Audience: OWASP crowd mainly; Developer secondly.
**History /etc/password**

```
/etc/password

root:0:0:EC90xWpTKCo
hjackson:100:100:KMEzyulaQQ2
bgoldthwa:101:101:Po2gweIEPZ2
jstev:102:500:EC90xWpTKCo
msoul:103:500:NTB4S.iQhwk
nminaj:104:500:a2N/98VTt2c
```

- Circa 1973
- 'One-way' password encryption
- `chown a+r /etc/passwd`
- DES took 1 sec per password
...bringing us to 2012

What do you see here?
How do we know what it is?
How could we figure this out?

In the news
LinkedIn
IEEE
Yahoo
...

SHA1('password') = 1e4c9b93f3f0682250b6cf8331b7ee68fd8
The Rules are based on quotes by the CEO of Merrill Lynch when Merrill brought their online banking system. Rule #1 was “Don’t be on the front page of the Wall Street Journal”. Rule #2 was “Don’t be on the front page of the New York Times.

Here the second rule addresses the fact that we should assume that every vulnerability in the system/application cannot be plugged and that the password table will get into the wild.
This talk describes the control when the password table is leaked.

Introduce the problem in terms of the threat model – the Threat Model will be used later to help show the defense-in-depth

Describe Diagram of the Threat Model
- You can try to prevent an injection or something that extracts the passwords
- You have to assume that they will get out
- We’re going to focus this talk on solutions for thwarting: T1 and T3 [T4 is outside of this discussion, but described below]
- See jOHN’s PW Storage Green field solution for a complete treatment on this diagram. T1 – T3 are excepted below
- The difference in terminology between this preso and jOHN’s is that T3 is referred to as “The Internal Threat” rather than “LAN-based threat”

The following actors participate in this threat model:

[V1] - Active System User: Compromises one’s self through use of the system
Accesses the system normally through a browser
May access the system through a compromised network (exposing them to
Engage the audience to see what they do. Gauge which people (companies) promote which solution.

The main point of this section is to look at Salt and Hash VS Adaptive Hash

The meta-point about this section is that all of the solutions in this section focus on a single control-point.

At the end of the section the point will be that Adaptive Hash

• Is more CPU intensive than a more conventional hash and will thus have a negative impact on scalability
• The notion of slowing down the attacker’s ability to brute force is commendable, but it’s debatable whether you can find a point that slows down the attacker enough while still maintaining reasonable CPU efficiency for (concurrent) logins.
• Thus, the hash-and-encrypt solution becomes a “bird in the hand” and the adaptive hash is a “two in the bush”. 

Current Industry Practices

• Plaintext
• Encrypted
• Hashed (using SHA)
• Salt and Hash
• Adaptive Hashes
  • PBKDF
  • bcrypt
  • scrypt
First to understand the Properties of the Crypto function. The properties will restrict/influence your design.
If SHA1 is breakable, then just use a stronger (bigger) hash.

The different variants of SHA-2 vary the output size. The max message size is larger for SHA-384/SHA-512.

If you believe that there is a collision-based attack (two plaintext hashing to the same value), then you won’t believe this.
Reprise or make the point about Threats Actor = Requirements.

For an unsalted hash (which is what an NTLM password is), we can use pre-computed rainbow tables.

Background on Rainbow tables (from Chandu). Note that this is a just to get you started in understanding.

---- Begin From Chandu ----
Rainbow tables are generated as "optimized lookup tables" to reverse engineer one-way hash functions.
Key idea is this -
  If I want to reverse engineer a 8-character password (alphanumeric – using upper and lower case + digits 0-9), then I would generate all possible 8-character strings and store these strings and their hashes in a table.
  If I have a hash, then I can simply do a lookup in this table to find the plaintext password.
Rainbow tables are optimized
  An 8-char password using lower/upper case letters (26+26) and digits (10), will require us to create (52p8) a table with few trillion rows. You can compute 52p8.
Rainbow Tables: Fast but Inherent Limitations

This graphic is from the ophcrach project on source forge. http://ophcrack.sourceforge.net/tables.php. What it shows is that RB tables have to be crafted with inherent limitations of size and valid-character space. Most of the tables are 99% accurate which means that not all passwords can be cracked with the table.

But, how many passwords do I need? All of them or just some of them?
Lookup table generates all possible combinations. The RB tables are about 99% accurate, but are limited in terms of the sizes and complexity of the passwords. They are, however, faster and most space efficient than a pure look up table.

“No Table” means that ophcrack doesn’t have a pre-calculated table – doesn’t mean that one can’t be generated.
What is the difference between “brute force” and “rainbow table”? Seems like the solution for breaking the salted hash is always a table and the question is whether once can pre-compute the table and store it or whether a per-user-table must be generated on the fly.
For a Salted Hash we have to use dynamically generated tables: one per salt.

How “well-equipped” is “well-equipped”?
These numbers were generated using InsiderPro’s PasswordPro password recovery program. Prices are from Amazon on 10/4/2012.

The point is that with cheap hardware, it’s possible to get generating enough hashes per second to realistically brute force a password table protected with salted-SHA1. If you combine using a Rainbow Table tool which uses fewer hashes the numbers only get better.

Justin White generated the data on his laptop. Without getting into a treatise about number of instructions needed to execute the hash operation and intricacies about which operations are available on the processor… All of the performance data is anecdotal. There are some clips that I found below, but the point is that it’s doable and with more money an attacker can construct a machine to using off-the-shelf hardware and tools from the Internet.

The ATI Radeon HD is not at the top performer. It’s an acceptable performing card. (http://www.videocardbenchmark.net/high_end_gpus.html)

---- Anecdotal discussions about how many hashes can be done per second ---
I have once made some experiences with SHA-1. A simple password hash with SHA-1 has the cost of processing a single "block" (SHA-1, like MD5, processes data by 64-
For an on-line attack, we can thwart T1 by increasing the time between tries. We can “Wait” for 1 second.

Adaptive Hashes are an off-line attack thwarting mechanism, but the problem is that there is no “wait”; there is only increased computation time.

“The added computational work makes password cracking much more difficult, and is known as key stretching.” – wikipedia (PBKDF2)
• PBKDF says that it can be any Pseudo Random Function (PRF), but the implementation of PBKDF2 only supports HMAC-SHA1

• Signature Parameters:
  - **PRF** is a pseudorandom function of two parameters with output length $hLen$ (e.g. a keyed HMAC)
  - **Password** is the master password from which a derived key is generated
  - **Salt** is a **cryptographic salt**
  - $c$ is the number of iterations desired
  - **dkLen** is the desired length of the derived key (length in bits)
  - **DK** is the generated derived key

• Invocation Actuals
  - HMAC-SHA-1 – (only PRF implemented by default Java JCE)
  - password – password
  - salt – salt
  - iOS4 uses 10,000
  - 160 – SHA-1 generates 160 bit hashes

**NOTE** that PBKDF was designed to generate a key for password-based crypto; the implication is that it’s not designed to be executed at scale. From RFC 2898:

“A key derivation function produces a derived key from a base key and other parameters. In a password-based key derivation function, the base key is a
FPGA – Field Programmable Gate Array – In short programmable hardware.
$2^{12} = 4096$

Parallelism is the key thing for the attacker.

BACKGROUND/REFERENCE:

   1. A bit unfathomable, but this pseudo-code was enlightening. cost is used in the setup of Blowfish. It’s a power of 2

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

      Application Code:
      salt = bcrypt.genSalt(12)
      c = 10000000
      c, salt, key = bcrypt(salt, pw, c)
      protected_pw = concat(c, salt, key)

      Underlying implementation:
      bcrypt(salt, pw, c){
        d = "OrpheanBeholderScryDoubt"
        keyState = Eks BlowfishSetup(c, salt, pw)
        for (int i=0, i < 64,i++){
          d = blowfish(keyState, d)
        }
        return c || salt || d
      }

      bcrypt(cost, salt, pwd)
      state = EksBlowfishSetup (cost, salt, key)
      ctext = "OrpheanBeholderScryDoubt"
      repeat (64)
      ctext = EncryptECB (state, ctext)
      return Concatenate (cost, salt, ctext)
      EksBlowfishSetup (cost, salt, key)
      state = InitState ()
      state = ExpandKey (state, salt, key)
```

2. $2^{cost}$ iterations slows hash operations
3. Is $2^{12}$ enough these days?
4. What effect does changing cost have on DB?
5. Outputting ‘c’ helps
6. Resists GPU parallelization, but not FPGA
scrypt is the next step in the “use more resources” line of password protection algorithms. Not a lot to say here except that it’s even slower than bcrypt.

Q to audience: Does that make it better? (This is a setup for the next slide)

From Wikipedia:

The scrypt function is specifically designed to hinder such attempts by raising the resource demands of the algorithm. Specifically, the algorithm is designed to use a large amount of memory compared to other password-based KDFs, making the size and the cost of a hardware implementation much more expensive, and therefore limiting the amount of parallelism an attacker can use (for a given amount of financial resources).

BACKGROUND/REFERENCES

Parameters (from the Standard)

Input:

- P Passphrase, an octet string.
- S Salt, an octet string.
- r Block size parameter.
- N CPU/Memory cost parameter, must be larger than 1, a power of 2 and
### 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

DEAL WITH MEAINTANCE CPU, Architect, and other problems
## Defender VS Attacker

### Defender

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

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

- **Burden:**
  \[ \text{validation cost} \times \text{users} / (\text{sec} / \text{hr.}) \]

- **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
Adaptive Hashes At Best
Strengthen a Single
Control Point

We Can Do Better with
Defense In Depth

Requiring a Key
Gains 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
    • $\geq 2^{128}$ 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 $\sim$ 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
Just Split the Digest?

No. They're not the same.

- Lacks key space (brute force expansion)
- Steal both pieces with the same technique
- **Remember** 000002e09ee4e5a8fcdaae7e3082c9d8ec3d304a5

```
Permanence:code jsteven$ python split_hash_test.py -v 07606374520 -h ../hashes.txt
+ Found ["75A8FF23C884601a79ae77F7452C4b272244b6b3c315491065d803"] verifying passwords
+ 1 total matching
```

```
Permanence:code jsteven$ python split_hash_test.py -h ../hashes_full.txt -v ex caliber -c 20
+ Found ["8F8E2B17E174C76b8597181a23e928664aadff17a32980a5bad898c"] verifying passwords
+ 1 total matching
```
Reversible Design

\[ \text{version|cipher = ENC(wrapper key}_{\text{site}}, <\text{pw digest}>) \]
\[ <\text{pw digest} > = \text{version|salt| digest = ADAPT(version|salt}_{\text{user}}|password) \]

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

version||cipher = ENC(wrapper key, <pw digest>)
<pw digest> = version||salt|| digest = ADAPT(version||saltuser||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 \textit{=} underlying 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 2nd 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)
   - version||salt||protected = scheme||salt||digest||mac
   - For reversible scheme: rotate key, version number

4. When user logs in:
   1. Validate credentials based on version (old, new); if old demand 2nd 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
Select Source Material

Trade material
- Password Storage Cheat Sheet
- Cryptographic Storage Cheat Sheet
- PKCS #5: PKA Password-Based Cryptography Standard
- Guide to Cryptography
- Kevin Wall's Signs of broken auth (ft related posts)
- John Steven's Securing password digests
- Graham-Cumming's Way to fix your rubbish PW DB
- IETF RFC2898

Other work
- Spring Security, Resin
- jssecpst

Apache: HTDDigest, HTTP Digest Specification, Shiro

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)
These are the threat actors, but we only care about a subset of these Threats for the purpose of this talk.

<table>
<thead>
<tr>
<th>Threat Actor</th>
<th>Attack Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>[T1] External Hacker</td>
<td>AV0 - Observe client operations</td>
</tr>
<tr>
<td></td>
<td>AV1 - Inject DB, bulk credentials lift</td>
</tr>
<tr>
<td></td>
<td>AV2 - Brute force PW w/ AuthN API</td>
</tr>
<tr>
<td></td>
<td>AV3 - AppSec attack (XSS, CSRF)</td>
</tr>
<tr>
<td>[T2] MiM</td>
<td>AV4 - Register 2 users, compare</td>
</tr>
<tr>
<td></td>
<td>AV1 - Interposition, Proxy</td>
</tr>
<tr>
<td></td>
<td>AV2 - Interposition, Proxy, SSL</td>
</tr>
<tr>
<td>[T3] Internal/Admin</td>
<td>AV3 - Timing attacks</td>
</tr>
<tr>
<td></td>
<td>AV1 - Bulk credential export</td>
</tr>
<tr>
<td></td>
<td>AV2 - [T1] style attack</td>
</tr>
<tr>
<td></td>
<td>AV3 - Direct action w/ DB</td>
</tr>
</tbody>
</table>
We’re going to look at just the secure storage requirements part of the overall solution. Threat T2 and the grey-ed out attacks for T1 are in-scope for the overall application, but are not germane to the issue of storage. T1-AV1 and T3-AV1 are germane because it’s through AV1 that the threat gets the password table.
COMPAT/FIPS Solution

<version_scheme>||<salt_user>||<digest> := HMAC(<key_site>, <mixed construct>)
<mixed construct> := <version_scheme>||<salt_user>||<pw_user>

- HMAC := hmac-sha256
- key_site := PSMKeyTool(SHA256()):32B;
- salt_user := SHA1PRNG():32B | FIPS186-2():32B;
- pw_user := <governed by password fitness>

Optional:
- <mixed construct> := <version_scheme>||<salt_user>||'':||<GUID_user>||<pw_user>
- GUID_user := NOT username or available to untrusted zones
<describe the solution properties>
The problem is that it’s difficult to rotate the key.
<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>0.30</td>
<td>9.6</td>
</tr>
<tr>
<td>0.05</td>
<td>0.50</td>
<td>2.0</td>
</tr>
<tr>
<td>0.10</td>
<td>0.50</td>
<td>2.0</td>
</tr>
<tr>
<td>0.15</td>
<td>0.20</td>
<td>8.7</td>
</tr>
<tr>
<td>0.20</td>
<td>0.80</td>
<td>13.6</td>
</tr>
<tr>
<td>0.24</td>
<td>1.63</td>
<td>28.8</td>
</tr>
<tr>
<td>0.26</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.30</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.35</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.40</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.45</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.50</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.55</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.60</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.65</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.70</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.75</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.80</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.85</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.90</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>0.95</td>
<td>3.26</td>
<td>48.5</td>
</tr>
<tr>
<td>1.00</td>
<td>3.26</td>
<td>48.5</td>
</tr>
</tbody>
</table>

The graph on the right shows the relationship between the number of defender machines and the average days until success, with a linear trend line indicating a strong positive correlation.
(More) Just Split the Digest

Comparing 20B PBKDF2 chunks created no collisions

- No spurious hit
- Worst-case: 20B chunk = 50/50 split
- 2×190,710 uniquely salted hashes
- 16 byte salt
- passwords
- mp3download
- REDROOSTER
- Dragon69
- 07606374520
- brazer1
- Bigsheel18
- Mastodon1
- Marthala
- screaming361

```
Permanence: jsteven split_hash_test.py -y passwords -h .../hashes.txt  
Permanence: jsteven python split_hash_test.py -y passwords -h .../hashes_full.txt -y excellent < 20
```

```
+ Found 1 ["MFF6EC58171F0C7690579181A12ee82B664aaef417e32990e5b50d9f4c"] matching passwords
+ Found 1 ["4F184CE70B64E0A9F614f0B7666B3b3bafaf50b7f1f46c39f568b99ab7f490"] matching passwords
+ Found 1 ["80B19276CC9555AF93F4B85476F403F05112ee03142ecB2a37bedd41"] matching passwords
+ Found 1 ["A6575917D82041DEc6ac6b40e2c80b8he7833e7289ed767e"] matching passwords
+ Found 1 ["E1411682E86390692c2c64c15ns3333a2f42f4f42e26ed8fa"] matching passwords
+ Found 1 ["752320B85294A87d46b69e1f3ec6e9545392be98b8a5a9e4453"] matching passwords
+ Found 1 ["044FAF31842464910c128435cc899f9b9c9e65ebd764bc06"] matching passwords
+ Found 1 ["0368EF39034401149853034c414d5450a25aad6a4388e37593d68"] matching passwords
+ Found 1 ["A657728AA7A473161f6929611c24892b68c844a0737739968c496b"] matching passwords
+ Found 1 ["BC806C4C0224670c5b9395e9a5d3a87c9925ad2155539a292c26178"] matching passwords
+ Found 1 ["A018E5DF1D23455164578052e8433c68aae4b53ee1134db989e843e"] matching passwords