OPAQUE is NOT Magic

Steve “Sc00bz” Thomas
What is a PAKE?
PAKEs

- Password authentication
- Encrypted tunnels
- Sending files
  - https://github.com/magic-wormhole
- Fighting phone spoofing
  - https://commsrisk.com/?p=35506
How PAKEs Work

\[
a = \text{random()}
\]
\[
A = a \cdot G
\]
Hide the Ephemeral Keys

\[ a = \text{random()} \]
\[ A = a \times G + P \]
Hide the Generator

\[ a = \text{random()} \]
\[ A = a \times P \]
Myth #1

- “Zero knowledge” means the server doesn't have a password hash
Myth #1

- “Zero knowledge” means the server doesn't have a password hash
- “Augmented PAKE for authentication: we recommend the usage of OPAQUE to avoid targeted dictionary attacks on user passwords by [the company].”
Myth #2

- OPAQUE is an augmented PAKE
Types of PAKES

- Balanced
  - Peer-to-Peer
- Augmented (aPAKE)
  - Client-Server

Don’t call these symmetric/asymmetric
Types of PAKEs

- Balanced
  - Peer-to-Peer
- Augmented (aPAKE)
  - Client-Server
- Doubly Augmented
  - Client-Server/Device-Server
- Identity
  - IoT
Myth #3

• OPAQUE should be used for TLS because other PAKEs need to send the user name
C: $P = \text{hashToCurve}(pw, id, ...)$
C: $r = \text{random}()$
C: $R = r \cdot P$
C->S: id, R
   S: salt = dbLookup(id)
   S: $R' = salt \cdot R$
C<-S: R'
C: BlindSalt = $(1/r) \cdot R'$
BlindSalt == $(1/r) \cdot r \cdot salt \cdot P == salt \cdot P$
Myth #4

- OPAQUE is the only PAKE that can prevent precomputation attacks
Myth #5

- Adding an OPRF to other PAKEs makes them much slower than OPAQUE
## Costs

<table>
<thead>
<tr>
<th></th>
<th>OPAQUE-3DH</th>
<th>BS-SPEKE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>fHI<strong>ii ff</strong>*ix</td>
<td>C: fHI<strong>ii f</strong>*xiH</td>
</tr>
<tr>
<td>S</td>
<td>f<em>i ff</em>**x</td>
<td>S: f<em>i ff</em>**i</td>
</tr>
</tbody>
</table>

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

i : Field invert
I : Scalar invert
f : From bytes
Costs

OPAQUE-3DH
C: fHI**ii ff***ix
S: f*i ff***x

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

(strong) AuCPace
C: fHI**ii ff***iH
S: f*i ff***xiH

i: Field invert
I: Scalar invert
f: From bytes
Costs

**OPAQUE-3DH**
- C: fHI**ii ff***ix
- S: f*i ff***x

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

**Double BS-SPEKE**
- C: fHI**ii f***xi*i**H
- S: f*i ff****i

i: Field invert
I: Scalar invert
f: From bytes
Costs

<table>
<thead>
<tr>
<th>OPAQUE-3DH</th>
<th>OPAQUE-Noise-KN-No-AEAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>C: fHİ<strong>ii  ff</strong>*ix</td>
<td>C: fHİ<strong>ii  f</strong>ixx</td>
</tr>
<tr>
<td>S: f<em>i      ff</em>**x</td>
<td>S: f*i      ff**x</td>
</tr>
</tbody>
</table>

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

i: Field invert
I: Scalar invert
f: From bytes
PAKE Properties

- Fragile
- Quantum Annoying
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

- “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
Myth #6

• OPAQUE is the only one that can be made post quantum
Myth #7

- If you have an HSM that does Curve25519 but not Ristretto255, then you can't use Ristretto255
Myth #8

- You can’t inverse a clamped scalar while preserving the clamp
ClampedScalar =
  data \% 2**254
  + 2**254
  - data \% 8

min = 0x40000000000000000000000000000000
max = 0x7fffffffffffffff8
Clamped Scalar Inverse (Curve25519)

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

• Normal scalar inverse
  – \( \text{ScalarInverse} = \text{power}\_\text{mod}(\text{Scalar}, L - 2, L) \)
Clamped Scalar Inverse (Curve25519)

- Prime sub group
  - \( L = 2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed \)
- \( \text{ScalarInverse1} = \text{power}_mod(\text{Scalar}, L - 2, 8 \times L) \)
- \( \text{ScalarInverse2} = 8 \times L - \text{ScalarInverse1} \)
- \( \text{checkBit}(\text{ScalarInverse1}, 254) \neq 0 \)
  - \( \text{ScalarInverse1} \)
  - Otherwise \( \text{ScalarInverse2} \)
Clamped Scalar Inverse Failure

Point at infinity
“0”
Clamped Scalar Inverse Failure
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 = \( \text{power\_mod}(8, L - 2, L) \times 8 \)

• NewScalar = Clear * Scalar (mod 8*L)
“Myth #9”

- OPAQUE's Footgun
“Myth #9”

- I found one in the wild
- They now wrap ChaChaPoly1305 with HMAC-SHA512
Cheat Sheet

- Balanced
  - CPace
- Augmented
  - BS-SPEKE
- Doubly Augmented
  - Double BS-SPEKE
- Identity
  - CHIP

- Balanced PAKEs don’t need key stretching

- bscrypt (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=highest per core cache level in KiB
  - t≥max(2, 1900000/1024/m/p)
  - p≤cores
What is bscrypt?

- See BSidesLV 2022 (PasswordsCon track)
  - “bscrypt – A Cache Hard Password Hash”
Minimum Password Settings

https://tobtu.com/minimum-password-settings/
Questions?

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

Bonus Slides

- bscrypt
- BS-SPEKE secure registration
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 `bscrypt` at higher run times.
**i5-6500**: 32 KiB L1, 256 KiB L2, 6 MiB L3

Settings for <10 kH/s/GPU

- **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)
BS-SPEKE Secure Registration

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)