous attacks would require up to 9 blocks to recover sk, so instead

1. Recover 4 blocks of q
2. Run exhaustive search for the last 16 bits (non-aligned block)

Recovering the full key

Naive use of previous attacks would require up to 9 blocks to recover sk, so instead

1. Recover 4 blocks of q
2. Run exhaustive search for the last 16 bits (non-aligned block)
3. Efficiently recover the remainder using lattice reduction

Attacks only get better

Attacks only get better

- MEGA claimed the previous attacks of Backendal, Haller and Paterson [BHP23] were not practical due to the number of login attempts needed

Attacks only get better

- MEGA claimed the previous attacks of Backendal, Haller and Paterson [BHP23] were not practical due to the number of login attempts needed
- ECB encryption oracle can be used to optimise one of the attacks on unpatched clients

Attacks only get better

- MEGA claimed the previous attacks of Backendal, Haller and Paterson [BHP23] were not practical due to the number of login attempts needed
- ECB encryption oracle can be used to optimise one of the attacks on unpatched clients
- Recover `privk` using only 2 login attempts (vs. 512 of [BHP23] and 6 of [RH23])

Responsible disclosure

Responsible disclosure

- We disclosed to MEGA in September 2022, suggesting mitigations

Responsible disclosure

- We disclosed to MEGA in September 2022, suggesting mitigations
- MEGA informed us that they were working on a larger redesign meant to

Responsible disclosure

- We disclosed to MEGA in September 2022, suggesting mitigations
- MEGA informed us that they were working on a larger redesign meant to
  - Change how private keys are stored
  - Remove the ECB encryption oracle
  - Replace the low-level crypto/bigint library

Responsible disclosure

- We disclosed to MEGA in September 2022, suggesting mitigations
- MEGA informed us that they were working on a larger redesign meant to
  - Change how private keys are stored
  - Remove the ECB encryption oracle
  - Replace the low-level crypto/bigint library
- Upgrade of clients in March 2023

Responsible disclosure

- We disclosed to MEGA in September 2022, suggesting mitigations
- MEGA informed us that they were working on a larger redesign meant to
  - Change how private keys are stored
  - Remove the ECB encryption oracle
  - Replace the low-level crypto/bigint library
- Upgrade of clients in March 2023
- MEGA awarded a bug bounty

Discussion

Discussion

- Root causes highlighted in previous work [BHP23], but it took multiple series of attacks for MEGA to agree to some of the suggested mitigations

Discussion

- Root causes highlighted in previous work [BHP23], but it took multiple series of attacks for MEGA to agree to some of the suggested mitigations
- ECB encryption oracle from an independent feature shows the fragility of the design

Discussion

- Root causes highlighted in previous work [BHP23], but it took multiple series of attacks for MEGA to agree to some of the suggested mitigations
- ECB encryption oracle from an independent feature shows the fragility of the design
- Our attacks serve as an example of key overwriting attacks, which deserve more exploration

Discussion

- Root causes highlighted in previous work [BHP23], but it took multiple series of attacks for MEGA to agree to some of the suggested mitigations
- ECB encryption oracle from an independent feature shows the fragility of the design
- Our attacks serve as an example of key overwriting attacks, which deserve more exploration

Cryptanalysis of protocols "in the wild" is needed to achieve the adoption of more secure and formally analysed cryptographic solutions in practice

Discussion

See more details in ia.cr/2023/329

Discussion

See more details in ia.cr/2023/329

Thank you for your attention. Any questions?

References

[BHP23] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. MEGA: Malleable Encryption Goes Awry. In *IEEE S&P 2023, to appear*, 2023. https://eprint.iacr.org/2022/959.

[BPH22] Lara Bruseghini, Kenneth G Paterson, and Daniel Huigens. Victory by KO: Attacking OpenPGP using key overwriting. In *ACM Conference on Computer and Communications Security (ACM CCS), to appear*, 2022.

[KR02] Vlastimil Klima and Tomas Rosa. Attack on private signature keys of the OpenPGP format, PGP(TM) programs and other applications compatible with OpenPGP. Cryptology ePrint Archive, Report 2002/076, 2002. https://eprint.iacr.org/2002/076.

[LL97] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 249–263. Springer, Heidelberg, August 1997.

[RH23] Keegan Ryan and Nadia Heninger. The hidden number problem with small unknown multipliers: Cryptanalyzing MEGA in six queries and other applications. In *PKC 2023, to appear*, 2023. https://eprint.iacr.org/2022/914.

[vW96] Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 332–343. Springer, Heidelberg, May 1996.