



Securing Password Storage

Increasing Resistance to Brute Force Attacks

-jOHN (Steven)
Internal CTO


 @m1splacedsoul

<http://mobro.co/m1splacedstache>

Chandu Ketkar
Technical Manager

 @cketkar

Scott Matsumoto
Principal Consultant

 @smatsumoto



v0.4

History /etc/passwd

etc/passwd

root:0:0:EC90xWpTKCo

hjackman:100:100:KMEzyu1aQQ2

bgoldthwa:101:101:Po2gweIEPZ2

jsteven:102:500:EC90xWpTKCo

msoul:103:500:NTB4S.iQhwk

nminaj:104:500:a2N/98VTt2c

- Circa 1973
- 'one-way' password encryption
- `chmod a+r /etc/passwd`
- DES took 1 sec per password

...bringing us to 2012

00000fac2ec84586f9f5221a05c0e9acc3d2e670
0000022c7caab3ac515777b611af73afc3d2ee50
deb46f052152cfed79e3b96f51e52b82c3d2ee8e
00000dc7cc04ea056cc8162a4cbd65aec3d2f0eb
00000a2c4f4b579fc778e4910518a48ec3d2f111
b3344eaec4585720ca23b338e58449e4c3d2f628
674db9e37ace89b77401fa2bfe456144c3d2f708
37b5b1edf4f84a85d79d04d75fd8f8a1c3d2fbde
00000e56fae33ab04c81e727bf24bedbc3d2fc5a
0000058918701830b2cca174758f7af4c3d30432
000002e09ee4e5a8fcdae7e3082c9d8ec3d304a5
d178cbe8d2a38a1575d3feed73d3f033c3d304d8
00000273b52ee943ab763d2bb3d83f5dc3d30904

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

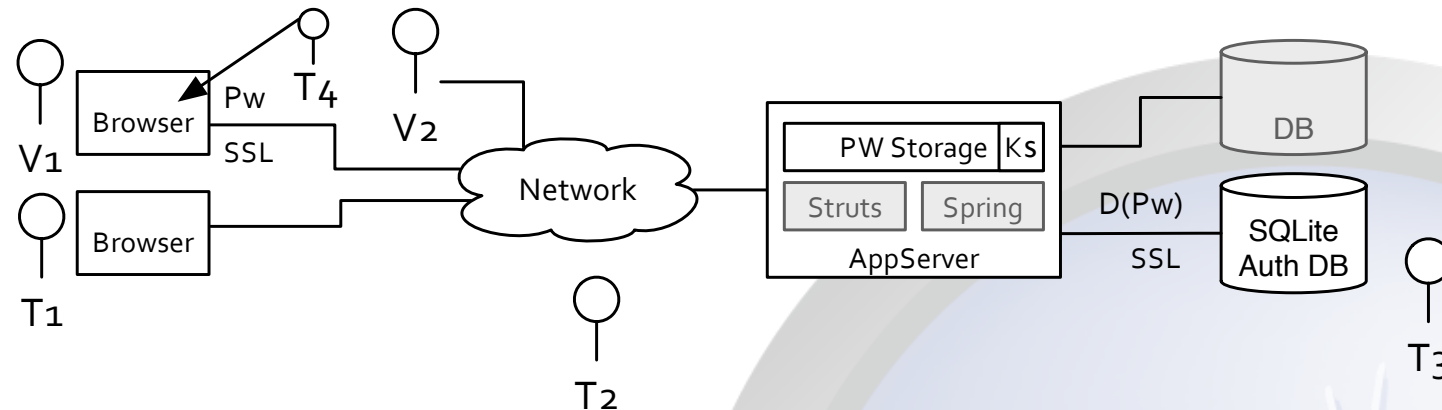


Golden Rules

- #1 – Don't be on the front page of InfoWeek
- #2 – Have a great story when you're on the front page of InfoWeek

Your passwords
WILL be
extracted from
your system

The Threat Model



1) Acquiring PW DB

2) Reversing PWs from stolen booty

- 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




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

Current Industry Practices



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?

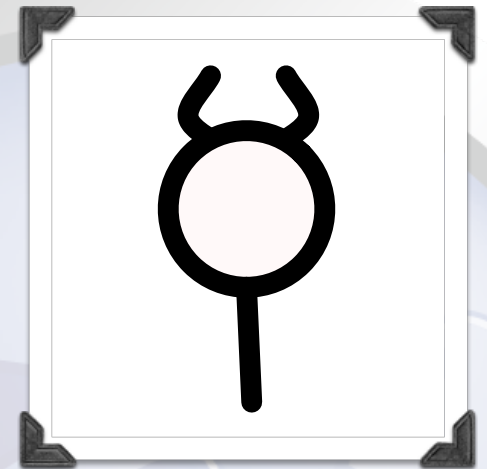
No.

Can We Successfully Attack a Hash?

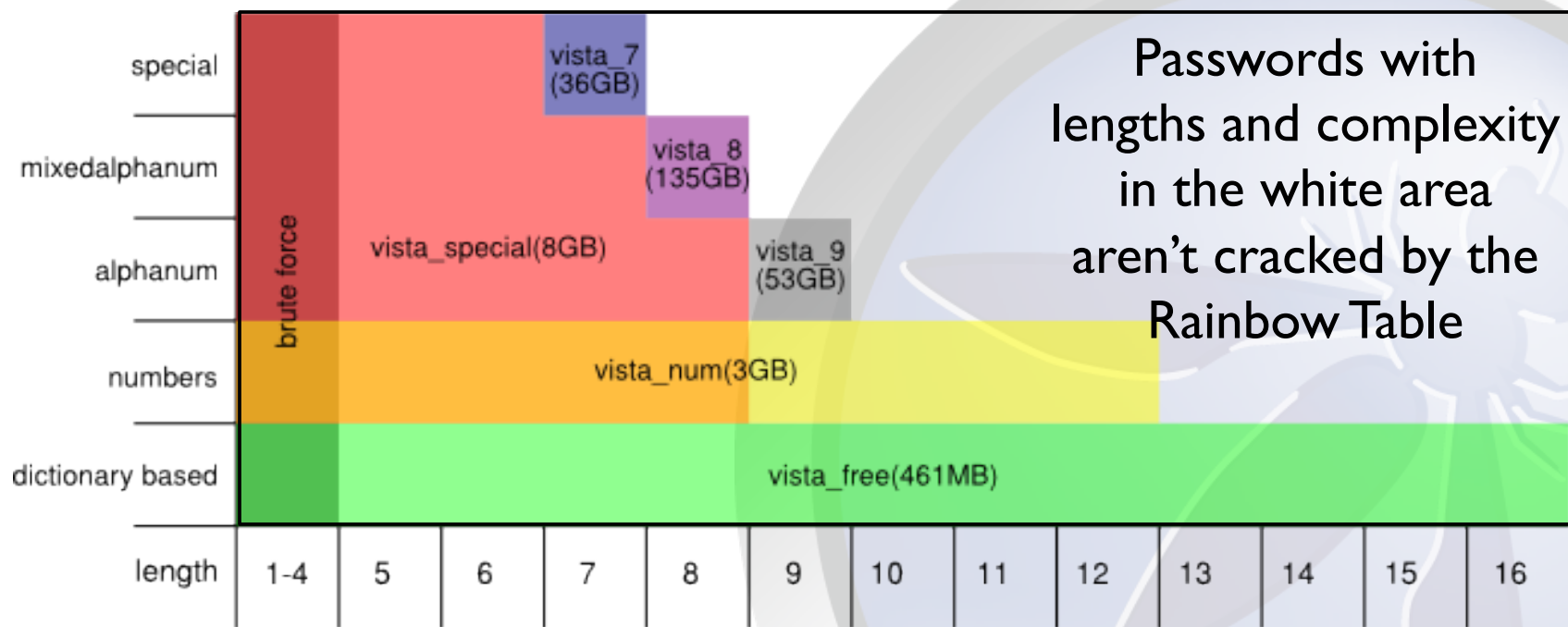
Depends on the threat-actor...

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

Is the algorithm supported by a
tool?



Rainbow Tables: Fast but Inherent Limitations



Source: ophcrack

Tables are crafted for specific complexity and length

Table Sizes

Search Space	Lookup Table (Brute Force)	Rainbow Table (NTLM hashes)
307,000 word dictionary	16 MB	461 MB
(a-z A-Z 0-9) ⁴	338 MB	8.0 GB
(a-z A-Z 0-9) ⁵	21 GB	8.0 GB
(a-z A-Z 0-9) ⁶	1.3 TB	8.0 GB
(a-z A-Z 0-9) ⁷	87 TB	8.0 GB
(a-z A-Z 0-9) ⁸	5,560 TB	134.6GB
(a-z A-Z 0-9) ⁹	357,000 TB	No table
(a-z A-Z 0-9) ¹⁰	22,900,149 TB	No table

What Does the Salt Do?

```
salt || digest = hash(salt || 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



Per User Table Building

Brute Force Time for SHA-1 hashed,
mixed-case-a alphanumeric password

		8 Characters	9 Characters
Attacking a single hash (32 M/sec)	NVS 4200M GPU (Dell Laptop)	80 days	13 years
Attacking a single hash (85 M/sec)	\$169 Nvidia GTS 250	30 days	5 years
Attacking a single hash (2.3 B/sec)	\$325 ATI Radeon HD 5970	1 day	68 days



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

Adaptive Hashes

PW-Based Key Derivation (PBKDF)

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

Application Code:

```
salt = random.getBytes(8)
c = 10000000

key = pbkdf2(salt, pw, c, )
protected_pw = concat(salt, key)
```

Underlying implementation:

```
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++)
    kp[j] = xor(md[1] || md[2]...md[j])

  dkb = concat(kp[1] || kp[2] ... kp[r])
  return dkb
}
```

Well-supported & vetted

HMAC key is password

Attacker has all entropy

What is the right 'c'?

- NIST: 1000
- iOS4: 10000
- Modern CPU: 10000000

***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:

```
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 = 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:

```
N = 16384
r = 8
P = 1
Key=scrypt(salt, pw, N, p, dkLen){
protected_pw = concat(salt, key)
```

Underlying implementation:

```
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 ($c=$) \neq 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

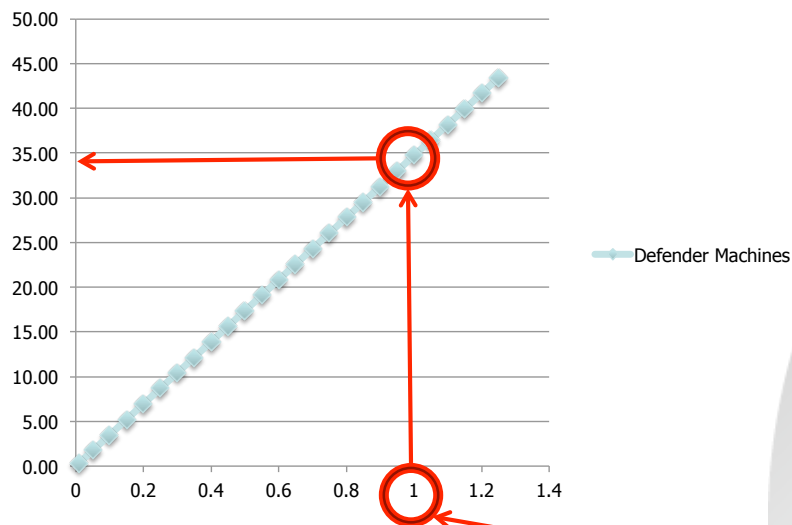
Hardware: Custom: 320+ GPUs / card, FPGA

Success Gauge: Days required to crack PW (ave)

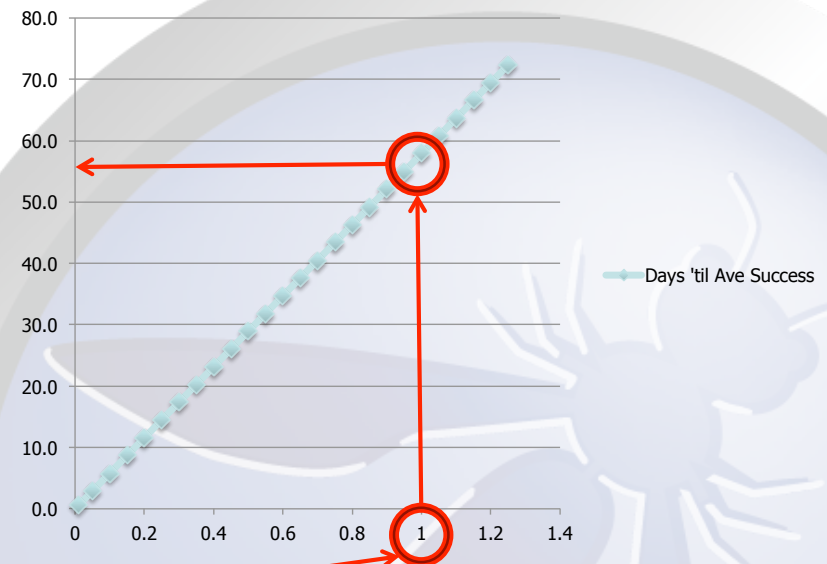
Keep cost asymmetric: assure attacker cost greater than defender's

Tradeoff Threshold

**Machines Required to Conduct Login
(@ full load)**



Days (average) Until Attacker Gets PW



- 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

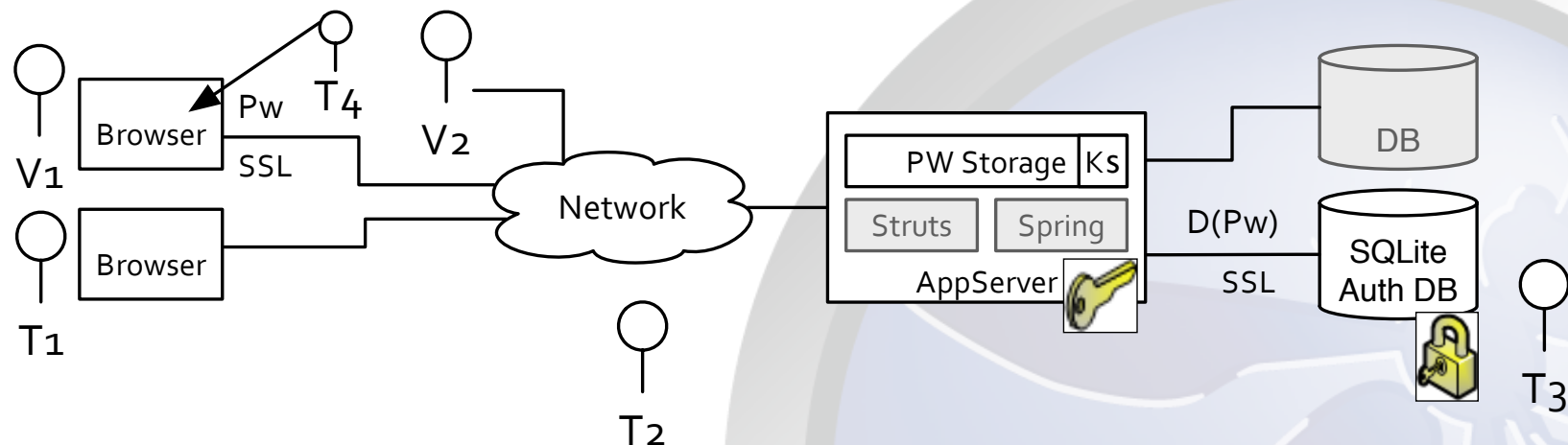
- Brute force out of reach
 - $\geq 2^{256}$ for SHA-2
- Requires 2 kinds of attacks
 - AppServer: RMIi Host keystore
 - DB: reporting, SQLi, backup

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 \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 000002e09ee4e5a8fcdae7e3082c9d8ec3d304a5 ?

```
Permanence:code jsteven$ python split_hash_test.py -v 07606374520 -h ../hashes.txt
```

```
+ Found ['75AA8FF23C8846D1a79ae7f7452cfb272244b5ba3ce315401065d803'] verifying passwords
```

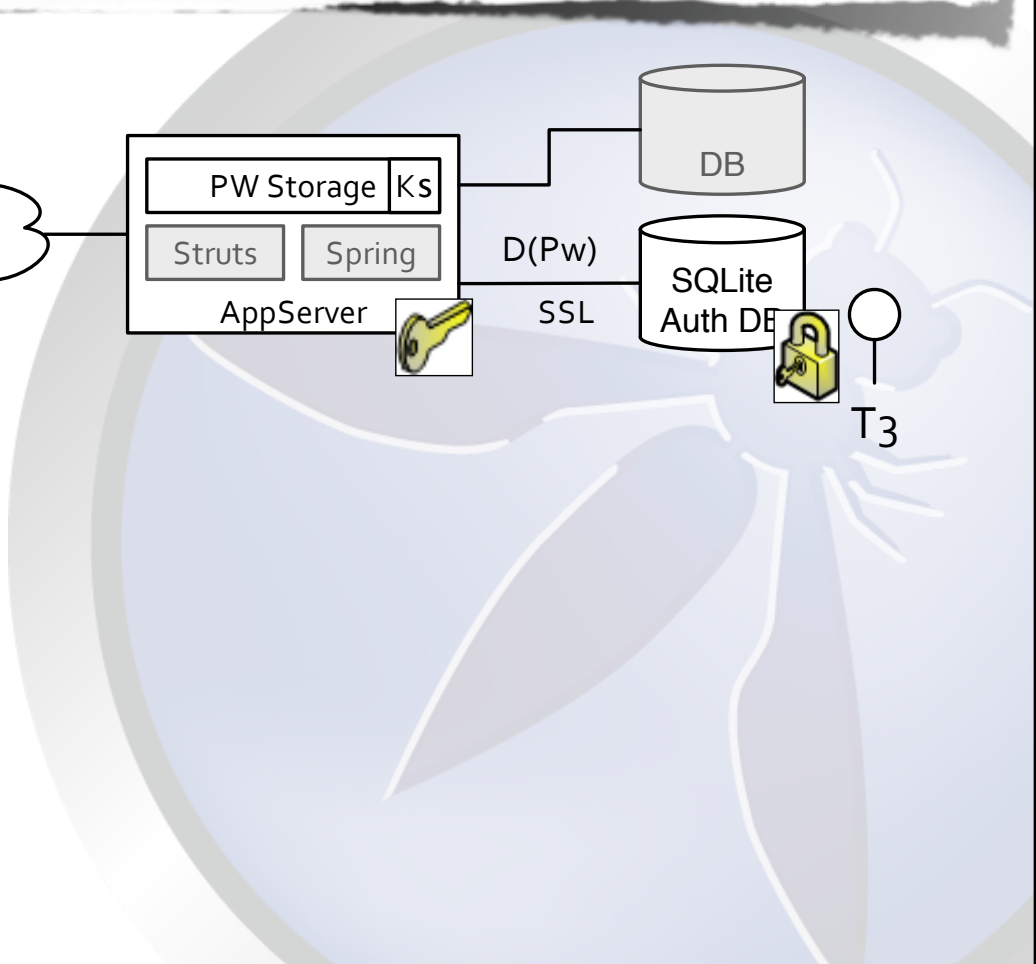
```
+ 1 total matching
```

```
Permanence:code jsteven$ python split_hash_test.py -h ../hashes_full.txt -v excal1ber -c 20
```

```
+ Found ['8FF8E2817E174C76b8597181a2ee028664aadff17a32980a5bad898c'] verifying passwords
```

```
+ 1 total matching
```

```
version||cipher = ENC(wrapper keysite, <pw digest>)
<pw digest> = version||salt|| digest = ADAPT(version||saltuser||password)
```



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

Reversible Properties

```
version||cipher = ENC(wrapper keysite, <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 ($\geq 2^{256}$)
 - Stolen key can be rotated **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 w/o user
4. Stolen key & db allows brute force
 1. Rate \sim 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||saltnew||protected = schemenew(saltold, digestexisting)` –Or–
 - 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

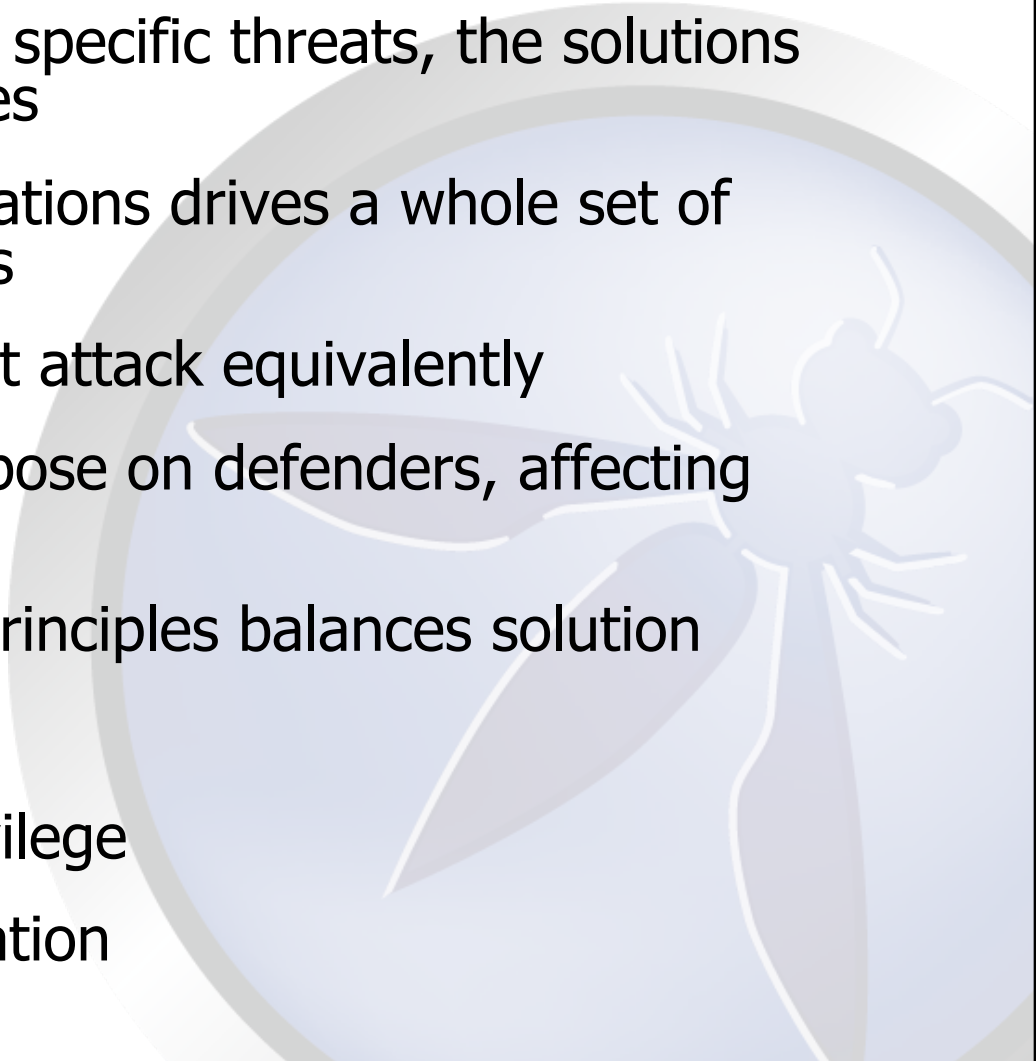


Thank You for Your
Time

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
- 



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](#)

Apache: [HTDigest](#), [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 \(Draft\)](#)

Threat Actors

Threat Actor	Attack Vector
[T1] External Hacker	AV0 - Observe client operations
	AV1 - Inject DB, bulk credentials lift
	AV2 - Brute force PW w/ AuthN API
	AV3 - AppSec attack (XSS, CSRF)
	AV4 - Register 2 users, compare
[T2] MiM	AV1 - Interposition, Proxy
	AV2 - Interposition, Proxy, SSL
	AV3 - Timing attacks
[T3] Internal/Admin	AV1 - Bulk credential export
	AV2 - [T1] style attack
	AV3 - Direct action w/ DB

Stored Passwords Requirements

Threat Actor	Attack Vector
[T1] External Hacker	AV0 - Observe client operations
	AV1 - Inject DB, bulk credentials lift
	AV2 - Brute force w/ AuthN API
	AV3 - AppSec attack (XSS, CSRF)
[T2] MiM	AV4 - Register 2 users, compare
	AV1 - Interposition, Proxy
	AV2 - Interposition, Proxy, SSL
[T3] Internal/Admin	AV3 - Timing attacks
	AV1 - Bulk credential export
	AV2 - [T1] style attack
	AV3 - Direct action w/ DB

Attack Vectors should be broken out by 1) acquisition of PW DB and 2) reversing the DB.

COMPAT/FIPS Solution

`<versionscheme>||<saltuser>||<digest> := HMAC(<keysite>, <mixed construct>)`
`<mixed construct> := <versionscheme>||<saltuser>||<pwuser>`

- HMAC := hmac-sha256
- key_{site} := PSMKeyTool(SHA256()):32B;
- salt_{user} := SHA1PRNG():32B | FIPS186-2():32B;
- pw_{user} := <governed by password fitness>

Optional:

`<mixed construct> := <versionscheme>||<saltuser>||':'||<GUIDuser>||<pwuser>`
`GUIDuser := NOT username or available to untrusted zones`

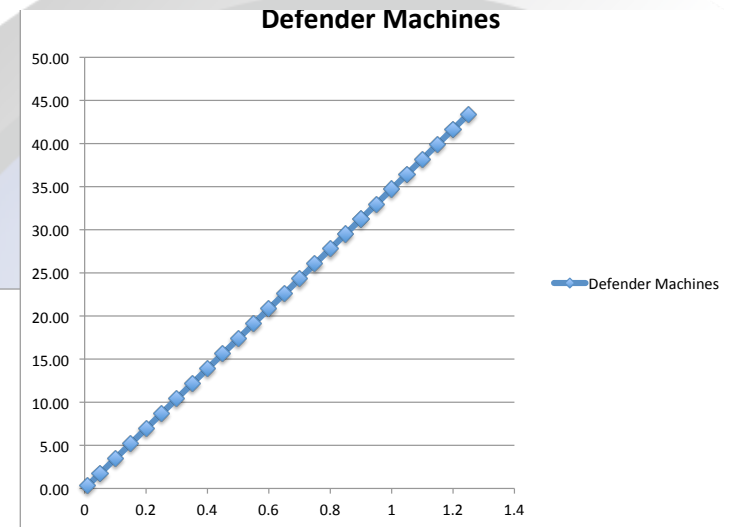
hmac Solution Properties

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

Attacker/Defender Worksheet

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

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



(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 excal1ber -c 20
```

```
+ Found 1 ['8FF8E2817E174C76b8597181a2ee028664aadff17a32980a5bad898c'] matching passwords
```

```
+ Found 1 ['4F10C870B4E94F814fd07046b8d3bea650073e564c39596b8990d74b'] matching passwords
```

```
+ Found 1 ['EBD19B279CC64554f83f485706073fab5a112ea63143ec82a37e6d41'] matching passwords
```

```
+ Found 1 ['A4575F1E7D4C41DEc0ae49c5ce48ce4a9dbe28b9e87635e7289eb7eb'] matching passwords
```

```
+ Found 1 ['E1301662EC6349E5021c4cd8c158533aa9342dde452f74f321ea0fa'] matching passwords
```

```
+ Found 1 ['72532DBFBF954FA1d9a068690ed1c3fc09459932be96bad5af4e1453'] matching passwords
```

```
+ Found 1 ['043EAF3FE8434630d9d513284835c0891f0fbfcbeaf1f6bb6f76bc06'] matching passwords
```

```
+ Found 1 ['636BEF93F99449114785304641f419d450ce24ddfa03f4383e7593e6'] matching passwords
```

```
+ Found 1 ['A66772BEAF7A47361f6929611cc24b92b86cb84403c7773996ac49bc'] matching passwords
```

```
+ Found 1 ['8C8066C40C224A6700c50395afa1d3a87c9b76a1215193a29226e170'] matching passwords
```

```
+ Found 1 ['AD10E9DF1D23435163457052e8433cc60aa4a853ee13143db90b0456'] matching passwords
```