Secure Design

Of Password Storage

-john Steven
Internal CTO, Cigital, Inc.
Secure Design

SHA-3 was just released.

So, we’re done.

(haha)
The Threat Model

1) Acquiring PW DB
2) Reversing PWs from stolen booty

By capability
- Script-kiddie
- AppSec Professional
- Well-equipped Attacker
- Nation-state

Tool support (for PW cracking) is very good
Attacks Specific to PW Storage

1. Dictionary attack
2. Brute-force attack
3. Rainbow Table attack
4. Length-extension attack
5. Padding Oracle attack
6. Chosen plaintext attack
7. Crypt-analytic attack
8. Side-channel attack

Well-equipped
Nation State
Breaking the Design Down

- Plaintext
- Encrypted
- Hashed (using SHA)
- Salt and Hash
- Adaptive Hashes
  - PBKDF
  - bcrypt
  - scrypt
Hash Properties

\[ \text{digest} = \text{hash(plaintext)}; \]

- Uniqueness
- Determinism
- Collision resistance
- Non-reversibility
- Non-predictability
- Diffusion
- Lightning fast
Use a Better Hash?

SHA-1

× SHA-2

SHA-224/256

SHA-384/SHA-512

SHA-3

What property of hashes do these effect? Collisions. – Was this the problem? No.
What Does the Salt Do?

\[
salt \ || \ digest = \text{hash}(salt \ || \ \text{plaintext});
\]

De-duplicates digest texts

Adds entropy to input space*

- increases brute force time
- requires a unique table per user
Designing for Security

Preventing Acquisition
Preventing Reversing
<table>
<thead>
<tr>
<th>Threat</th>
<th>Attack Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[T1] – AppSec</td>
<td>AVA00 - Attack code running in browser</td>
</tr>
<tr>
<td></td>
<td><strong>AVA01 – Inject database and lift (bulk) credentials</strong></td>
</tr>
<tr>
<td></td>
<td>AVR01 – Use API to brute force credentials</td>
</tr>
<tr>
<td>[T4] – MitB</td>
<td>AVA10,11 – Keylogger or other scripted attack on client data/entry</td>
</tr>
<tr>
<td>[T2] – MitM</td>
<td>AVA03,04 – Interposition, Proxy, or SSL-based attack</td>
</tr>
<tr>
<td>[T5] – Concerted</td>
<td>AVA12 – Infrastructure Attack (Network operators, DNS, or CA compromise)</td>
</tr>
</tbody>
</table>
Preventing SQLi

“Best Practices”

• Separate cred./app stores
• Parameterize SQL queries
• Limit character sets

Remember hash properties?

• Fixed output size, character-set
• hash(“password’); …”) → AF68B0E4…
Attacks via Host

“Irreversible”

• Treat DB as “untrusted”
• Store secrets elsewhere
• Validate protected form
<table>
<thead>
<tr>
<th>Threat</th>
<th>Attack Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[T3] – Admin</td>
<td>AVA05 – Bulk credential export</td>
</tr>
<tr>
<td></td>
<td>AVA06 – [T1]-style attack from LAN</td>
</tr>
<tr>
<td>[T4] - MitB</td>
<td>AVA07 – Direct interaction w/ database</td>
</tr>
<tr>
<td></td>
<td>AVA08 – Interaction w/ database backups</td>
</tr>
<tr>
<td></td>
<td>AVA09 – Access to logs (SEIM, etc.)</td>
</tr>
<tr>
<td>[T5] – Concerted</td>
<td>AVR03– Stored data organization, sort, duplicate-detection</td>
</tr>
<tr>
<td></td>
<td>Dictionary Attack</td>
</tr>
<tr>
<td></td>
<td>Brute Force Attack</td>
</tr>
<tr>
<td></td>
<td>Rainbow Table Attack</td>
</tr>
<tr>
<td></td>
<td>Cryptanalytic attacks, as applicable</td>
</tr>
</tbody>
</table>
Current Industry Practices

- Plaintext
- Encrypted
- Hashed (using SHA)
- Salt and Hash
- Adaptive Hashes
  - PBKDF
  - bcrypt
  - scrypt
Hash Properties

digest = hash(plaintext);

- Uniqueness
- Determinism
- Collision resistance
- Non-reversibility
- Non-predictability
- Diffusion
- Lightning fast
Use a Better Hash?

× SHA-1
SHA-2

SHA-224/256
SHA-384/SHA-512
SHA-3

What property of hashes do these effect? Collisions. – Was this the problem? No.
Rainbow Tables: Fast but Inherent Limitations

Passwords with lengths and complexity in the white area aren’t cracked by the Rainbow Table.

Source: ophcrack

Tables are crafted for specific complexity and length.
Brute Force Time for SHA-1 hashed, mixed-case-a alphanumeric password

<table>
<thead>
<tr>
<th>Attack Description</th>
<th>Hardware</th>
<th>8 Characters</th>
<th>9 Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attacking a single hash (32 M/sec)</td>
<td>NVS 4200M GPU (Dell Laptop)</td>
<td>80 days</td>
<td>13 years</td>
</tr>
<tr>
<td>Attacking a single hash (85 M/sec)</td>
<td>$169 Nvidia GTS 250</td>
<td>30 days</td>
<td>5 years</td>
</tr>
<tr>
<td>Attacking a single hash (2.3 B/sec)</td>
<td>$325 ATI Radeon HD 5970</td>
<td>1 day</td>
<td>68 days</td>
</tr>
</tbody>
</table>
What Does the Salt Do?

\[ \text{salt} \ || \ \text{digest} = \text{hash(salt} \ || \ \text{plaintext}); \]

De-duplicates digest texts

Adds entropy to input space*

• increases brute force time

• requires a unique table per user
Can salted hashes be Attacked?

Depends on the threat-actor...

- Script-kiddie
- Some guy
- Well-equipped Attacker
- Nation-state

Attacking a table of salted hashes means building a Rainbow Table per user
Adaptive Hashes

Algorithms designed specifically to remove the “lightning-fast” property of hashes

Thus: protecting passwords from Brute Force and Rainbow Table attacks

Adaptive Hashes increase the amount of time each hash takes through iteration
PW-Based Key Derivation (PBKDF)

```java
salt || digest = PBKDF(hmac, salt, pw, c);
```

**Application Code:**

```java
salt = random.getBytes(8)
c = 10000000
key = pbkdf2(salt, pw, c, )
protected_pw = concat(salt, key)
```

**Underlying implementation:**

```java
pbkdf2(salt, pw, c, b){
    r = computeNumOutputBlock(b)
    md[1] = SHA1-HMAC(p, s || 1)
    for (i=2; i <= c; i++)
        md[i] = SHA1-HMAC(p, md[i-1])
    for (j=0; j < b; j++)
    dk_b = concat(kp[1] || kp[2] ... kp[r])
    return dk_b
}
```

Well-supported & vetted

HMAC key is password

Attacker has all entropy

What is the right ‘c’?

- NIST: 1000
- iOS4: 100000
- Modern CPU: 100000000

***SIMPLIFIED Code: see IEEE RFC2898 for details
See Java JCE Documentation for details on Java API***
bcrypt

c || salt || digest = bcrypt(salt, pw, c=);

Application Code:

```java
salt = bcrypt.genSalt(12)
c = 10000000

c, salt, key = bcrypt(salt, pw, c)
protected_pw = concat(c, salt, key)
```

Underlying implementation:

```java
bcrypt(salt, pw, c){
  d = “OrpheanBeholderScryDoubt”
  keyState = EksBlowfishSetup(c, salt, pw)

  for (int i=0, i < 64,i++){
    d = blowfish(keyState, d)
  }

  return c || salt || d
}
```

Not supported by JCE

$2^{cost}$ iterations slows hash operations

Is $2^{12}$ enough these days?

What effect does changing cost have on DB?

Outputting ‘c’ helps

Resists GPU parallelization, but not FPGA
scrypt

salt || digest = scrypt(salt, pw, N, r, p, dkLen);

**Application Code:**

<table>
<thead>
<tr>
<th>N</th>
<th>16384</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>8</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
</tr>
</tbody>
</table>

Key = scrypt(salt, pw, N, p, dkLen){
protected_pw = concat(salt, key)
}

**Underlying implementation:**

```c
scrypt(pw, salt, N, p, c){
    for (i=0, i < p1, i++)
        b[i] = PBKDF2(pw, salt, 1, p*Mflen)
    for (i=0, i < p1, i++)
        b[i] = ROMmix(b[i], N)
    return PBKDF2(pw, b[1]|b[2]|...b[p-1], 1, dkLen)
}

MF(b, N){
    x = b
    for (i=0, i < N-1, i++)
        v = /* Chain BlockMix(x) over N*/
    for (i=0, i < N-1, i++)
        j= /* Integrify(b) mod N */
        x = /* Chain BlockMix(x xor v[j]) */
    return x
}

BlockMix(r, b) ( /* Chain Salse20(b) over r) */ }
```

**Packages emerging, well-trodden than bcrypt**

**Designed to defeat FPGA attacks**

**Configurable**

- N = CPU time/Memory footprint
- r = block size
- P = defense against parallelism

***DRAMATICALLY SIMPLIFIED Code:***

See scrypt by C. Percival
See scrypt kdf-01, Josefsson for spec.
Adaptive Hash Properties

Motivations

- Resists most Threats’ attacks
  - Concerted (nation-state) can succeed w/ HW & time
- Simple implementation
- Scale CPU-difficulty w/ parameter*

Limitations

1. Top priority is convincing SecArch
   - C=10,000,000 == definition of insanity
   - May have problems w/ heterogeneous arches
2. API parameters (c=) != devops
   - Must have a scheme rotation plan
3. Attain asymmetric warfare
   - Attacker cost vs. Defender cost
4. No password update w/o user
Defender VS Attacker

Defender

Goal:
Log user in w/out > 1sec delay

Rate:
20M Users, 2M active / hr.

Burden:
validation cost * users / (sec / hr.)

Hardware:
4-16 CPUs on App Server
2-64 servers

Success Gauge:
# of machines required for AuthN

Attacker

Goal(s vary):
Crack a single password, or particular password
Create media event by cracking n passwords

Rate:
Scales w/ Capability

Burden:
Bound by PW reset interval
Population / 2 = average break = 10M

Hardware:
Custom: 320+ GPUs / card, FPGA

Success Gauge:
Days required to crack PW (ave)

Keep cost asymmetric: assure attacker cost greater than defender’s
Tradeoff Threshold

- Is more than 8 AuthN machines reasonable?
- Is less than 2 months to average crack good enough?

Keep cost asymmetric: assure attacker cost greater than defender’s
# Attacker/Defender Worksheet

- **Attacker Speedup**: 2
- **Average Pop. Yielding success**: 10000000
- **Defender CPU**: 16
- **Defender work (/ sec)**: 556

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Defender Machines</th>
<th>Days 'til Ave Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.35</td>
<td>0.6</td>
</tr>
<tr>
<td>0.05</td>
<td>1.74</td>
<td>2.9</td>
</tr>
<tr>
<td>0.1</td>
<td>3.47</td>
<td>5.8</td>
</tr>
<tr>
<td>0.15</td>
<td>5.21</td>
<td>8.7</td>
</tr>
<tr>
<td>0.2</td>
<td>6.94</td>
<td>11.6</td>
</tr>
<tr>
<td>0.25</td>
<td>8.68</td>
<td>14.5</td>
</tr>
<tr>
<td>0.3</td>
<td>10.42</td>
<td>17.4</td>
</tr>
<tr>
<td>0.35</td>
<td>12.15</td>
<td>20.3</td>
</tr>
<tr>
<td>0.4</td>
<td>13.89</td>
<td>23.1</td>
</tr>
<tr>
<td>0.45</td>
<td>15.63</td>
<td>26.0</td>
</tr>
<tr>
<td>0.5</td>
<td>17.36</td>
<td>28.9</td>
</tr>
<tr>
<td>0.55</td>
<td>19.10</td>
<td>31.8</td>
</tr>
<tr>
<td>0.6</td>
<td>20.83</td>
<td>34.7</td>
</tr>
<tr>
<td>0.65</td>
<td>22.57</td>
<td>37.6</td>
</tr>
<tr>
<td>0.7</td>
<td>24.31</td>
<td>40.5</td>
</tr>
<tr>
<td>0.75</td>
<td>26.04</td>
<td>43.4</td>
</tr>
<tr>
<td>0.8</td>
<td>27.78</td>
<td>46.3</td>
</tr>
<tr>
<td>0.85</td>
<td>29.51</td>
<td>49.2</td>
</tr>
<tr>
<td>0.9</td>
<td>31.25</td>
<td>52.1</td>
</tr>
<tr>
<td>0.95</td>
<td>32.99</td>
<td>55.0</td>
</tr>
<tr>
<td>1</td>
<td>34.72</td>
<td>57.9</td>
</tr>
<tr>
<td>1.05</td>
<td>36.46</td>
<td>60.8</td>
</tr>
<tr>
<td>1.1</td>
<td>38.19</td>
<td>63.7</td>
</tr>
<tr>
<td>1.15</td>
<td>39.93</td>
<td>66.6</td>
</tr>
<tr>
<td>1.2</td>
<td>41.67</td>
<td>69.4</td>
</tr>
<tr>
<td>1.25</td>
<td>43.40</td>
<td>72.3</td>
</tr>
</tbody>
</table>

---

A line graph titled **Defender Machines** shows the relationship between the number of defender machines and the days until average success. The graph includes a scatter plot with data points aligning along a line, indicating a linear relationship.

A similar line graph titled **Days 'til Ave Success** illustrates the correlation between the number of defender machines and the days until average success. The data points on this graph also show a linear trend.
Requiring a Key
Gains Defense
In Depth

Adaptive Hashes At Best
Strengthen a Single
Control Point

We Can Do Better with
Defense In Depth
Hmac Properties

digest = hash(key, plaintext);

Motivations

Inherits hash properties
• This includes the lightning speed

Resists all Threats’ attacks
• Brute force out of reach
  • >= 2\(^{256}\) for SHA-2
• Requires 2 kinds of attacks
  • AppServer: RMIi Host keystore
  • DB: reporting, SQLi, backup

Limitations

1. Protecting key material challenges developers
   • Must not allow key storage in DB!!!

2. Must enforce design to stop T3
   • compartmentalization and
   • privilege separation (app server & db)

3. No password update w/o user

4. Stolen key & db allows brute force
   • Rate \(~=\) underlying hash function
COMPAT/FIPS Design

version || salt || digest = hmac(key, version || salt || password)

• Hmac = hmac-sha-256
• Version per scheme
• Salt per user
• Key per site

• Add a control requiring a key stored on the App Server
• Threats who exfiltrate password table also needs to get hmac key
COMPAT/FIPS Solution

\[
\langle \text{version}_{\text{scheme}} \rangle \| \langle \text{salt}_{\text{user}} \rangle \| \langle \text{digest} \rangle := \text{HMAC}(\langle \text{key}_{\text{site}} \rangle, \langle \text{mixed construct} \rangle)
\]

\[
\langle \text{mixed construct} \rangle := \langle \text{version}_{\text{scheme}} \rangle \| \langle \text{salt}_{\text{user}} \rangle \| \langle \text{pW}_{\text{user}} \rangle
\]

- HMAC
  \[
  := \text{hmac-sha256}
  \]
- \text{key}_{\text{site}}
  \[
  := \text{PSMKeyTool}() \text{: 32B}
  \]
- \text{salt}_{\text{user}}
  \[
  := \text{SHA1PRNG() : 32B | FIPS186-2() : 32B}
  \]
- \text{pW}_{\text{user}}
  \[
  := \langle \text{governed by password fitness} \rangle
  \]

Optional:

- \langle \text{mixed construct} \rangle := \langle \text{version}_{\text{scheme}} \rangle \| \langle \text{salt}_{\text{user}} \rangle \| \langle \text{GUID}_{\text{user}} \rangle \| \langle \text{pW}_{\text{user}} \rangle
- \text{GUID}_{\text{user}}
  \[
  := \text{NOT username or available to untrusted zones}
  \]
Just Split the Digest?

No. They’re not the same.

- Lacks key space (brute force expansion)
- Steal both pieces with the same technique
- Remember 00002e09ee4e5a8fcdae7e3082c9d8ec3d304a5 ?

```bash
Permanence:code jsteven$ python split_hash_test.py -v 07606374520 -h ../hashes.txt
+ Found ['75AA8FF23C8846D1a79ae7f7452cfb272244b5ba3ce315401065d803'] verifying passwords
+ 1 total matching

Permanence:code jsteven$ python split_hash_test.py -h ../hashes_full.txt -v excal1ber -c 20
+ Found ['8FF8E2817E174C76b8597181a2ee028664aadff17a32980a5bad898c'] verifying passwords
+ 1 total matching
```
(More) Just Split the Digest

Comparing 20B PBKDF2 chunks created no collisions

No spurious hit

Worst-case:

20B chunk = 50/50 split

• 2,150,710 uniquely salted hashes
• 16 byte salt

- passwords
- mp3download
- REDROOSTER
- Dragon69
- 07606374520
- brazer1
- Bigwheel18
- Mastodon1
- Martha1a
- screaming36!

Permanence: jsteven$ grep passwords ../hashes.txt
Permanence: jsteven$ python split_hash_test.py -v passwords -h ../hashes.txt
  + Found [] matching passwords
Permanence: jsteven$ python split_hash_test.py -h ../hashes_full.txt -v excaliber -c 20
  + Found 1 ['8FF8E2817E174C76b8597181a2ee028664aadff17a32980a5bad898c'] matching passwords
  + Found 1 ['4F10C870B4E94F814fd07046b8d3bea650073e564c39596b8990d74b'] matching passwords
  + Found 1 ['EBD19B279CC64554f83f485706073fab5a112ea63143ec82a37e6d41'] matching passwords
  + Found 1 ['A4575F1E7D4C41DEc0ae49c5ce48ce4a9dbe28b9e87635e7289eb7eb'] matching passwords
  + Found 1 ['E1301662EC6349E5021c4cd8c158533aa9342ddee452f74f321ea0fa'] matching passwords
  + Found 1 ['72532DBFBF954FA1d9a068690ed1c3fc09459932be96bad5af4e1453'] matching passwords
  + Found 1 ['043EAF3F8434630d9d51324835c0891f0fbfcbeaf1f6bb6f76bc06'] matching passwords
  + Found 1 ['636BEF93F99449114785304641f419d450ce24ddfa03f4383e7593e6'] matching passwords
  + Found 1 ['A66772BEAF7A47361f6929611cc24b92b86cb84403c7773996ac49bc'] matching passwords
  + Found 1 ['8C8066C40C224A6700c50395afa1d3a87c9b76a1215193a29226e170'] matching passwords
  + Found 1 ['AD10E9DF1D23435163457052e8433cc60aa4a853ee13143db90b0456'] matching passwords
Reversible Design

version||cipher = ENC(wrapper key_site, <pw digest>)
<pw digest> = version||salt|| digest = ADAPT(version||salt_user||password)

- ENC = AES-256
- ADAPT = pbkdf2 | scrypt
- Version per scheme
- Salt per user
- Key per site
## hmac Solution Properties

<table>
<thead>
<tr>
<th>Attack</th>
<th>Resistance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Resist chosen plain text attacks</td>
<td><strong>Yes</strong>, Scheme complexity based on ((\text{salt}<em>{\text{user}} &amp; \text{pw}</em>{\text{user}}) + \text{key}_{\text{site}})</td>
</tr>
<tr>
<td>1.2 Resist brute force attacks</td>
<td><strong>Yes</strong>, (\text{Key}<em>{\text{site}} = 2^{256}, \text{salt}</em>{\text{user}} = 2^{256})</td>
</tr>
<tr>
<td>1.3 Resist D.o.S. of entropy/randomness exhaustion</td>
<td><strong>Yes</strong>, 32B on password generation or rotation</td>
</tr>
<tr>
<td>1.4 Prevent bulk exfiltration of credentials</td>
<td>Implementation detail: &lt;various&gt;</td>
</tr>
<tr>
<td>1.5 Prevent identical (&lt;\text{protected}&gt;\text{(pw)}) creation</td>
<td><strong>Yes</strong>, provided by salt</td>
</tr>
<tr>
<td>1.6 Prevent (&lt;\text{protected}&gt;\text{(pw)}) w/ credentials</td>
<td><strong>Yes</strong>, provided by (\text{Key}_{\text{site}})</td>
</tr>
<tr>
<td>1.7 Prevent exfiltration of ancillary secrets</td>
<td>Implementation detail: store (\text{Key}_{\text{site}}) on application server</td>
</tr>
<tr>
<td>1.8 Prevent side-channel or timing attacks</td>
<td><strong>N/A</strong></td>
</tr>
<tr>
<td>1.9 Prevent extension, similar</td>
<td><strong>Yes</strong>, (\text{hmac}()) construction ((\text{i_pad, o_pad}))</td>
</tr>
<tr>
<td>1.10 Prevent multiple encryption problems</td>
<td><strong>N/A</strong> ((\text{hmac}()) construction)</td>
</tr>
<tr>
<td>1.11 Prevent common key problems</td>
<td><strong>N/A</strong> ((\text{hmac}()) construction)</td>
</tr>
<tr>
<td>1.12 Prevent key material leakage through primitives</td>
<td><strong>Yes</strong>, (\text{hmac}()) construction ((\text{i_pad, o_pad}))</td>
</tr>
</tbody>
</table>
Reversible Properties

version||cipher = ENC(wrapper key_{site}, <pw digest>)
<pw digest> = version||salt|| digest = ADAPT(version||salt_{user}||password)

Motivations

• Inherits
  • “compat” solution benefits
  • Adaptive hashes’ slowness

• Requires 2 kinds of attacks
  • App Server & DB
  • Brute forcing DB out of reach (>=2^{256})
  • Stolen key can be rotated \textit{w/o} user interaction
  • Stolen DB + key still requires reversing

Limitations

1. Protecting key material challenges developers
   1. Must not allow key storage in DB!!!

2. Must enforce design to stop T3
   1. compartmentalization and
   2. privilege separation (app server & db)

3. No password update \textit{w/o} user

4. Stolen key & db allows brute force
   1. Rate \sim underlining adaptive hash
MOST IMPORTANT TOPIC
Responding once attacked

Operations
Replacing legacy PW DB

1. Protect the user’s account
   - Invalidate authN ‘shortcuts’ allowing login w/o 2\textsuperscript{nd} factors or secret questions
   - Disallow changes to account (secret questions, OOB exchange, etc.)

2. Integrate new scheme
   - Hmac(), adaptive hash (scrypt), reversible, etc.
   - Include stored with digest

3. Wrap/replace legacy scheme: (incrementally when user logs in--#4)
   - \texttt{version||salt\textsubscript{new}||protected = scheme\textsubscript{new}(salt\textsubscript{old}, digest\textsubscript{existing})} –or–
   - For reversible scheme: rotate key, version number

4. When user logs in:
   1. Validate credentials based on version (old, new); if old demand 2\textsuperscript{nd} factor or secret answers
   2. Prompt user for PW change, apologize, & conduct OOB confirmation
   3. Convert stored PWs as users successfully log in
Thank You for Your Time

Questions
Conclusions

• Without considering specific threats, the solutions misses key properties
• Understanding operations drives a whole set of hidden requirements
• Many solutions resist attack equivalently
• Adaptive hashes impose on defenders, affecting scale
• Leveraging design principles balances solution
  • Defense in depth
  • Separation of Privilege
  • Compartmentalization
TODO

• Revamp password cheat sheet
• Build/donate implementation
1. Protection schemes
2. Database storage
3. Key store ← Vital to preventing dev err
4. Password validation
5. Attack response
Additional Material for longer-format presentations

Supporting Slides
Select Source Material

Trade material

- Password Storage Cheat Sheet
- Cryptographic Storage Cheat Sheet
- PKCS #5: RSA Password-Based Cryptography Standard
- Guide to Cryptography
- Kevin Wall’s Signs of broken auth (& related posts)
- John Steven’s Securing password digests
- Graham-Cumming 1-way to fix your rubbish PW DB
- IETF RFC2898

Other work

- Spring Security, Resin
- jascrypt

Applicable Regulation, Audit, or Special Guidance

- COBIT DS 5.18 - Cryptographic key management
- Export Administration Regulations (“EAR”) 15 C.F.R.
- NIST SP-800-90A

Future work:

- Recommendations for key derivation NIST SP-800-132
- Authenticated encryption of sensitive material: NIST SP-800-38F (Draft)