



# Securing Password Storage

## Increasing Resistance to Brute Force Attacks

-jOHN (Steven)  
Internal CTO

 @m1splacedsoul

Chandu Ketkar  
Technical Manager  
 @cketkar

Scott Matsumoto  
Principal Consultant  
 @smatsumoto

**\*\*\*Comments from  
presentation  
discussion in boxes  
like this  
throughout\*\*\***

# History /etc/password

```
etc/password
```

```
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

```
0000fac2ec84586f9f5221a05c0e9acc3d2e670
000022c7caab3ac515777b611af73afc3d2ee50
deb46f052152cfed79e3b96f51e52b82c3d2ee8e
0000dc7cc04ea056cc8162a4cbd65aec3d2f0eb
0000a2c4f4b579fc778e4910518a48ec3d2f111
b3344eaec4585720ca23b338e58449e4c3d2f628
674db9e37ace89b77401fa2bfe456144c3d2f708
37b5b1edf4f84a85d79d04d75fd8f8a1c3d2fbde
0000e56fae33ab04c81e727bf24bedbc3d2fc5a
000058918701830b2cca174758f7af4c3d30432
00002e09ee4e5a8fcdae7e3082c9d8ec3d304a5
d178cbe8d2a38a1575d3feed73d3f033c3d304d8
0000273b52ee943ab763d2bb3d83f5dc3d30904
```

```
SHA1('password') = 1e4c9b93f3f0682250b6cf8331b7ee68fd8
```

What do you see here?

How do we know what it is?

How could we figure this out?

**In the news**

LinkedIn

IEEE

Yahoo

...

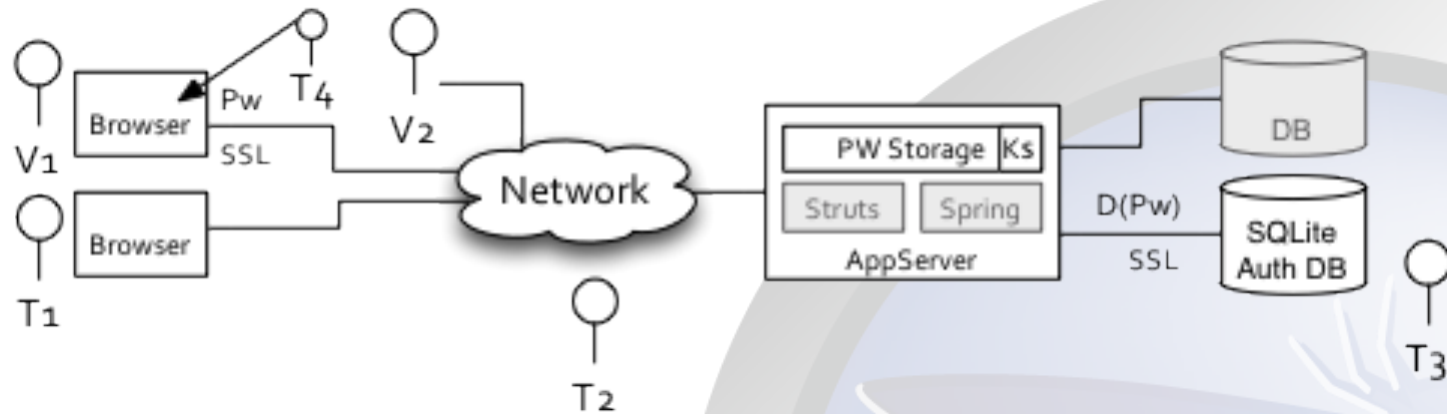


## Golden Rules

- #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



TM => Requirements = Threats your going to address



# 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/ 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


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



# The Threat's Tool box

Reverse it...

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



Thwarting these attacks is the focus of this presentation





- 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?

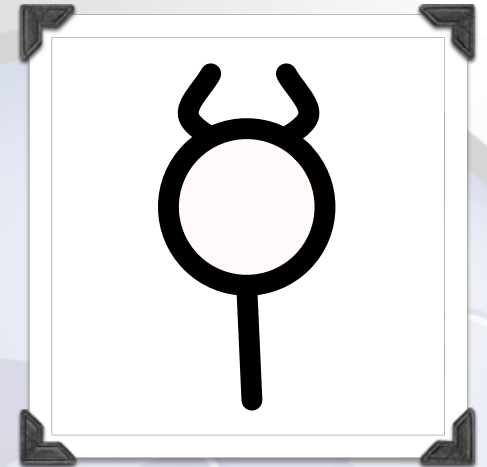


# Can We Successfully Attack a Hash?

Depends on the threat-actor...

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

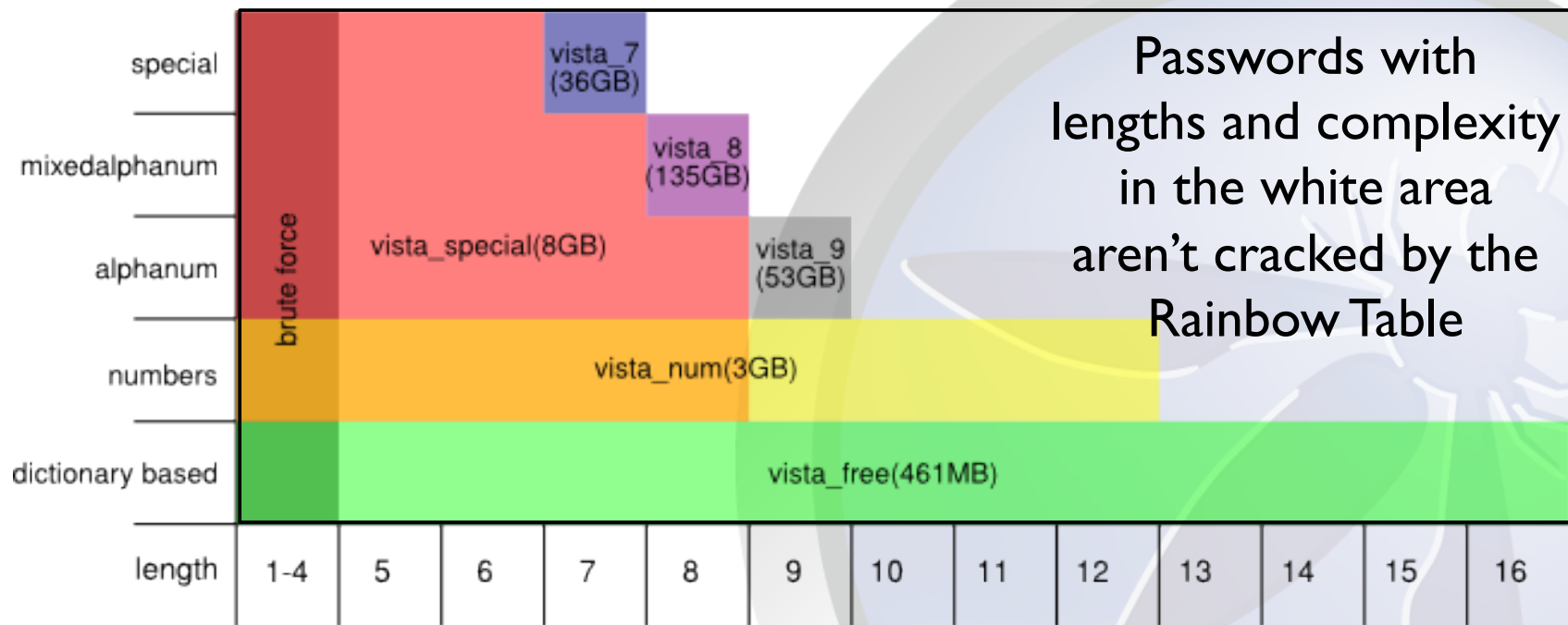
Is the algorithm supported by  
*your script-kiddie tool?*



# 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) <sup>4</sup>	338 MB	8.0 GB
(a-z   A-Z   0-9) <sup>5</sup>	21 GB	8.0 GB
(a-z   A-Z   0-9) <sup>6</sup>	1.3 TB	8.0 GB
(a-z   A-Z   0-9) <sup>7</sup>	87 TB	8.0 GB
(a-z   A-Z   0-9) <sup>8</sup>	5,560 TB	134.6GB
(a-z   A-Z   0-9) <sup>9</sup>	357,000 TB	No table
(a-z   A-Z   0-9) <sup>10</sup>	22,900,149 TB	No table

# Rainbow Tables: Fast but Inherent Limitations



Source: ophcrack

Tables are crafted for specific complexity and length

# 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

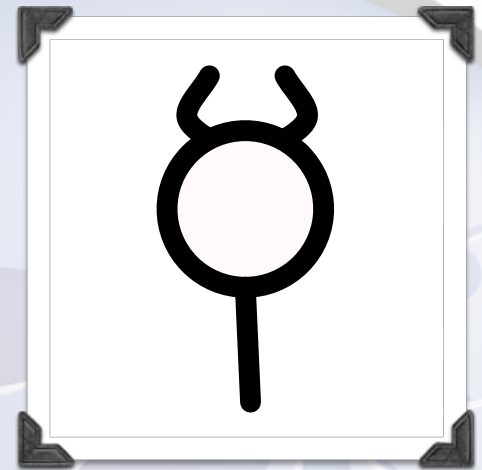


# Can salted hashes be Attacked?

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



We need a “Well-equipped Attacker” to build one table per users – right?



# Per User Table Building

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

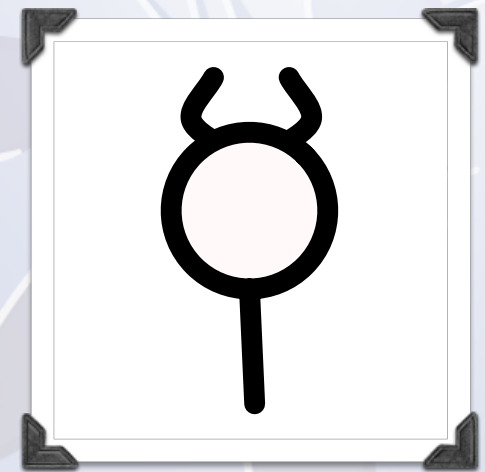
		8 Characters	9 Characters
Attacking a single hash (32 M/sec)	NVS 4200M GPU (Dell Laptop)	80 days	13 years
Attacking a single hash (85 M/sec)	\$169 Nvidia GTS 250	30 days	5 years
Attacking a single hash (2.3 B/sec)	\$325 ATI Radeon HD 5970	1 day	68 days

# We Can Attack a Salted Hash?

A salted-(SHA)hash can be broken with a modest investment in hardware

Our threat actor didn't need to be as well-equipped as we thought

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





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=);
```

```
pbkdf2(salt, pw, c){  
  hmac="hmac-sha-1"  
  key=pw  
  d=salt  
  
  for (int i=0, i < c, i++){  
    d = hmac(key, d)  
  }  
  
  return d  
}
```

Loop  $c$  times over a  
HMAC-SHA-1\*

HMAC: key is the  
password; text is the  
salt

How many 0s are needed  
for  $c$

- NIST: 1000
- iOS4: 10000

\*\*\* Pseudo-code ignores some detail for clarity's sake

As stated, this is sufficiently  
simplified as to be misleading. See  
[http://tools.ietf.org/html/  
rfc2898#section-5.2](http://tools.ietf.org/html/rfc2898#section-5.2) for detail.

# bcrypt

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

```
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  
}
```

$2^{\text{cost}}$  iterations slows down each  
hash operation

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

Resists GPU parallelization, but  
not FPGA

pw\_hash contains the salt

# scrypt

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

```
scrypt(salt, pw, N, p, dkLen){  
  b[p]  
  for (int i=0, i < p, i++){  
    b[i] = PBKDF2(pw, salt, 1, p·MFLen)  
  }  
  for (int I = 0, i < p, i++){  
    b[i] = MF(b[i], N)  
  }  
  dk = PBKDF2(pw, b, 1, dkLen)  
}  
**MF involves (HMAC-SHA-256, r)
```

Designed to defeat  
FPGA attacks

Configurable

- N =
  - memory footprint
  - CPU time
- P = defense against parallelism

**\*\*\*Follow-on discussions revealed that positives of this approach were not communicated. Update with explicit “benefits/limitations” material.**

# Defender VS Attacker

Defender	onus	Attacker
CPU on App Server	>	Custom hardware GPU or FPGA
4-16 processors / server	>	160+ GPUs / card, FPGA configurable
2-64 Application Servers	>	Scales with capabilities / crack value
20M Users,	>	May need only one (1) credential set
2M active / hr	>	Unlimited time
Knows scheme, key	<	Must discern scheme, steal key material

@tqbf points out this slide is misleading citing cost to verify vs. brute force the db. Both verify, so that's a not the issue. Point taken however: this slide **is** qualitative and misleading. Next version of this presentation must replace this slide with a quantitative effort comparison of Apples vs. Apples (e.g. costOfLogin vs. costOfSingleReverse)

Cost for defender is greater than the cost for the attacker if  
Adaptive Hash is the only control.



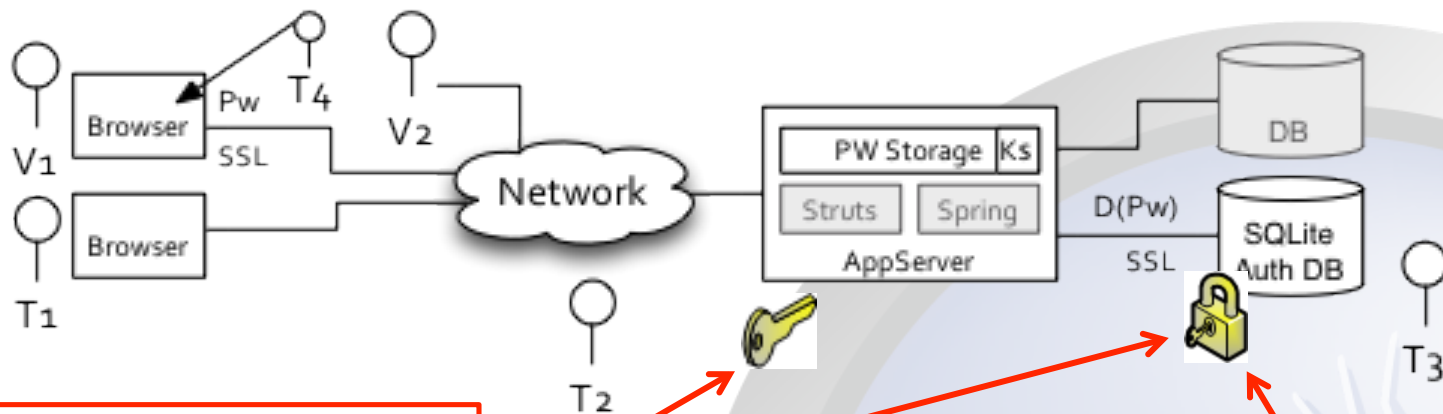
Adaptive Hashes At Best  
Strengthen a Single  
Control Point

We Can Do Better with  
Defense In Depth

Requiring a Key  
Gains Defense  
In Depth



# The Threat Model Revisited



@tqbf points out that if developers store keys in the DB, as they may be prone to do, compartmentalization falls apart.

Keyed transforms **do** differ from split digest texts because they resist attacks differently digests DB stolen (see slide #3 w/ leading 00000's)


Add a control that requires a key stored on the App Server

Threats can exfiltrate the password table, but now needs to also get the key



# hmac properties

```
salt || digest = hash(key, salt || plaintext);
```



extends hash (inherits hash properties)

Adds key

Resists padding / length extension attacks

# COMPAT/FIPS Solution

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

- HMAC := hmac-sha256
- key<sub>site</sub> := PSMKeyTool(SHA256()):32B;
- salt<sub>user</sub> := SHA1PRNG():32B | FIPS186-2():32B;
- pw<sub>user</sub> := <governed by password fitness>

## Optional:

• `<mixed construct> := <versionscheme>||<saltuser>||':'||<GUIDuser>||<pwuser>`  
• GUID<sub>user</sub> := **NOT** username or available to untrusted zones

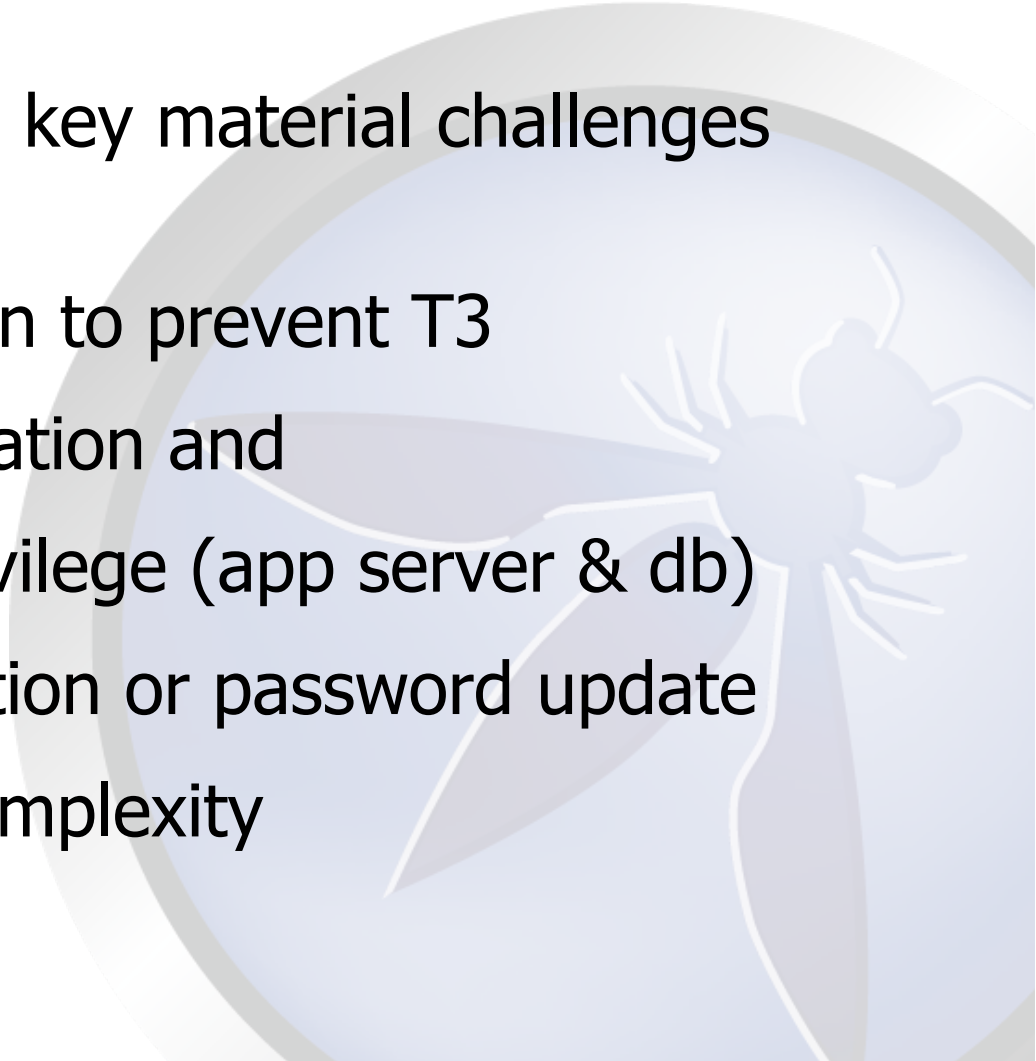
# hmac Solution Properties

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

Two people pointed out (I **wholly** agree) the timing attack vector does not apply to proposed approaches. We listed it to record closing an issue brought up by an external reviewer.



# hmac Limitations

1. Properly protecting key material challenges developers
  2. Must enforce design to prevent T3
    1. compartmentalization and
    2. separation of privilege (app server & db)
  3. No support of rotation or password update
  4. Versioning adds complexity
- 

# Reversible Solution

`<versionscheme>||<ciphertext> := ENC(<wrapper keysite>, <protected pw>)`

`<protected pw> := <versionscheme>||<saltuser>||<round 1>`

`<round 1> := ONEWAY(<keysite>, <mixed construct>)`

`<mixed construct> := <versionscheme>||<saltuser>||<pwuser>`

`ENC := AES-256`

`ONEWAY := hmac-sha-512 | scrypt`

`<keysite> := PSMKeyTool(<saltsite>, <pwsite>, <c>, DkL)`

`<wrapper keysite> := PSMKeyTool(<saltwrapper>, <pwwrapper>, <c>, DkL)`

`Versionscheme := integer (4B)`

`PSMKeyTool := PBKDF2(<saltsite>, c=1000000, DkL=32B):32B;`

`saltuser,site,wrapper := SHA1PRNG():32B;`

`pwuser := <governed by password fitness>`

Slide as presented. However, internal (unpublished) documentation reflects using an adaptive (rather than a keyed) scheme on <mixed construct> to produce <round 1> result. This slide needs wholesale re-work.



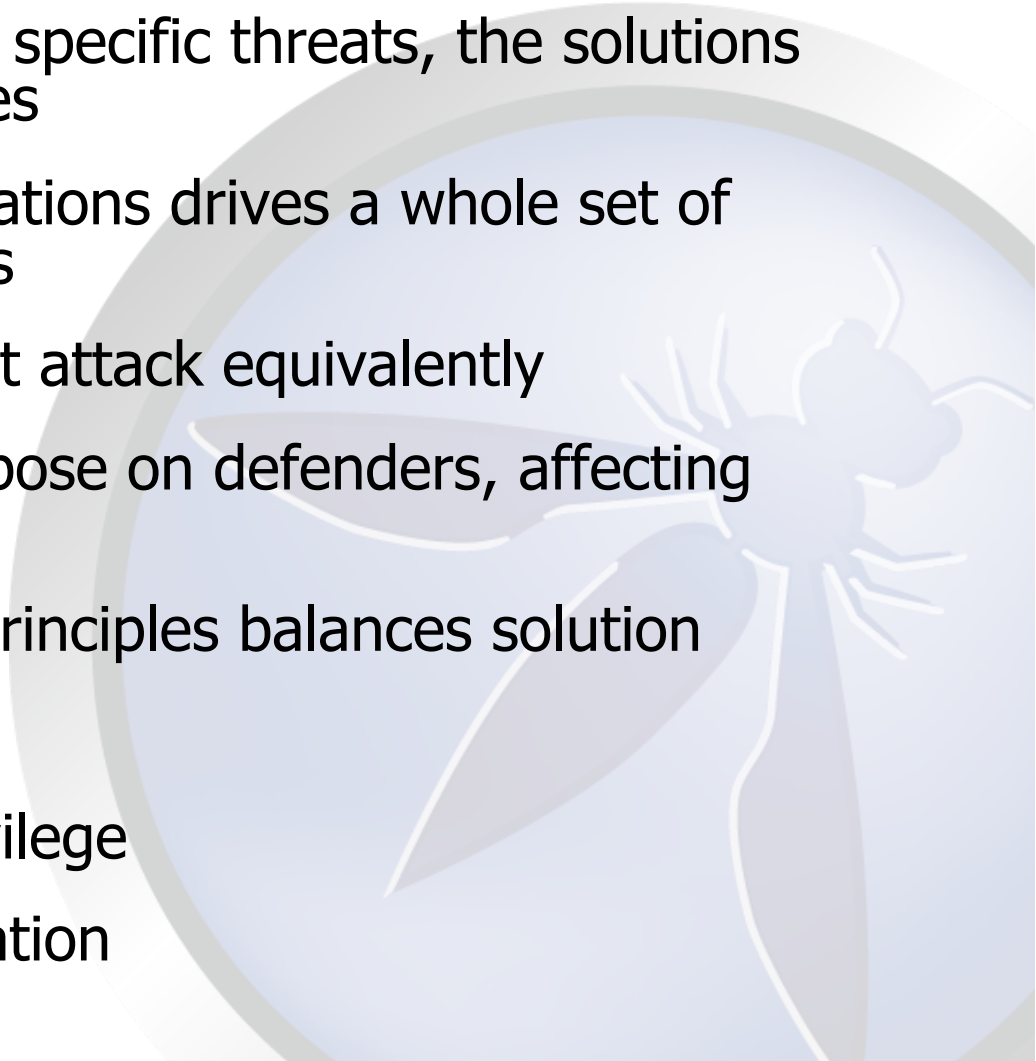
# Reversible Solution Properties

- Inherits “compat” solution properties
- Symmetric scheme supports
  - Versioning
  - Encryption policies (key rotation, etc.)
- Stolen PW DB useless
- Stolen PW DB + AES key still requires reversing one-way function

Versioning / rotation features (requirements) designed to address incident response and maintenance under attack. Without having explained this portion of the workflow/design, this looks unnecessary.



# 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
- 





Thank You for Your  
Time

Questions

# 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](#)

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 \(Draft\)](#)