Enhancing Password Based Key Derivation Techniques

PasswordsCon 2014

Presented by Stephen Lombardo & Nick Parker
**SQLCipher** is an open source extension to SQLite that provides transparent 256-bit AES encryption of database files.
SQLCipher Platform Targets

C/C++, Obj-C, QT, Win32/.NET, Java, Python, Ruby, Linux, Mac OS X, iPhone/iOS, Android, Xamarin.iOS, and Xamarin.Android

Broad spectrum of use cases in both mobile and desktop devices

Our focus on securing user data where part of the key material is provided by the user
How it Works

- Transparent interaction
- On-the-fly
- Multiple crypto providers
- Standard KDF (salt + passphrase)
  - PBKDF2
  - Predates Scrypt
Current State of the Union

- SQLCipher uses 64,000 iterations when computing a key using PBKDF2
- SQLCipher previously used 4,000 iterations
How Can We Do Better

- Adaptive key derivation work factor
- Multifactor hardware token integration
Device and Platform Challenges

Our world isn't static

Ioerror commented on May 30, 2012

I think that it would be awesome if PRAGMA kdf_iter was adaptive on a per device basis. My G1 phone is crappy but my newest phone isn't - I'd like them to use a different kdf_iter value. If adaptive isn't possible, I'd prefer something randomly generated in a range - so that brute force is highly impractical before the database is acquired.

Moxie Marlinspike has done something similar to this with WhisperCore's full disk encryption. I think his implementation was adaptive by some number of seconds of computation, so the value was likely within a given distribution for a given device.
Problems with static KDF length

- Desktop and mobile hardware differ
- Technology evolves (i.e., GPU acceleration)
- Different security requirements / risk profiles / UX experience
Adaptive KDF Goals

- Fast sampling across platforms
- Compute ideal work factor limited by time
- Allow sampling to occur on any platform
Select KDF length By Security Needs

sqlcipher> PRAGMA cipher_kdf_compute;

• Sample KDF on device
• Compute iteration length based on desired runtime
• Runs by default
• Tunable for time
Tune the Sampling

```
./sqlcipher foo.db
sqlcipher> PRAGMA key = 'foo';
sqlcipher> PRAGMA cipher_kdf_compute;
cipher_kdf_compute
-------------
1,096,007
```

```
./sqlcipher foo.db
sqlcipher> PRAGMA key = 'foo';
sqlcipher> PRAGMA cipher_kdf_compute = 2.0;
cipher_kdf_compute
-------------
2,278,910
```

```
./sqlcipher foo.db
sqlcipher> PRAGMA key = 'foo';
sqlcipher> PRAGMA cipher_kdf_compute = .5;
cipher_kdf_compute
-------------
575,280
```
## PBKDF2 Sampling Results

<table>
<thead>
<tr>
<th>Device</th>
<th>Computed Work Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mac Book Pro (2.3 GHz)</td>
<td>1,161,162</td>
</tr>
<tr>
<td>iOS Simulator (7.1)</td>
<td>1,060,260</td>
</tr>
<tr>
<td>iPhone 5S</td>
<td>481,882</td>
</tr>
<tr>
<td>Android Emulator (4.4.2)</td>
<td>44,139</td>
</tr>
<tr>
<td>Android Nexus S (2.3.6)</td>
<td>72,800</td>
</tr>
<tr>
<td>Android Galaxy Tab 2 (4.2.2)</td>
<td>80,640</td>
</tr>
</tbody>
</table>
Persisting Configuration

- Previously hard coded KDF work factor
- Now persist KDF work factor
New Database Structure
Adaptive KDF Summary

Pros:
- Fast sampling across platforms
- Compute ideal work factor limited by time
- Allow sampling to occur on any platform

Cons:
- Cross device performance
- Additional complexity within SQLCipher
Multi Factor Key Derivation

- Introduce an addition factor into key derivation process
  - Something you know: Passphrase
  - something you have: Hardware Token
Stepping Back - Current KDF

- Secret database key DKey
- Random database Salt (public) DSalt
- Iterations / Work Factor (adaptive!) I
- Key Length

PBKDF2(DKey, DSalt, I, Length)
Token Requirements

- Works offline
- Simple interface (USB?)
- Widely available
- Onboard crypto
- Secure key storage
- Multi-use
- Inexpensive
Yubikey

- Long history
- Multiple form factors
- Practically indestructable
- $25 / $40
- http://www.yubico.com/
DaPlug / Plug-Up

- New entrant
- Only Available in Europe
- €8.00 ($110 Shipping!)
- http://www.daplug.io/
Common Denominator

- Onboard HMAC-SHA1 Challenge / Response API
- Programmable write-only key
Simple Implementation

- Onboard Token Key and HMAC
- Permute database salt before use
- Uses SQLCipher provider callback
Simple MFA Process

- Secret database key DKey
- Random database Salt (public) DSalt
- Iterations / Work Factor (adaptive) I
- Key Length
- Token Key TKey
- HMAC-SHA1

PBKDF2(DKey, HMAC-SHA1(TKey, DSalt), I, Length)
Results

Pros:
- Database can only be opened with token in place
- Very simple implementation
- Key can't be extracted from token
- Operating on salt does not disclose non-public data to token hardware

Cons:
- USB Required
- Custom code
- API dependencies
More Information

- http://sqlcipher.net
- http://github.com/sqlcipher/sqlcipher/tree/vfs
- http://github.com/sqlcipher/sqlcipher-mfa
Feedback

Join the SQLCipher discussion
https://discuss.zetetic.net/category/sqlcipher

Questions?