Demystifying Key Stretching and PAKEs

Steve “Sc00bz” Thomas
Who am I? Why am I here?
Who am I? Why am I here?

- Password cracker
- Cryptography enthusiast
- I just wanted a pw manager
  - Bugs and vulns galore
  - How would I make one?
- PHC Panelist
  - I broke Schvrch and old Makwa
Agenda

• Key Stretching
  – What?
  – Why?
  – Types
  – What goes wrong?
  – How?
  – Settings

• Password Authenticated Key Exchange (PAKE)
  – What?
  – Why?
  – Types
  – How?
  – Properties
Key Stretching

- Passwords
  - Hashing (Authentication)
  - KDF (Key Derivation Function)

- Fingerprints
  - Signal’s Safety Numbers ($2^{99.7} \rightarrow 2^{112}$)
Key Stretching – Why?

- Ashley Madison data breach (2015)
  - 36.15 million bcrypt cost 12 hashes
    - 113 H/s/GPU (GTX 980 Ti, the best at the time)
    - 89 GPU-hours/password
Key Stretching – Why?

- Ashley Madison data breach (2015)
  - 36.15 million bcrypt cost 12 hashes
    - 113 H/s/GPU (GTX 980 Ti, the best at the time)
    - 89 GPU-hours/password
  - 15.26 million salted, case-insensitive MD5 hashes\(^1\)
    - 11.2 million bcrypt cracked in 10 days
    - 73% with MD5 hashes
Key Stretching – Types

- Computationally hard
  - Amount of work done (number of blocks hashed)
    - Parallel vs Sequential
- Memory hard
  - Amount of memory used
  - Bandwidth consumed
- Cache hard
  - Random small transactions
Key Stretching – Types

• Computationally hard
  – Parallel PBKDF2
  – PBKDF2

• Memory hard
  – Argon2
  – Balloon Hashing
  – scrypt

• Cache hard
  – bcrypt
  – bcrypt
Key Stretching – How?

1) seed = H(inputs)
   a) [optional] independent seed = H(non-secret inputs)

2) work = doWork(settings, seed[, independent seed])

3) key = KDF(output size, work, seed or inputs)
Key Stretching Bugs

- md5crypt (CVE-2012-3287)
- PBKDF2 (CVE-2013-1443)
- phpass (CVE-2014-9034)
- shacrypt (CVE-2016-20013)
- bcrypt's $2$, $2a$→$2b$, $2x$, truncation, and null characters
Key Stretching Bugs

- bcrypt silently truncates at 72 bytes

```php
$passwordhash = password_hash(
    phash('P3rv4d3_extrasalt') .
    $fields['password'] .
    phash('S0ftw4r3_extrapepper'),
    PASSWORD_BCRYPT);
```

Note “phash()” is SHA-256 hex output

Source: https://twitter.com/Paul_Reviews/status/1538124477317451777
Key Stretching Bugs

- bcrypt silently truncates at 72 bytes

```php
$passhash = password_hash(
    phash('P3rv4d3_extrasalt') .
    $fields['password'] .
    phash('S0ftw4r3_extrapepper'),
    PASSWORD_BCRYPT);
```

Note “phash()” is SHA-256 hex output

Source: https://twitter.com/Paul_Reviews/status/1544735763807539200
Key Stretching Bugs

- Bouncy Castle’s bcrypt compare `.indexOf()` vs `.charAt()` (CVE-2020-28052)
- Checks the first occurrences of ./0123456789
- $2y$10$UnluckySalt./3456789..HashValueWontMatter..........$
  - 1 in 1,030,319 (for costs 11 and 12)
  - 1 in 197,153 (for all other normal costs)
Agenda

• Key Stretching
  – What?
  – Why?
  – Types
  – What goes wrong?
  – **How?**
  – Settings

• Password Authenticated Key Exchange (PAKE)
  – What?
  – Why?
  – Types
  – **How?**
  – Properties
Key Stretching – How?

1) seed = H(inputs)
2) work = doWork(settings, seed)
3) key = KDF(outSize, work, seed)
Key Stretching – How?

1) seed = H(inputs)
2) work = doWork(settings, seed)
3) key = KDF(outSize, work, seed)
PBKDF2

HMAC-Init

Password

Salt || 1

HMAC-Finish

HMAC-Finish

HMAC-Finish

XOR

XOR

XOR

Salt || 2

HMAC-Finish

HMAC-Finish

HMAC-Finish

XOR

XOR

XOR

Concatenate

...
Parallel PBKDF2

work = xorBlocks(
    pbkdf2(password, salt,
           iterations:1024,
           length:128*cost*hashLength))

output =
    pbkdf2(password, work,
           iterations:1,
           length:outputLength)
Password Settings

- **Minimum**
  - Such that an attacker gets <10 KH/s/GPU\(^{[17]}\)

- **Maximum**
  - Doesn’t take too much time \(\approx 100\) ms
  - Doesn’t use too much memory
  - Meets your needed throughput on your hardware
bscrypt Minimum Settings

- m=256 (256 KiB), t=8, p=1
- m=256 (256 KiB), t=4, p=2
- m=256 (256 KiB), t=3, p=3
- General
  - m=highest per core cache level in KiB
  - t≥max(2, 1900000/1024/m/p)
  - p≤cores

https://tobtu.com/minimum-password-settings/
bcrypt Minimum Settings

• Cost 9
  - Technically it's like “8.1” but it's an integer.
  - This should be about 5.3 kH/s on an RTX 3080 12GB.

https://tobtu.com/minimum-password-settings/
Argon2 Recommended Settings

- RFC9106
  1) Argon2id: m=2097152 (2 GiB), t=1, p=4
  2) Argon2id: m=65536 (64 MiB), t=3, p=4

https://tobtu.com/minimum-password-settings/
Argon2 Recommended Settings

- RFC9106
  1) Argon2id: m=2097152 (2 GiB), t=1, p=4
  2) Argon2id: m=65536 (64 MiB), t=3, p=4

Just kidding. Those are wildly different strengths.

https://tobtu.com/minimum-password-settings/
Argon2 Minimum Settings

- Argon2\{id,d\}: m=45056 (44 MiB), t=1, p=1
- Argon2\{id,d\}: m=18432 (18 MiB), t=2, p=1
- Argon2: m=11264 (11 MiB), t=3, p=1
- Argon2: m=8192 (8 MiB), t=4, p=1
- Argon2: m=7168 (7 MiB), t=5, p=1
- General
  - Argon2i: m\geq89062.5/(3*t-1)*\alpha, t\geq3, p=1
  - Argon2\{id,d\}: m\geq89062.5/(3*t-1)*\alpha, t\geq1, p=1

https://tobtu.com/minimum-password-settings/
scrypt Minimum Settings

- $N=2^{17} \ (128\ \text{MiB}), \ r=8, \ p=1$
- $N=2^{16} \ (64\ \text{MiB}), \ r=8, \ p=2$
- $N=2^{15} \ (32\ \text{MiB}), \ r=8, \ p=3$
- $N=2^{14} \ (16\ \text{MiB}), \ r=8, \ p=5$
- $N=2^{13} \ (8\ \text{MiB}), \ r=8, \ p=9$

General
- $N\geq 570000/r/p^*\alpha, \ r=8, \ p\geq1$

https://tobtu.com/minimum-password-settings/
PBKDF2 Settings Poll

A) 1’000’000 iterations
B) 100’000 iterations
C) 10’000 iterations
D) 1’000 iterations
PBKDF2 Minimum Settings

- PBKDF2-HMAC-BLAKE-512*
  - 170’000 iterations
- PBKDF2-HMAC-SHA-512
  - 130’000 iterations
- PBKDF2-HMAC-SHA-256
  - 350’000 iterations
- PBKDF2-HMAC-SHA-1
  - 860’000 iterations

Best but not a NIST approved hash

https://tobtu.com/minimum-password-settings/
Parallel PBKDF2 Minimum Settings

- PPBKDF2-SHA-256
  - Cost 3
- PPBKDF2-SHA-512
  - Cost 1
- Each cost is equivalent to 131’072 ($2^{17}$) iterations of PBKDF2

https://tobtu.com/minimum-password-settings/
• Key Stretching
  – What?
  – Why?
  – Types
  – What goes wrong?
  – How?
  – Settings

• Password Authenticated Key Exchange (PAKE)
  – What?
  – Why?
  – Types
  – How?
  – Properties
PAKEs

- Password authentication
- Encrypted tunnels
- Sending files
  - https://github.com/magic-wormhole
- Fighting phone spoofing
  - https://commsrisk.com/?p=35506
Why not SCRAM?

- “Salted Challenge Response Authentication Mechanism”
- Untrusted channels
  - Messages are equivalent to a password hash
Types of PAKEs

- Balanced
  - Peer-to-Peer
- Augmented (aPAKE)
  - Client-Server
- Doubly Augmented
  - Client-Server/Device-Server
- Identity
  - IoT

Don’t call these symmetric/asymmetric
Types of PAKEs

- Balanced
  - Peer-to-Peer
- Augmented (aPAKE)
  - Client-Server
- Doubly Augmented[9]
  - Client-Server/Device-Server
- Identity
  - IoT
PAKE Hierarchy

- Balanced
- Augmented
- Doubly Augmented
- Identity
Agenda

- Key Stretching
  - What?
  - Why?
  - Types
  - What goes wrong?
  - How?
  - Settings

- Password Authenticated Key Exchange (PAKE)
  - What?
  - Why?
  - Types
  - How?
  - Properties
Standard Diffie-Hellman

A:    a = random()
A:    A = a*G
A->B: A
    B: b = random()
    B: B = b*G
    B: S_B = b*A
A<-B: B
A:    S_A = a*B
Hide the Ephemeral Keys

Standard Diffie-Hellman

A: \( a = \text{random()} \)
A: \( A = a \cdot G \)
A->B: A
B: \( b = \text{random()} \)
B: \( B = b \cdot G \)
B: \( S_B = b \cdot A \)
A<-B: B
A: \( S_A = a \cdot B \)

Both: \( P = \text{hashToCurve}(H(pw)) \)
A: \( a = \text{random()} \)
A: \( A = a \cdot G + P \)
A->B: A
B: \( b = \text{random()} \)
B: \( B = b \cdot G + P \)
B: \( S_B = b \cdot (A - P) \)
A<-B: B
A: \( S_A = a \cdot (B - P) \)
Standard Diffie-Hellman

A: $a = \text{random}()$
A: $A = a \times G$
A->B: A
    B: $b = \text{random}()$
    B: $B = b \times G$
    B: $S_B = b \times A$
A<-B: B
A: $S_A = a \times B$

Both: $P = \text{hashToCurve}(H(pw))$

A: $a = \text{random}()$
A: $A = a \times P$
A->B: A
    B: $b = \text{random}()$
    B: $B = b \times P$
    B: $S_B = b \times A$
A<-B: B
A: $S_A = a \times B$
Standard Diffie-Hellman

A: \ a = \text{random()}
A: \ A = a*G
A->B: A
B: \ b = \text{random()}
B: \ B = b*G
B: \ S_B = b*A
A<-B: B
A: \ S_A = a*B

Both: \ P = \text{hashToCurve}(H(pw))
A: \ a = \text{random()}
A: \ A = a*P
A->B: A
B: \ b = \text{random()}
B: \ B = b*P
B: \ S_B = b*A
A<-B: B
A: \ S_A = a*B

Patent Expired 2017
Hide the Salt (OPRF)

C: \( P = \text{hashToCurve}(pw, id, \ldots) \)
C: \( r = \text{random()} \)
C: \( R = r \times P \)
C->S: id, R
   S: salt = \text{dbLookup}(id)
   S: R' = salt \times R
C<-S: R'
C: \text{BlindSalt} = (1/r) \times R'
BlindSalt == (1/r) \times r \times salt \times P == salt \times P
PAKEs – How?

- Balanced (Noise-NN)
- Augmented (Noise-KN)
- Doubly Augmented ("Noise-KK" but 3DH)
- Identity (Identity exchange+Balanced PAKE)
Balanced (Noise-NN)

Alice  Ephemeral Key  Bob

Ephemeral Key  ↔  Ephemeral Key
Augmented (Noise-KN)

Alice

Static Key

Ephemeral Key

Bob

Ephemeral Key

Ephemeral Key
Doubly Augmented (3DH)

Alice

Static Key

Ephemeral Key

Bob

Static Key

Ephemeral Key
## PAKE Algorithms

<table>
<thead>
<tr>
<th>Balanced</th>
<th>Augmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPace</td>
<td>(strong) AuCPace*</td>
</tr>
<tr>
<td>SPEKE&lt;sup&gt;[7]&lt;/sup&gt;</td>
<td>SPAKE2+&lt;sup&gt;[8]&lt;/sup&gt;</td>
</tr>
<tr>
<td>SPAKE2&lt;sup&gt;[8]&lt;/sup&gt;</td>
<td>SPAKE2+EE&lt;sup&gt;[9]&lt;/sup&gt;</td>
</tr>
<tr>
<td>SPAKE2-EE&lt;sup&gt;[9]&lt;/sup&gt;</td>
<td>BS-SPEKE*</td>
</tr>
<tr>
<td></td>
<td>SPAKE2-EE&lt;sup&gt;[9]&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Identity</th>
<th>Doubly Augmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIP&lt;sup&gt;[12]&lt;/sup&gt;</td>
<td>Double BS-SPEKE*</td>
</tr>
</tbody>
</table>

Hiding the: **Generator, Ephemeral Keys, Salt**
### PAKE Algorithms

<table>
<thead>
<tr>
<th>Balanced</th>
<th>Augmented</th>
<th>Doubly Augmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPace</td>
<td>(strong) AuCPace*</td>
<td>Double BS-SPEKE*</td>
</tr>
<tr>
<td>SPEKE[^7]</td>
<td>SPAKE2[^8]</td>
<td>OPAQUE[^11]</td>
</tr>
<tr>
<td></td>
<td>SPAKE2-EE[^9]</td>
<td></td>
</tr>
</tbody>
</table>

- Identity
- CHIP\[^{12}\]
- CRISP\[^{12}\]
- “FRY”

- Augmented
- B-SPEKE
- BS-SPEKE*
- SPAKE2+\[^{8}\]
- SPAKE2+EE\[^{9}\]
- SRP6a

Hiding the: Generator, Ephemeral Keys, Salt
0) Forward secrecy (every PAKE has this)
1) Prevent precomputation
2) Secure registration
3) Quantum annoying (Paper\textsuperscript{[13]}, PQCrypto 2021\textsuperscript{[14]})
4) Fragile
5) Number of trips (3 vs 4)
PAKE Properties

0) Forward secrecy (every PAKE has this)
1) Prevent precomputation
2) Secure registration
3) Quantum annoying (Paper\textsuperscript{[13]}, PQCrypto 2021\textsuperscript{[14]})
4) Fragile
5) Number of trips (3 vs 4)
Quantum Annoying

• “It is noted in [BM92] that if we assume that a discrete log pre-computation has been made for the modulus, a password attack must also compute the specific log for each entry in the password dictionary (until a match is found).”
  
  – SPEKE paper 1996\cite{7}

• “With EKE, the password $P$ is used to superencrypt such values; it is not possible to essay a discrete logarithm calculation except for all possible guesses of $P$.”
  
  – EKE paper 1992\cite{16}
PAKE Properties

1) Prevent precomputation
   - (strong) AuCPace

2) Secure registration
   - CPace
   - BS-SPEKE
   - Double BS-SPEKE

3) Quantum annoying
   - Double BS-SPEKE

4) Fragile
   - OPAQUE

5) 3 Trips
   - OPAQUE
PAKE Costs

- BS-SPEKE
  - C: fHI**ii  f***xiH
  - S: f*i      f***i
- OPAQUE-3DH
  - C: fHI**ii  f***xi
  - S: f*i      f***x

*: Scalar point multiply
x: Scalar base point multiply
H: Hash to point

- i: Field invert
  - I: Scalar invert
  - f: From bytes
message, status =
    start(myId, otherId, secret,
          pakeUser = PAKE_USER_CLIENT,
          pakeMode = PAKE_MODE_USE)

message, status =
    receiveMessage(message)

https://gist.github.com/Sc00bz/9d5c8e98143f68377e17dc82c5955f2b “pake-api.md”
sessionKey = getPakeKey()
storedSecret = getStoredSecret()

passwordKey = getPasswordKey()

https://gist.github.com/Sc00bz/9d5c8e98143f68377e17dc82c5955f2b “pake-api.md”
Cheat Sheet

- Balanced
  - CPace
- Augmented
  - BS-SPEKE
- Doubly Augmented
  - Double BS-SPEKE
- Identity
  - CHIP
- Balanced PAKEs don’t need key stretching
- bcrypt (minimums)
  - $m=256$ (256 KiB), $t=8$, $p=1$
  - $m=256$ (256 KiB), $t=4$, $p=2$
  - $m=256$ (256 KiB), $t=3$, $p=3$
- General
  - $m=\text{highest per core cache level in KiB}$
  - $t \geq \max(2, 1900000/1024/m/p)$
  - $p \leq \text{cores}$
Agenda

• Key Stretching
  – What? [Slide 5]
  – Why? [Slide 6]
  – Types [Slide 8]
  – What goes wrong? [Slide 11]
  – How? [Slide 16]
  – Settings [Slide 20]

• Password Authenticated Key Exchange (PAKE)
  – What? [Slide 30]
  – Why? [Slide 31]
  – Types [Slide 33]
  – How? [Slide 38]
  – Properties [Slide 48]
Questions?

- Twitter: @Sc00bzT
- Github: Sc00bz
- steve at tobtu.com
References

References


[15] pake-api.md https://gist.github.com/Sc00bz/9d5c8e98143f68377e17dc82c5955f2b


• BS-SPEKE secure registration
• bcrypt
• Preserve clamped scalar invert
S: Check client verifier
S: verifierS = H(...)
S: sessionKey = H(...)
S: encReg = aead_encrypt(sessionKey, reg || regMac)
C<-S: verifierS, encReg
C: Check server verifier
C: sessionKey = H(...)
C: reg || regMac = aead_encrypt(sessionKey, encReg)
C: Checks regMac == MAC(macKey, reg * G)
• The fun slides from my BSidesLV 2022 talk
• But first one info slide
Accumulators

R ^= sbox0[L >> 32 & mask];
R += sbox1[L & mask];
L ^= sbox0[R >> 32 & mask];
L += sbox1[R & mask];
...

Overlapping S-boxes

S0

S1

S0

S1
**i5-6200U**: Settings for ~5300 KH/s/GPU

Note bcrypt is better than 9x at higher run times.
i5-6500: 32 KiB L1, 256 KiB L2, 6 MiB L3

Settings for <10 kH/s/GPU

Time (ms)

Memory (KiB)

Settings for <10 kH/s/GPU

Time (ms)

Memory (KiB)

- p=1
- p=2
- p=4
i5-6500: 32 KiB L1, 256 KiB L2, 6 MiB L3

Settings for <85 H/s/GPU (equivalent to bcrypt cost 15)

- Blue line: p=1
- Red line: p=2
- Yellow line: p=3
- Green line: p=4

Time (ms) vs. Memory (KiB) chart.
Clamped Scalar Invert (Curve25519)

- Prime sub group
  - $L = 2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed$

- Normal scalar invert
  - $\text{InvertedScalar} = \text{power}_\text{mod}(\text{Scalar}, L - 2, L)$
Clamped Scalar Invert (Curve25519)

- Prime sub group
  - \( L = 2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed \)
- InvertedScalar1 = \( \text{power}_\text{mod}(\text{Scalar}, L - 2, 8 \times L) \)
- InvertedScalar2 = \( 8 \times L - \text{InvertedScalar1} \)
- checkBit(InvertedScalar1, 254) != 0
  - InvertedScalar1
  - Otherwise InvertedScalar2
Clear the Cofactor (Curve25519)

- **Prime sub group**
  - \( L = 2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed \)

- **Multiply scalar by “inverse of 8 multiplied 8”**
  - That’s “8/8” which is “1”

- **Clear** = \( 8 \times \text{power\_mod}(8, L - 2, L) \)

- **NewScalar** = Clear * Scalar (mod 8*L)