Audience: OWASP crowd mainly; Developer secondly.

History /etc/password

etc/password

```
root:0:0:EC90xWpTKCo
hjackman:100:100:KMEzyulaQQ2
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

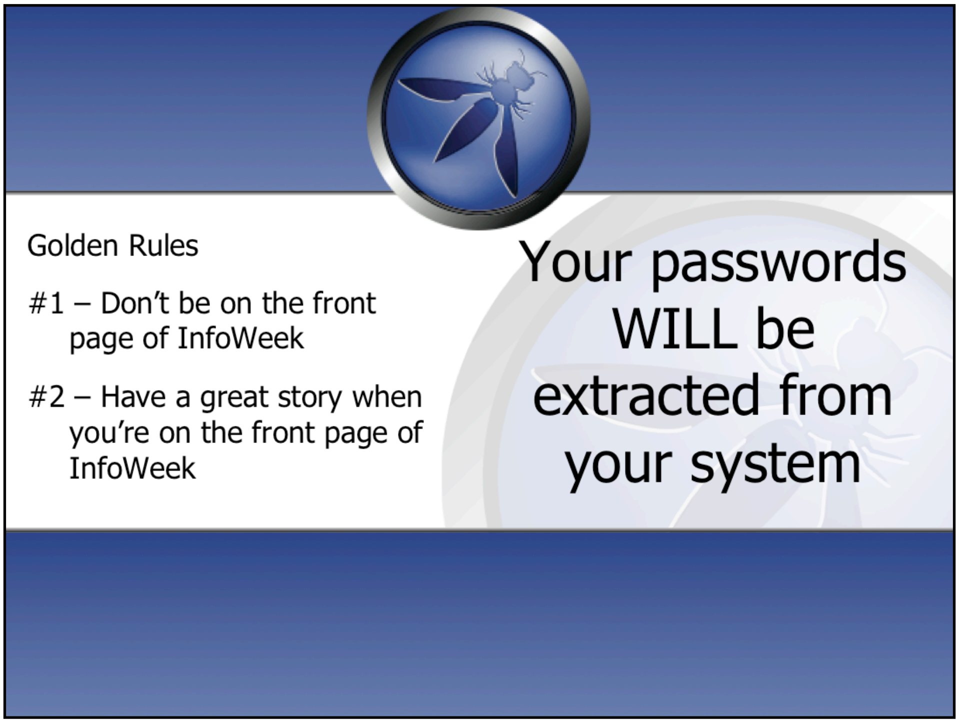
LinkedIn

IEEE

Yahoo

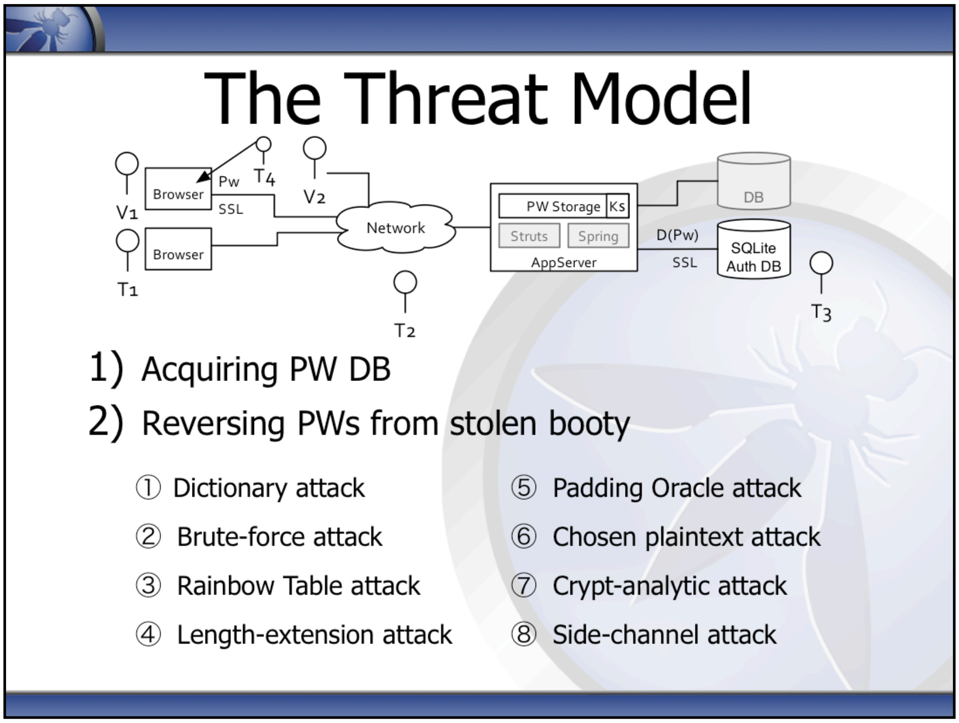
...

SHA1('password')= 1e4c9b93f3f0682250b6cf8331b7ee68fd8



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

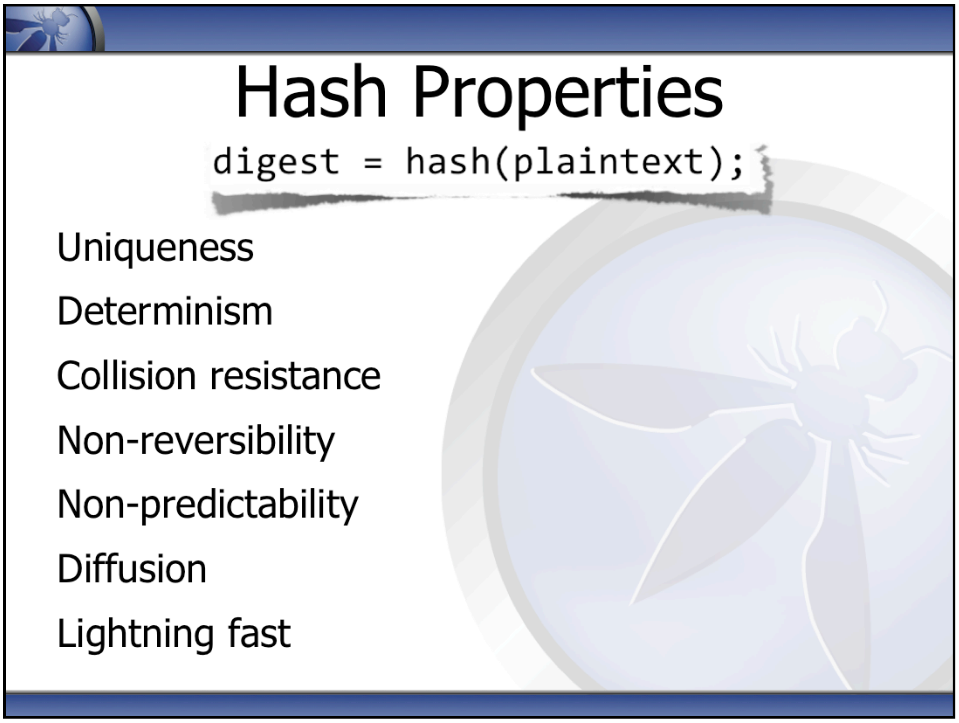
Engage the audience to see what they do. Gauge which people (companies) promote which solution.

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

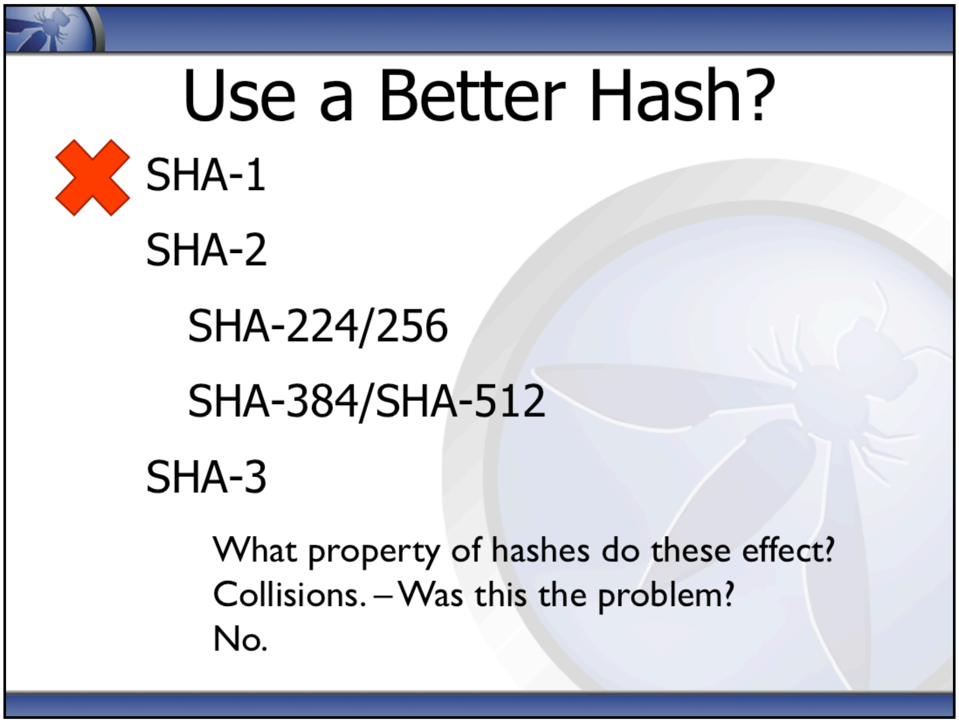
The meta-point about this section is that all of the solutions in this section focus on a single control-point.

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".



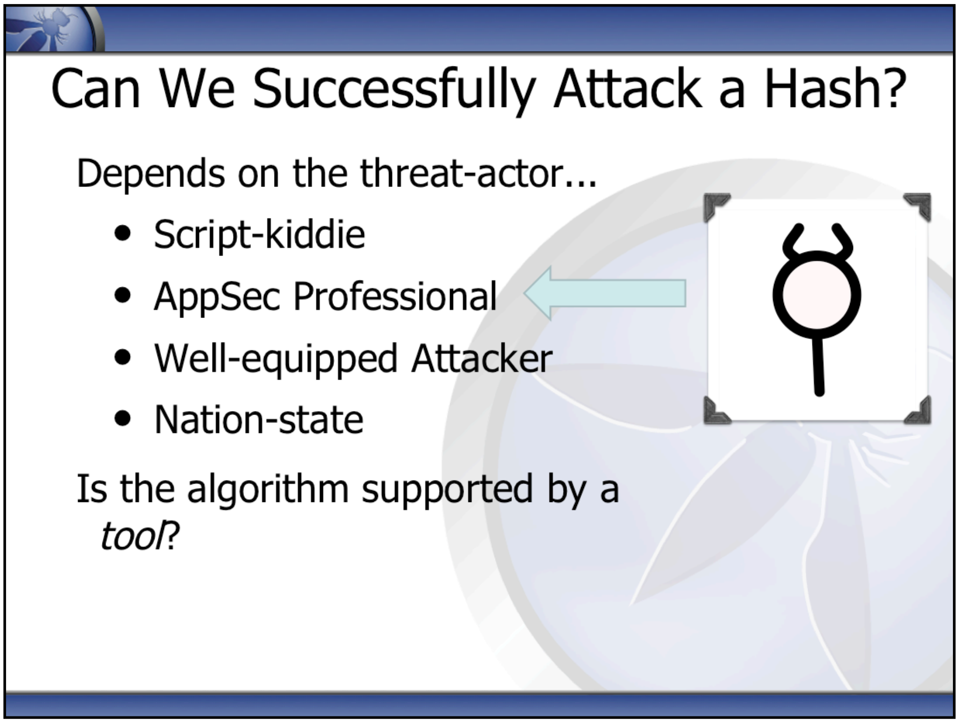
First to understand the Properties of the Crypto function. The properties will restrict/ influence your design.



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.

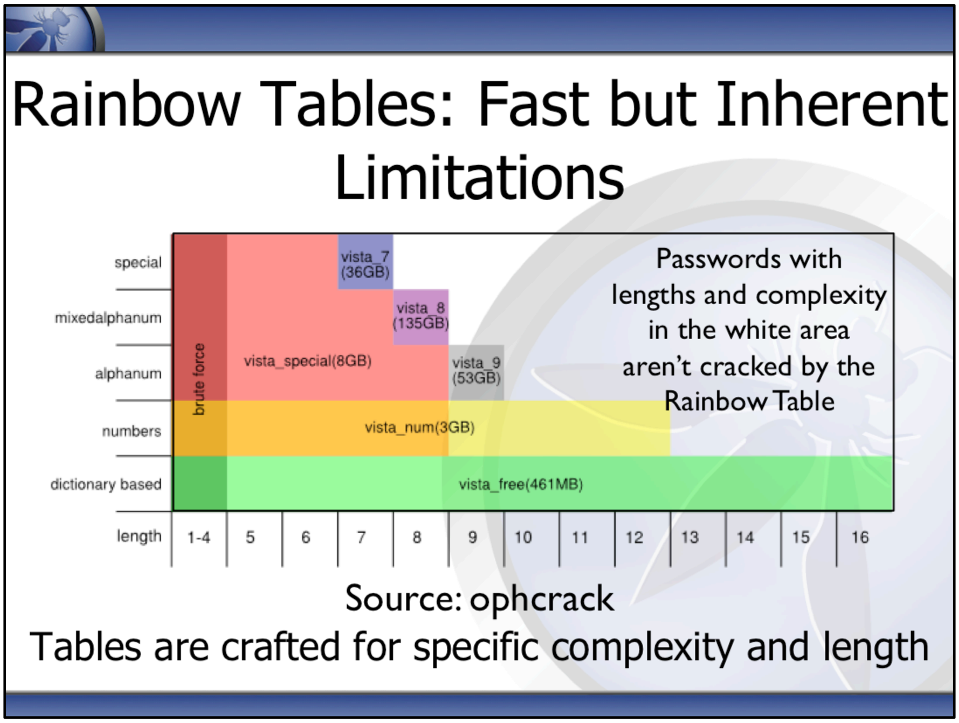
Background on Rainbow tables (from Chandu). Note that this is a just to get you started in understanding.

---- 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.



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?

Table Sizes

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

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 Does the Salt Do?

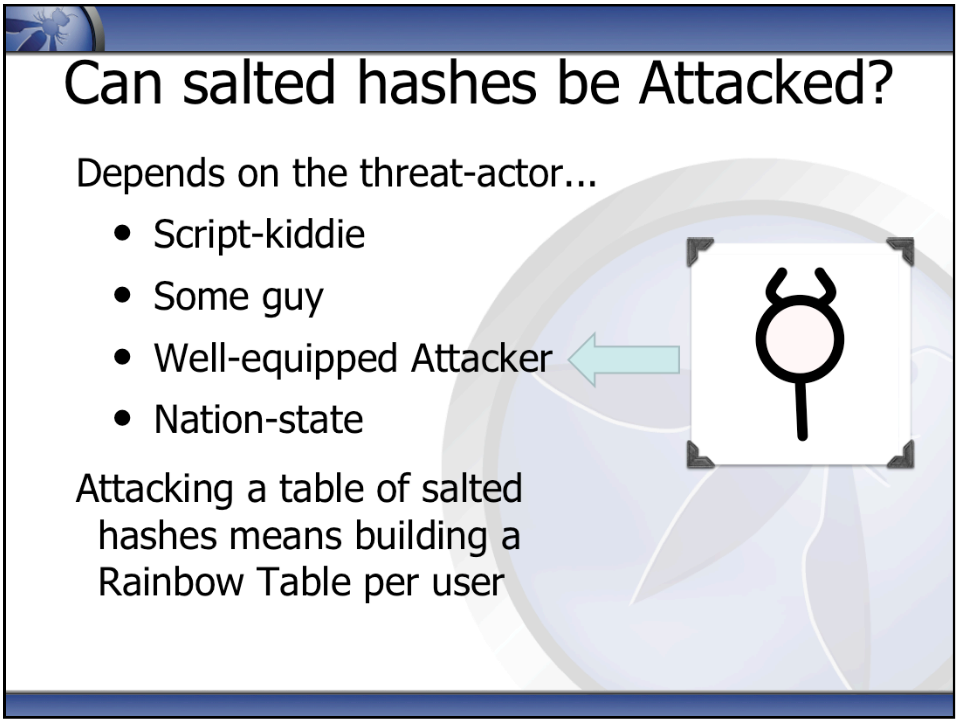
```
salt || digest = hash(salt || plaintext);
```

De-duplicates digest texts

Adds entropy to input space*

- increases brute force time
- requires a unique table per user

<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"?

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

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.

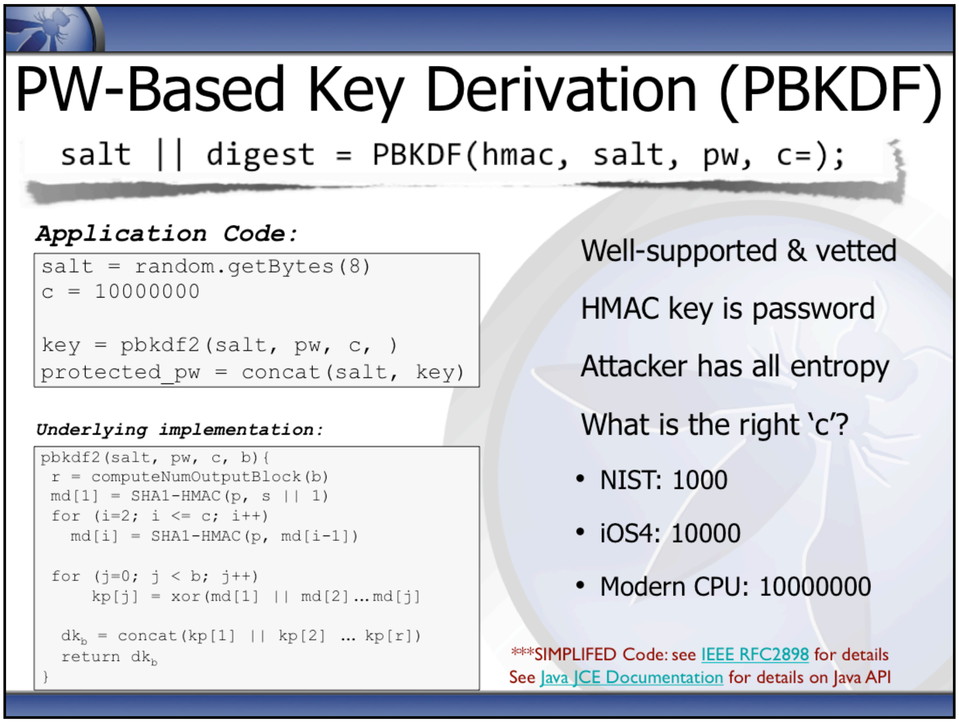
THe ATI Radeon HD is not at the top performer. It's an acceptable performing card. (http://www.videocardbenchmark.net/high_end_gpus.html)

---- 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)



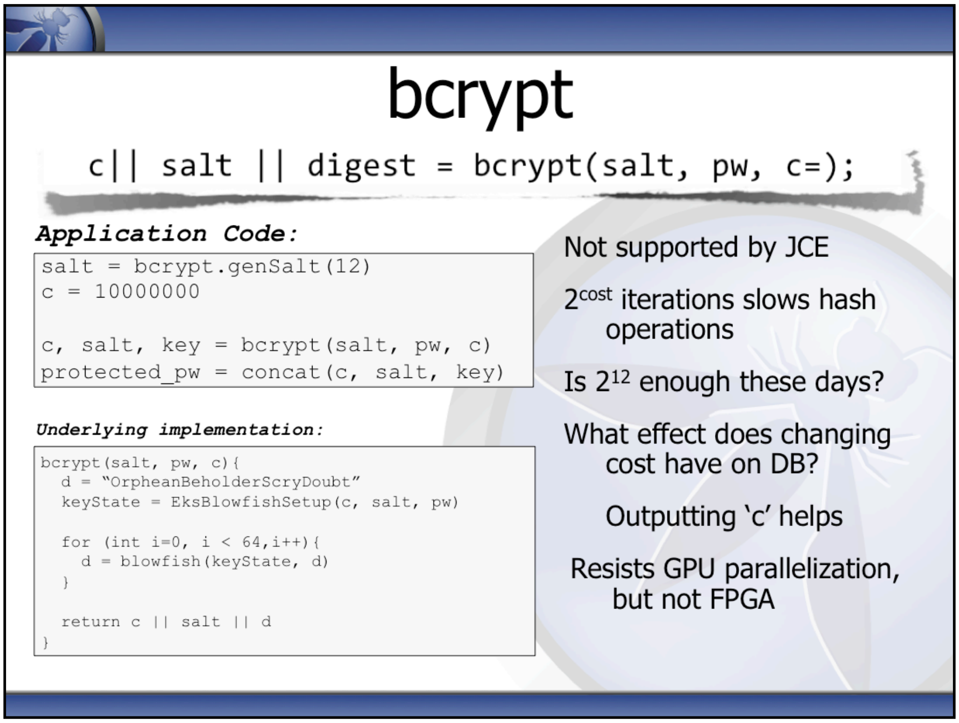
- PBKDF says that it can be any Pseudo Random Function (PRF), but the implementation of PBKDF2 only supports HMAC-SHA1
- Signature Parameters:

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



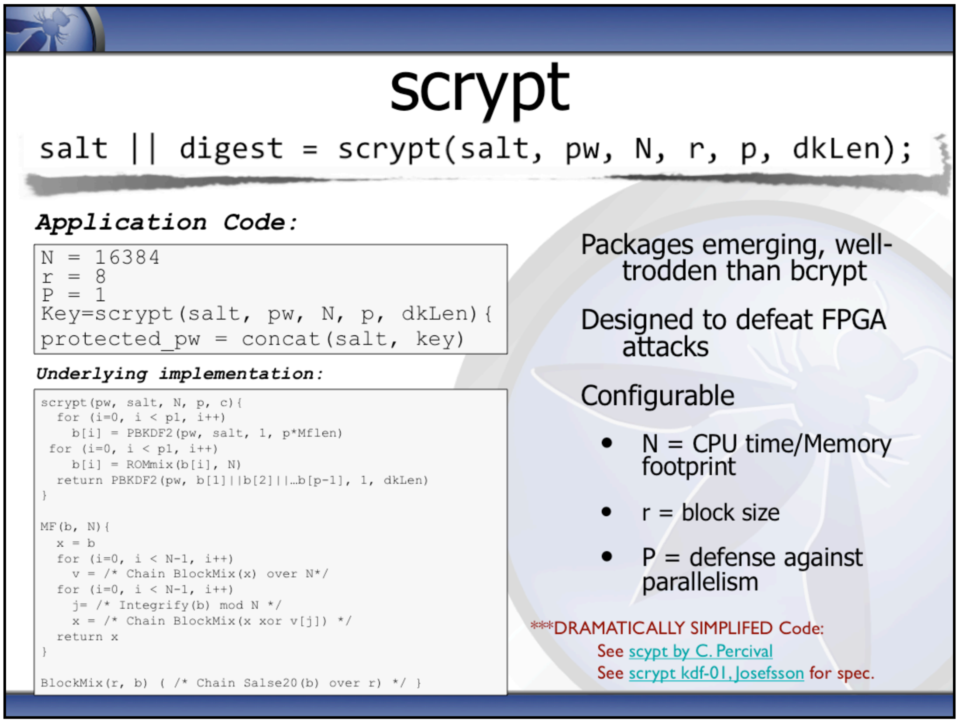
FPGA – Field Programmable Gate Array – In short programmable hardware.
2**12 = 4096

Parallelism is the key thing for the attacker.

BACKGROUND/REFERENCE:

1. Niels Provos and David Mazières, *The OpenBSD Project* http://static.usenix.org/events/usenix99/provos/provos.pdf
   1. A bit unfathomable, but this pseudo-code was enlightening. cost is used in the setup of Blowfish. It's a power of 2

```
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)
```



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:

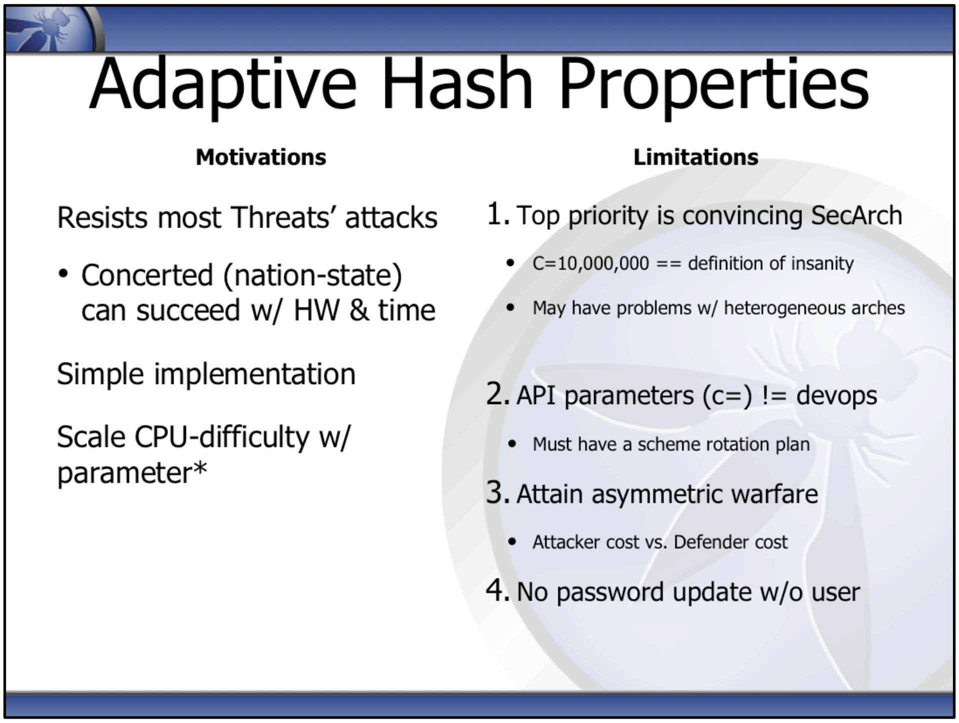
> The scrypt function is specifically designed to hinder such attempts by raising the resource demands of the algorithm. Specifically, the algorithm is designed to use a large amount of memory compared to other password-based KDFs, making the size and the cost of a hardware implementation much more expensive, and therefore limiting the amount of paralleling 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



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:

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

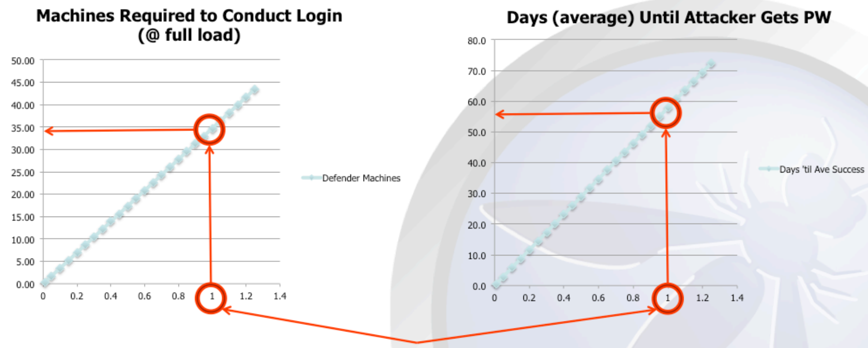
Population / 2 = average break = 10M

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

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

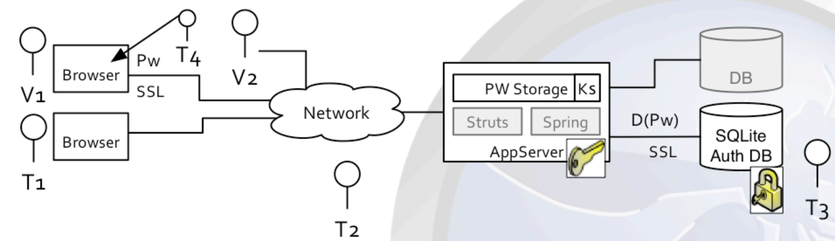
- Brute force out of reach
  - >= 2^{256} for SHA-2
- Requires 2 kinds of attacks
  - AppServer: RMIi Host keystore
  - DB: reporting, SQLi, backup

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

Just Split the Digest?

No. They're not the same.

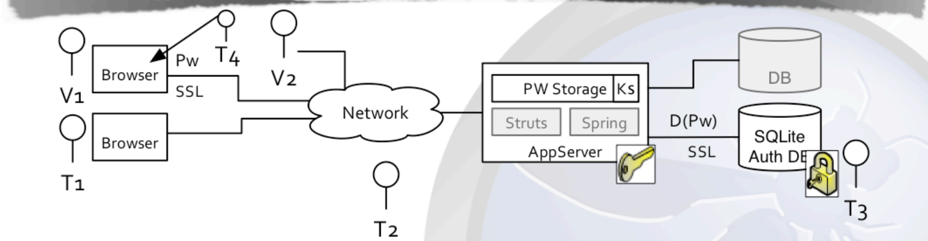
- Lacks key space (brute force expansion)
- Steal both pieces with the same technique
- Remember `000002e09ee4e5a8fcdae7e3082c9d8ec3d304a5` ?

```
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
```

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

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 ***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 ~= 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)
   - $\texttt{version||salt}_{\texttt{new}}\texttt{||protected = scheme}_{\texttt{new}}\texttt{(salt}_{\texttt{old}}\texttt{, digest}_{\texttt{existing}}\texttt{)}$ –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

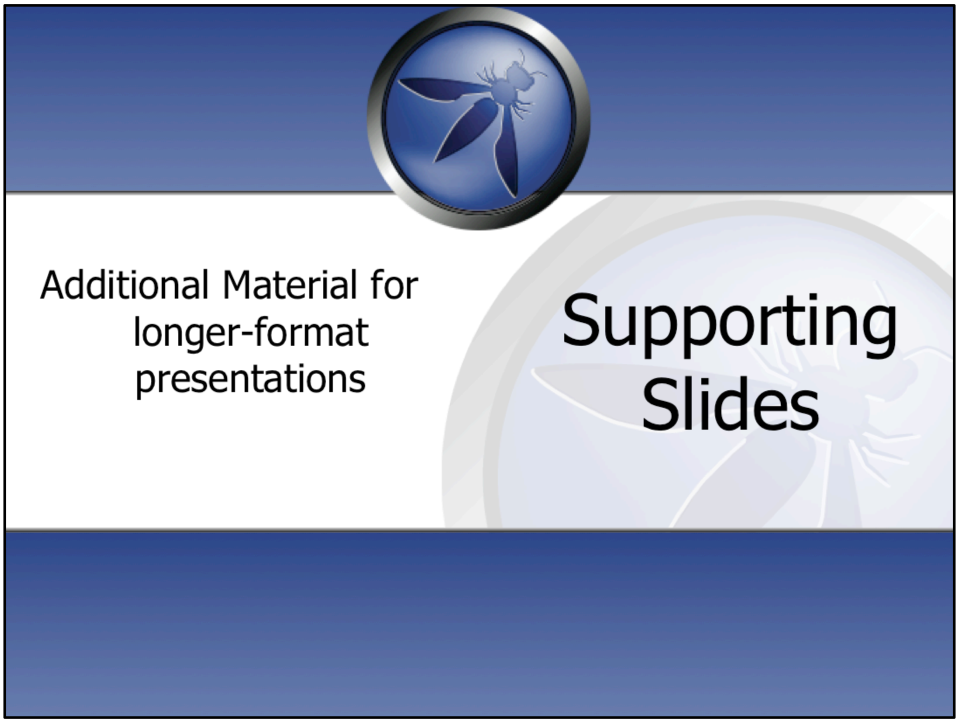
Conclusions

- 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

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

Stored Passwords Requirements

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

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

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

hmac Solution Properties

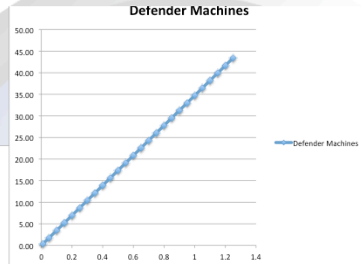
	Attack	Resistance
1.1	Resist chosen plain text attacks	**Yes**, Scheme complexity based on ($salt_{user}$ & pw_{user}) + key_{site}
1.2	Resist brute force attacks	**Yes**, $Key_{site} = 2^{256}$, $salt_{user} = 2^{256}$
1.3	Resist D.o.S. of entropy/randomness exhaustion	**Yes**, 32B on password generation 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)

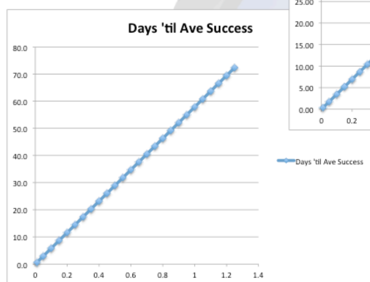
<describe the solution properties>
The problem is that it's difficult to rotate the key.

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 ['E1301662EC6349E5021c4cd8c158533aa9342ddee452f74f321ea0fa'] 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
```