RSACONFERENCE2014
FEBRUARY 24 – 28 | MOSCONE CENTER | SAN FRANCISCO

Share.
Learn.
Secure.
Capitalizing on
Collective Intelligence

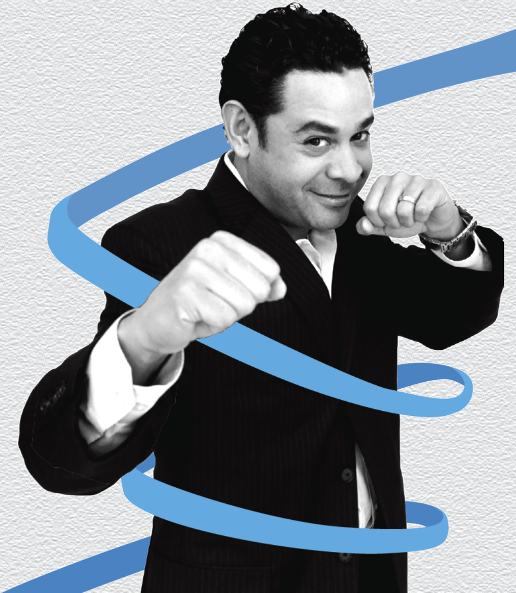
Honeywords:
A New Tool for Protection from Password Database Breach

SESSION ID: DSP-W02

Kevin Bowers
Senior Research Scientist
RSA Laboratories
kbowers@rsa.com

Ronald L. Rivest
Vannevar Bush Professor
MIT EECS
CSAIL
rivest@mit.edu

(some slides adapted from those of Ari Juels)



Outline

- Motivation – theft of password hash files
- *Honeywords* – enables detection of theft, prevents impersonation
    - Honeywords are ``decoy passwords'' (many for each user)
    - Separate ``honeychecker'' aids in password checking
- How to generate good honeywords?
- Experimental results (can you tell honeywords from real passwords?)
- Implementation guidance (Django)

Motivation: Theft of Password Hash Files

Good and bad news about password breaches

- The good news: when talking about password (or PII) breaches, a convenient recent example is always available!
  - October 2013: Adobe lost 130 million ECB-encrypted passwords

1.5 million passwords
June 2012

6+ million pas
June 2012

450,000 passwords
July 2012

50 million passwords
March 2013

- The bad news: This is all **bad news**.

Passwords usually stored in hashed form

- *P* = Alice’s password
- System stores mapping “Alice” ➔*h(P)* in database, for a suitable hash function *h*.
- When someone (perhaps Alice) tries to log in as Alice, system computes *h(P’)* of submitted password *P’* and compares it to *h(P)*. If equal, login is allowed.
- Hash function *h* should be easy to compute, hard to invert. Such ``one-wayness’’ makes a stolen hash not so useful to adversary.

Password hashing

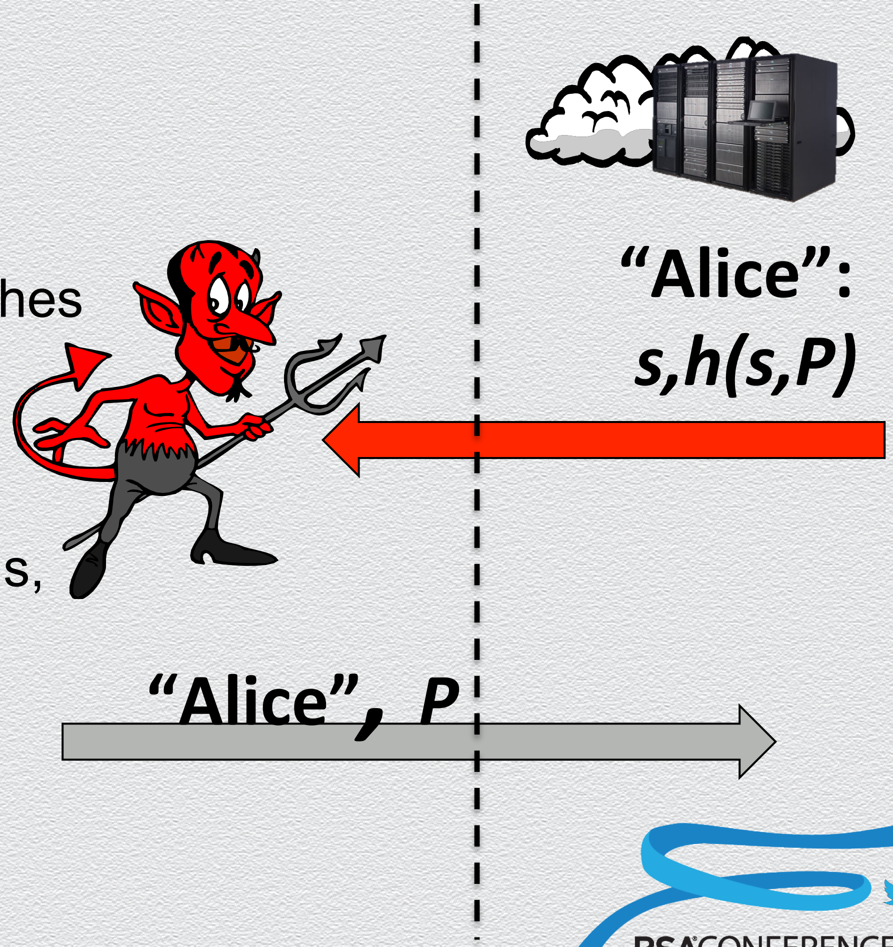
- To defeat precomputation attack, a per-user ``salt'' value *s* is used: system stores mapping "Alice"➔*(s,h(s,P))*. Hash *h(s,P')* computed for submitted password *P'* and compared.
- Hashing with salting forces adversary who steals hashes and salts to find passwords by brute-force offline search: adversary repeatedly guesses *P'* until a *P'* is found such that $h(s,P') = h(s,P)$
- Also, hashing can be hardened (slowed) in various ways (e.g. bcrypt)
- This all seems good, but…

Password hashing

- Real passwords are often *weak* and *easily guessed*.
  - Study of 69M Yahoo passwords [B12] shows that:
    - 1.08% of users had *same* password (is *your* password “123456” ?)
    - About half had strength no more than 22 bits (4M tries to break)
- Password-hash crackers now use models or sets of real passwords:
  - [WAdMG09] uses probabilistic context-free grammar
  - Crackers use, e.g., RockYou 2009 database of 32 million passwords
- We assume in this talk that hashes can be cracked and passwords are effectively stored in the clear.

Adversarial game

- Adversary compromises system ephemerally, steals password hashes
- Adversary cracks hash, finding *P*
- Impersonate user(s) and logs in.
- Adversary almost always succeeds, and is often undetected.



Honeywords are “Decoy Passwords”

Decoys

- Decoys, fake objects that look real, are a time-honored counterintelligence tool.
- In computer security, we have “honey objects”:
  - Honeypots [S02]
  - Honeytokens, honey accounts
  - Decoy documents [BHKS09] (many others by Keromytis, Stolfo, et al.)
- Honey objects seem undervalued.

``Honeywords” proposed 2013 by Juels & Rivest

- ACM CCS 2013

Honeywords: Making Password Cracking Detectable

Terminology

Alice :

P_1

P_2

...

P_i

...

P_n

Terminology

Alice :

P_1

P_2

…

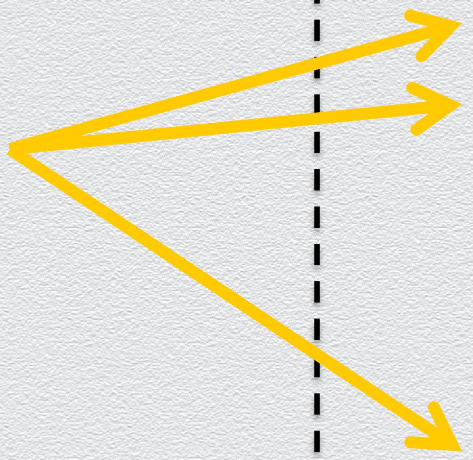
True password → $\mathbf{P_i = P}$

…

P_n

Terminology

Honeywords (decoys)



Alice :

P_1

P_2

…

$\mathbf{P_i = P}$

…

P_n

Terminology

Alice :

Sweetwords

P_1

P_2

…

$\mathbf{P_i}$

…

P_n

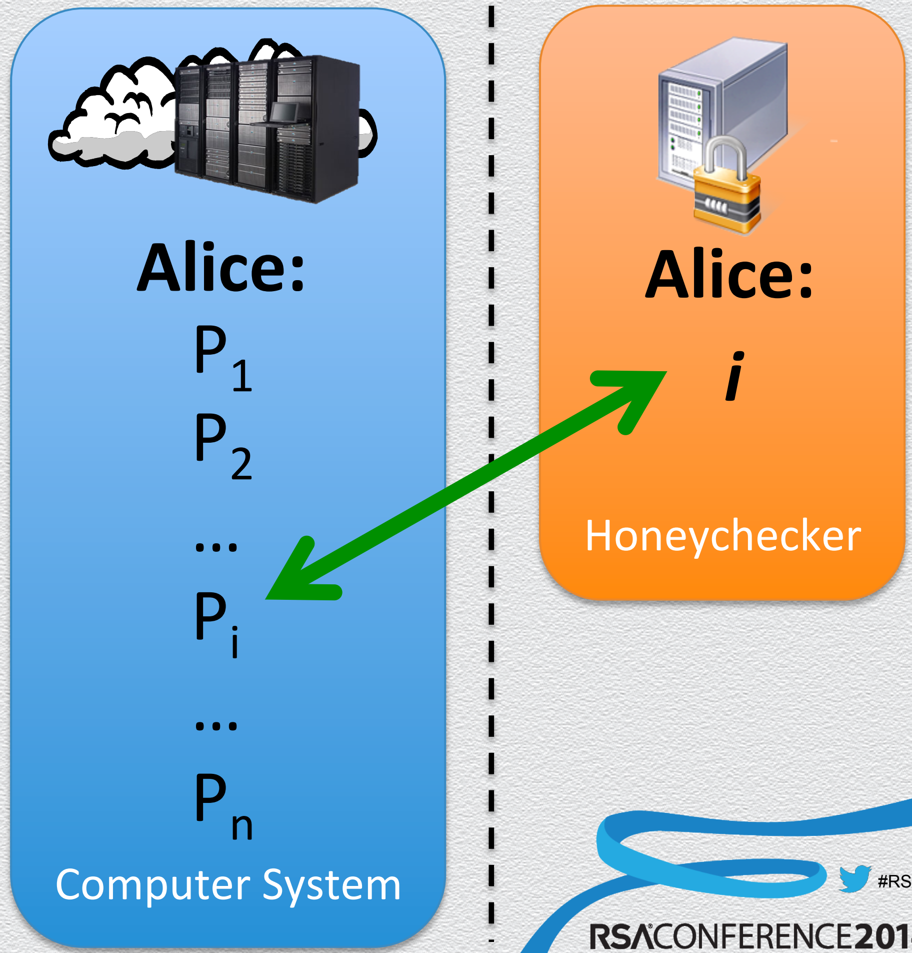
Honeyword design questions

- **Verification:** How does the check whether a *submitted* password *P'* is the *true* password P_i?
  - How is index *i* verified without storing *i* alongside passwords?
- **Generation:** How to generate honeywords?
  - How to make realistic decoy passwords?

(Many other design questions, e.g., how to respond when breach is detected…)

Honeywords: Verification

- The authentication system stores a mapping from Alice to her set of passwords
- A “honeychecker” stores the index of the correct password for Alice



Alice:

P_1

P_2

…

P_i

…

P_n

Computer System

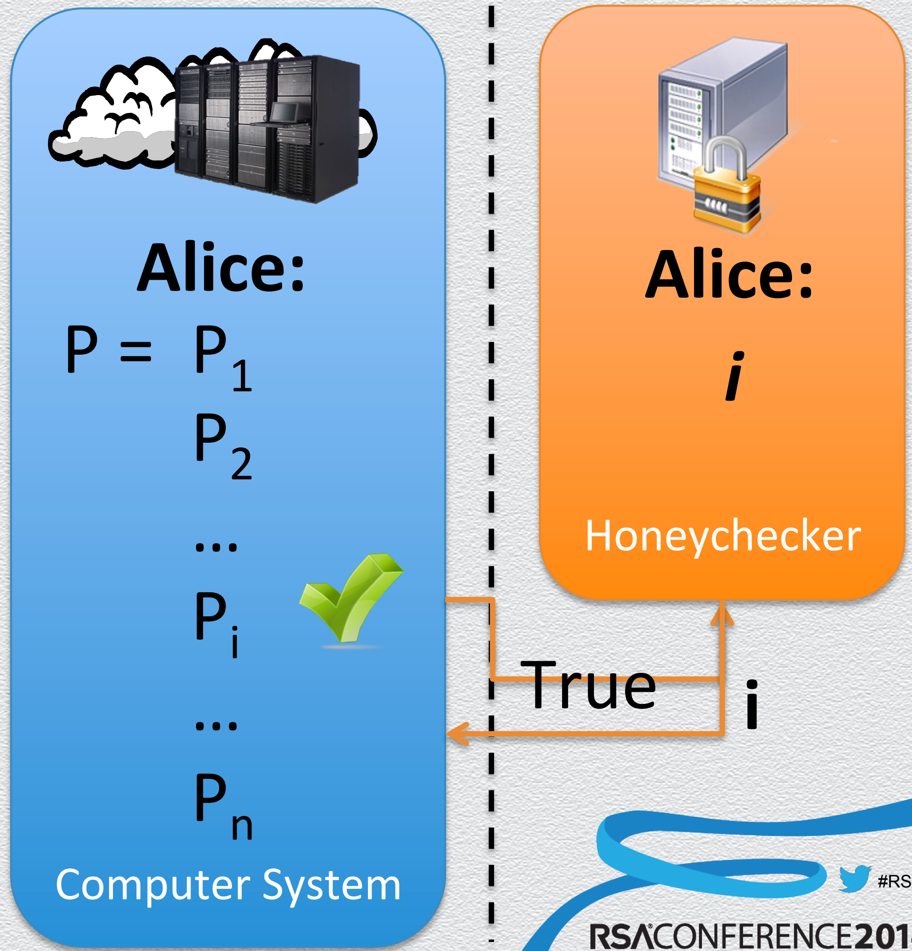
Alice:

i

Honeychecker

Honeywords: Verification

- Alice authenticates by submitting her password P
- The computer system checks her password against all those it stores
- If a match is found, the index of that match is sent to the honeychecker for verification
- If the index is correct, Alice is authenticated



The adversarial game

What is *i*?

“Alice”, P_j

With ideal honeywords, adversary guesses correctly ($j = i$), with probability only $1/n$

Alice:

P_1

P_2

…

P_i

…

P_n

Computer System

The adversarial game

Which is the (true) password?

Alice:

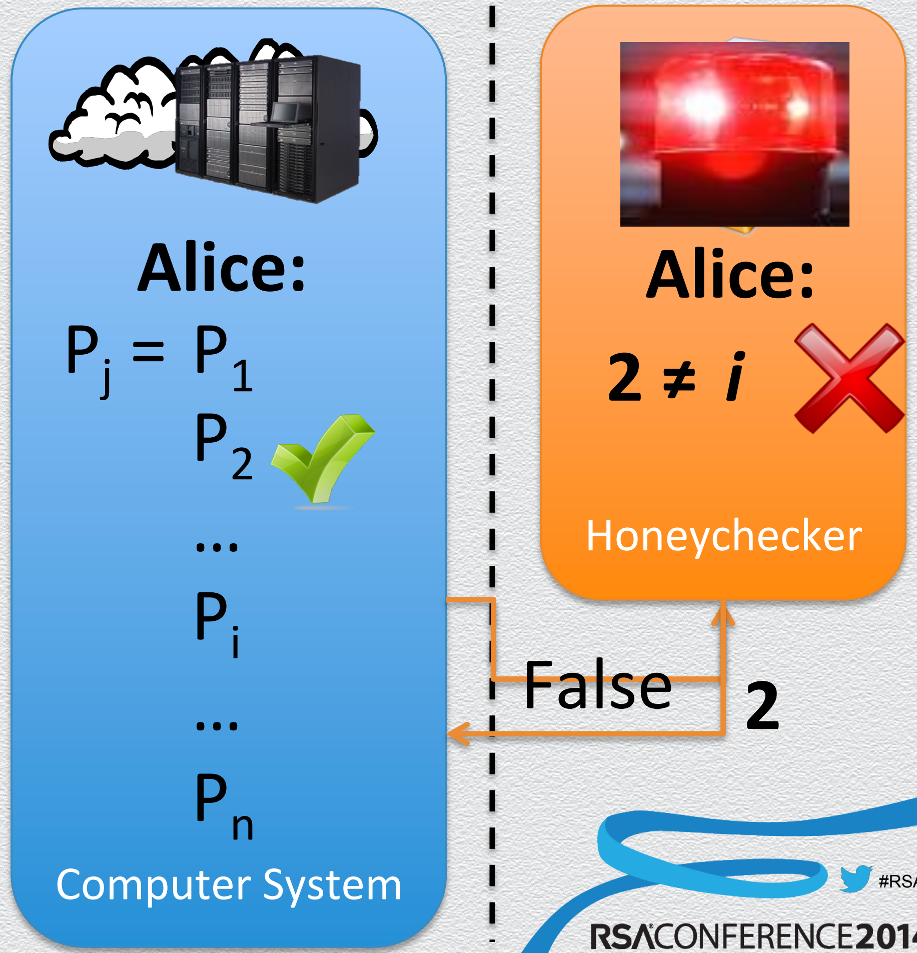
5512lockerno.
tribal_3
cshcsh.meowr.18
28/07/89rm
anto_2001_jesu
CRFRALAASS$4
!v0nn3

Computer System

Honeywords: Verification

- An attacker will submit a sweetword
- The computer system checks the password against all those it stores
- If a match is found, the index of that match is sent to the honeychecker for verification
- If the index is incorrect, an alarm is raised

P_j

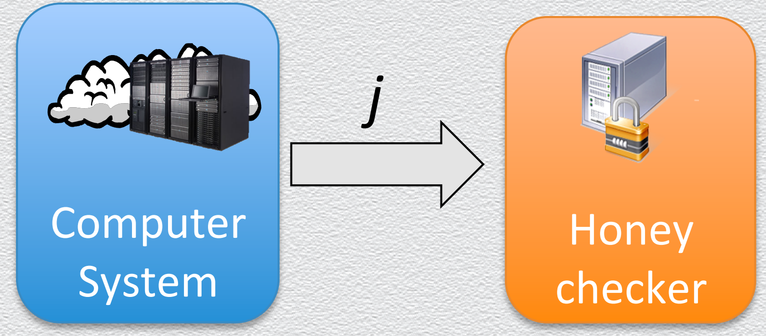


Honeywords: Verification Rule

- If *true password* P_i submitted, user authentication succeeds.
- Submitted password P' not in $P_1 \ldots P_n$ is handled as typical password authentication failure.
- If *honeyword* P_j is submitted, an alarm is raised by the honeychecker.
  - This is strong indication of theft of password hash file!
  - Honeywords (if properly chosen) will rarely be submitted otherwise.
- **No change in the user experience!**

Some nice features of this design

- System just transmits sweetword index *j* to honeychecker
  - *Little modification needed*
- We get benefits of distributed security
  - Compromise of either component isn't fatal
  - No single point of compromise
  - Compromise of both is just hashed case
- Honeychecker can be minimalist, (nearly) *input-only*
  - Only (rare) output is alarm



Another nice feature – offline operation

- Honeychecker can be *offline*
  - E.g., honeychecker sits downstream in security operations center (SOC)
  - Not active in authentication itself, but gives rapid alert in case of breach
  - If honeychecker goes down, users can still authenticate (using usual password); we really just lose breach detection (detection of password file theft).

How to generate good honeywords ?

Honeyword generation

Which is Alice’s real password?

Alice:
• QrMdmkQt
• AP9LXEEa
• m7xnQVV4
• [✓ kingeloi]
• y5BJKWhA

Honeyword generation:
Chaffing with a password model

- Password-hash crackers learn model from lexicon of breached passwords (e.g., RockYou database)
  - Make guesses from model probability distribution
- Simple (splicing) generator in our paper yields…

Alice:

- qivole
- [✓ paloma]
- 123asdf
- Compaq
- asdfway

But there are problem cases…

Which is Alice's real password?

Alice:
• hi4allaspls
• #1spongebobsmymansodonttouchhim
• Travis46
• #1bruinn
• KJGS^!*ss

Honeyword generation: Chaffing by tweaking

- [ZMR10] observed users tweak passwords during reset (e.g., HardPassword1, HardPassword2, …)
  - Proposed tweak-based cracker
- Idea: ``Tweak" password to generate honeywords!
- E.g., tweak numbers in true password…

Alice:

- yamahapacificer32145678987654321
- yamahapacificer12345678987654321
- yamahapacificer12345678901234567
- yamahapacificer62145678987654322

Honeyword generation: A research challenge

- Blink-182 is a rock band
- Blink-182 is *semantically significant*
  - Tweaking would break it
  - Generation is unlikely to yield it
- Dealing with such passwords is a special challenge—like natural language processing
- Subject of an upcoming paper

Alice:
• Blink123
• Graph128
• Froggy%71
• Blink182
• Froggy!83

How good does honeyword generation have to be?

- Suppose user chooses password *P* with probability *U(P)*
- Suppose honeyword procedure generates *P* with probability *G(P)*
- Given sweetword list $P_1, \ldots, P_n,$ adversary's best strategy is to pick P_j maximizing $U(P_j) / G(P_j)$
- For example, given chaffing-with-a-password-model, a particularly dangerous password is

  #1spongebobsmymansodonttouchhim

  (*much* more likely to be picked by user than as a honeyword!)

How good does honeyword generation have to be?

- We imagine practical choice of, say, $n = 20$
- With perfect honeyword distribution $U \approx G$ and adversary picks a honeyword (and sets off alarm!) with probability 95%
- Perfect honeyword distribution isn't required: even if adversary can rule out all but two sweetwords, we still detect a breach *systematically* with high probability
  - E.g., 50% guessing success means prob. 2^{-m} of compromising m accounts without detection

How good does honeyword generation have to be?

- Generation strategies can be *hybridized* as a hedge against failure of one strategy, e.g.,

• qivole!	• qivole#
• 123asdf	• 111asdf
• PleaseDismantle TheGreenLine89	• PleaseDismantle TheGreenLine12
• Froggy%71	• Froggy!88

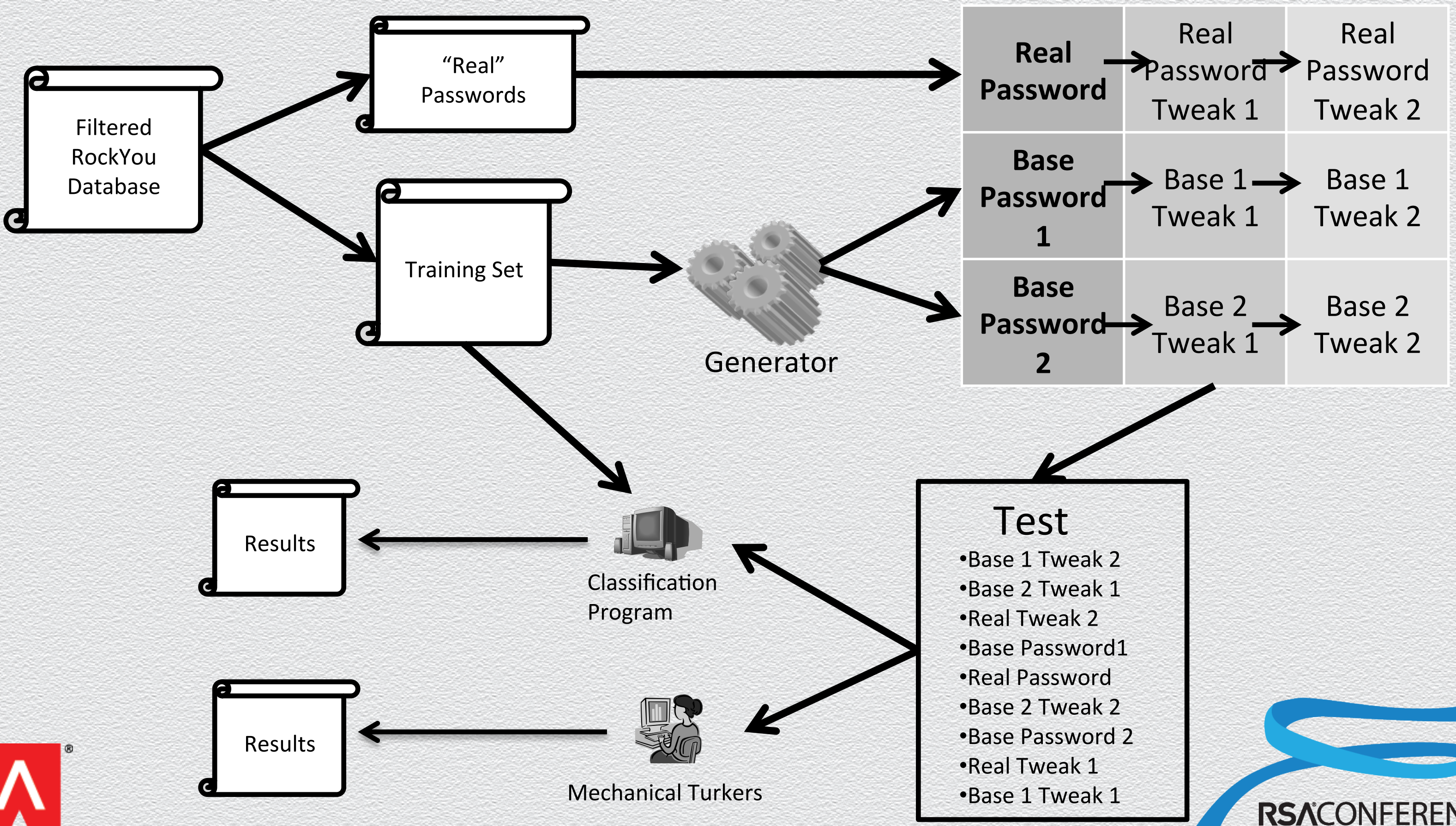
?

Experimental Results

Experimental Goals

- We attempt to measure how hard an attacker's task is to complete
  - Assume the password file is stolen and all hashes are reversed
  - Attacker must then determine the real password from a set of sweetwords
- Additional information about the user is not provided
- Test is performed both algorithmically (using a probabilistic model built from real passwords) and manually (leveraging Mechanical Turk)

Experimental Design



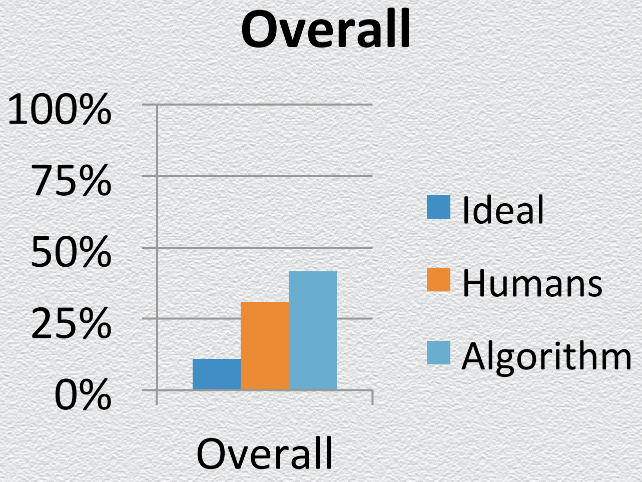
Real Password	Real Password Tweak 1	Real Password Tweak 2
Base Password 1	Base 1 Tweak 1	Base 1 Tweak 2
Base Password 2	Base 2 Tweak 1	Base 2 Tweak 2

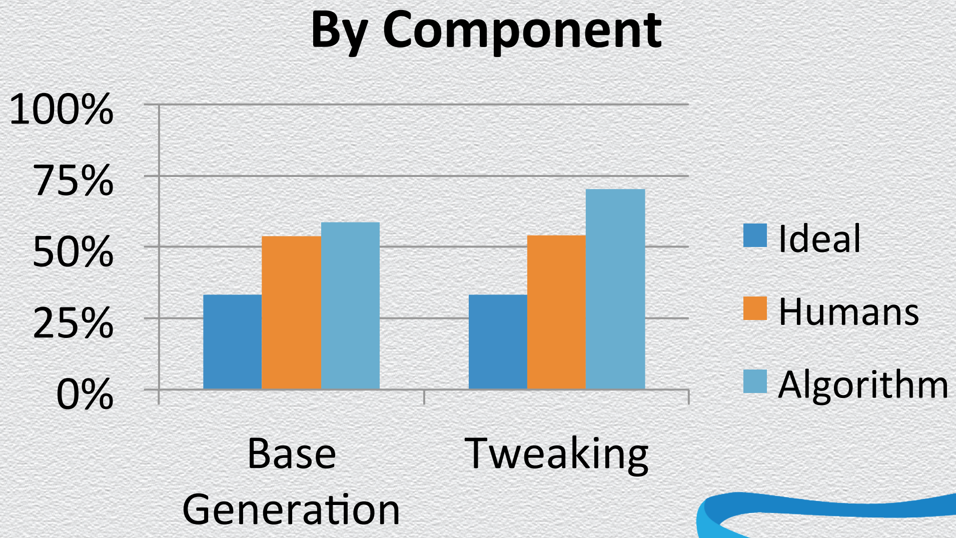
Test

- Base 1 Tweak 2
- Base 2 Tweak 1
- Real Tweak 2
- Base Password1
- Real Password
- Base 2 Tweak 2
- Base Password 2
- Real Tweak 1
- Base 1 Tweak 1

Results

- Even with only 9 choices, the attacker was unable to correctly guess the real password even just half of the time.





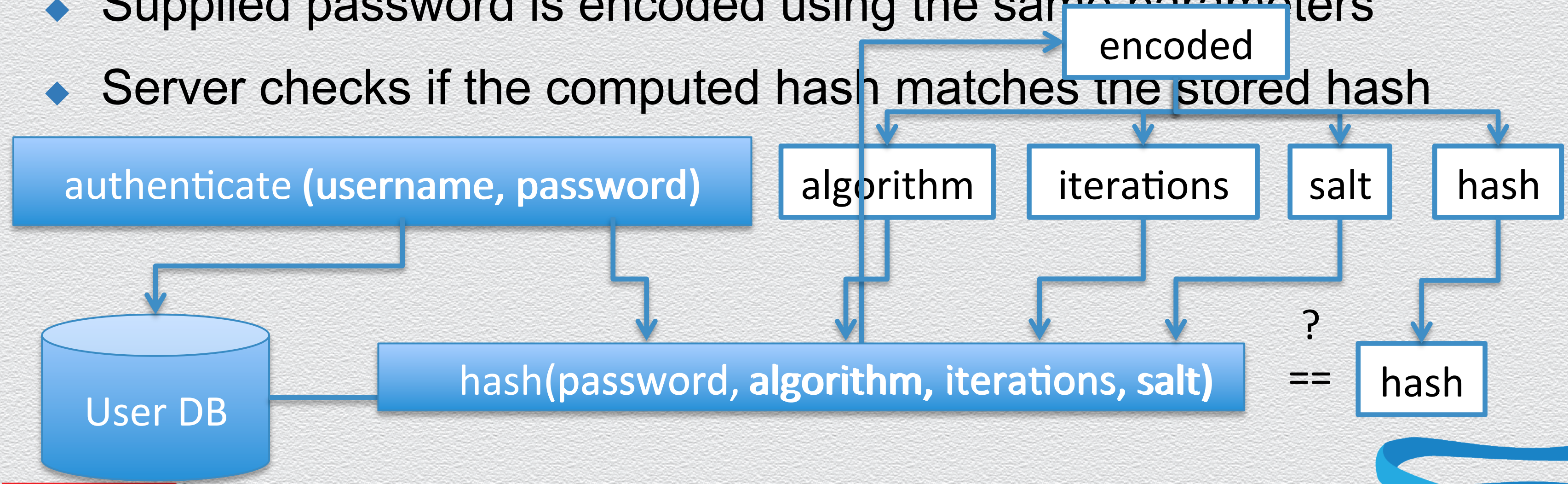
Implementation Guidance (Django)

Implementing Honeywords

- Goal: Walk through an implementation of honeywords, demonstrating components and pieces that are required for deployment
- High level presentation to identify major steps
- General principles should be easily translated to most frameworks
- Example implementation done in Django
  https://www.djangoproject.com/
- Code will be presented at the very end for those interested
  - Email for more information or access to the code.

Current Authentication

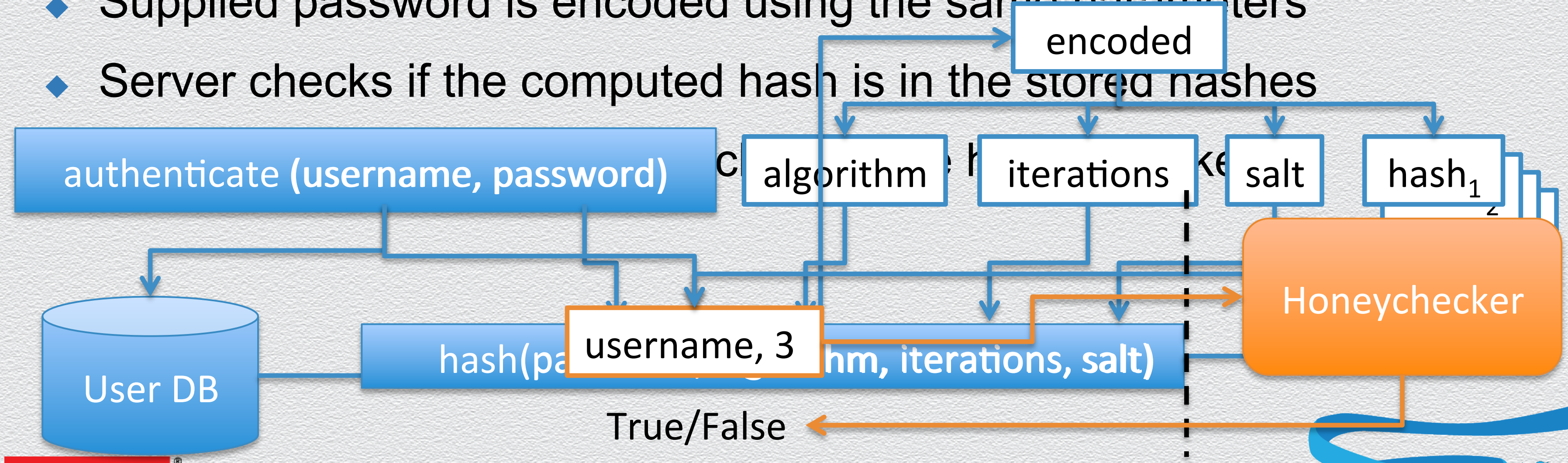
- Website calls authenticate(username, password)
- User's encoded hashed password is retrieved from the User DB
- Supplied password is encoded using the same parameters
- Server checks if the computed hash matches the stored hash



encoded
authenticate (username, password)
algorithm
iterations
salt
hash
User DB
hash(password, algorithm, iterations, salt)
?
==
hash

Desired Authentication

- Website calls authenticate(username, password)
- User’s encoded hashed passwords are retrieved from the User DB
- Supplied password is encoded using the same parameters
- Server checks if the computed hash is in the stored hashes



How do we get there?

- Modify the password verification function to implement new logic
- Enable communication with a remote system (honeychecker)
- Change what is stored as the user's password
- Build the honeychecker to store indices and verify them
- Modify the encoding function to generate honeywords and store their hashes, as well as notifying the honeychecker of the correct index

Changing the Verifier

Hashers

- Verification happens within a “hasher”
  - Implements both the verify and encode functions
- Different hashers implement different hashing algorithms
- System maintains an ordered list of hashers
  - At verification, they are tried in order
  - Password is re-encoded if it doesn’t use the first listed hasher
  - Placing a new hasher at the top of the list will upgrade users automatically as they log in

Hashers

PBKDF2PasswordHasher
HoneywordHasher
BCryptPasswordHasher
SHA1PasswordHasher
MD5PasswordHasher

Honeyword Hasher

- Needs a unique name (algorithm)
- Needs to communicate with the honeychecker
- Modify the implementation of
  - verify(password, encoded) – verifies that stored encoded password is an encoding of the submitted password
  - encode(password, salt, iterations) – given a password, salt and number of iterations computes the encoded password that will be stored in the database
- Additional functions that we will override
  - salt() – used to generate a salt value when the user changes or upgrades their password

Storing Sweetwords

Django Authentication

- Django maintains a database of users and their hashed passwords
- Usernames (max 30 characters) must be unique
- Password (max 128 characters) is actually a tuple describing the:
  - <algorithm>: Algorithm used to compute the hash
  - <iterations>: Number of times to apply the hashing algorithm
  - <salt>: A user-specific salt
  - <hash>: The Base64 encoding of the resulting hash value
- What django calls the encoded password is the concatenation of those strings separated by dollar

User DB

encode('passw0rd', 'pbkdf2_sha256', 12000, 'nR9uayYDhouC') =

'pbkdf2_sha256$12000$nR9uayYDhouC$yIVCfAB/UfLaEVAo0HSoPcSzwShmNYdmhRLB6pCu0yg='

- To avoid breaking things, we'd prefer not to replace the User model

Where can we store the sweetwords?

- Store the sweetwords in their own table, User DB stores a key into that table
- Need a key, known to the hasher, that can be used as an index into this table
  - Hasher knows algorithm, iterations and salt
  - Hasher can override the salt-generation function, giving even more control
  - Use the salt as the key
- Sweetwords database then stores a mapping from a salt to a number of sweetword hashes
- The salt should be changed every time the user changes password
- Ideally old sweetwords are deleted when they are no longer in use

Honeychecker

Honeychecker

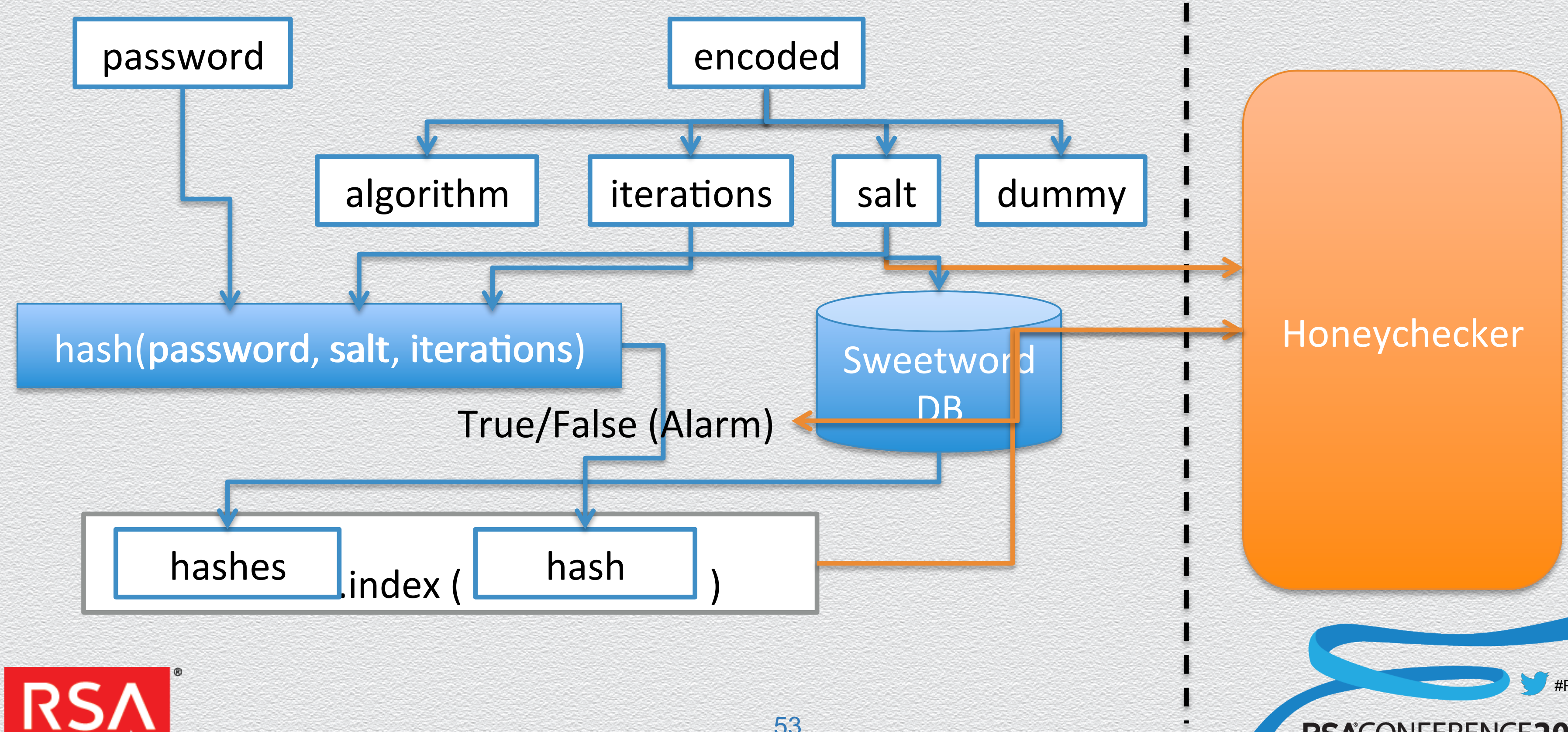
- Stores the index corresponding to a user
- Ideally runs on a separate machine or at least separate VM
- API supports updates (additions) and index checking
  - update_index(salt, index)
  - check_index(salt, index)
- Ideally old, unused salt/index pairs are removed from the honeychecker
- To further harden the system, these calls should only be allowed from known servers over trusted channels
- Probably want to backend the honeychecker by a database as well

Verification Function

Verify

- Coming back to the verify function in the HoneywordHasher…
- In the ideal model, the verify function checks if the hash of the submitted password is in the local database.
    - If not, the password was either mis-typed or an online guessing attack is occurring
    - If so, the index in the database is sent to the honeychecker for verification
        - If the index is correct, the user is authenticated
        - If the index is incorrect, it is likely that the database has been stolen and appropriate action should be taken.
- The parameters needed to hash the submitted password are stored in the database as well and must be extracted from the encoded password
- This is complicated a little in our case because we had to create a separate sweetword database

Verify(password, encoded)

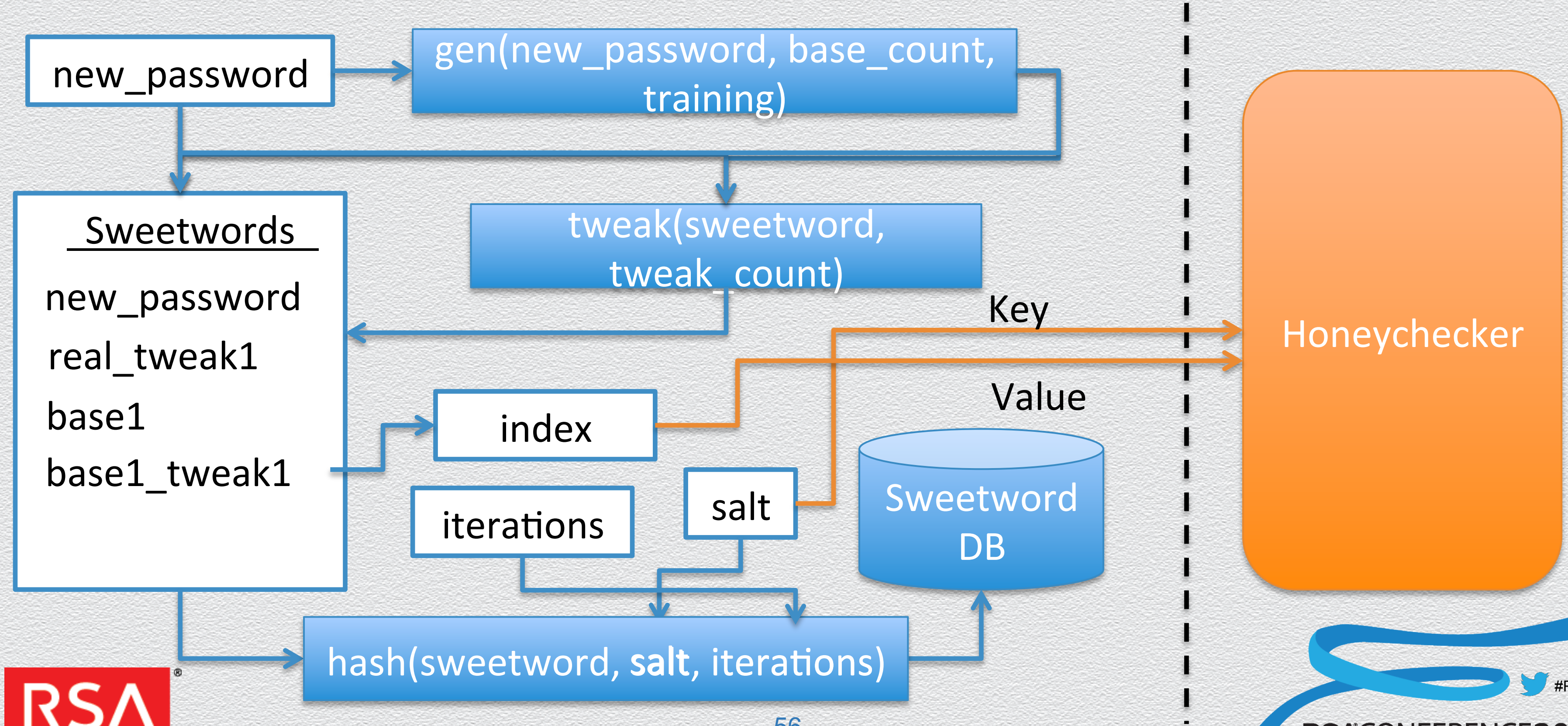


Encoding Function

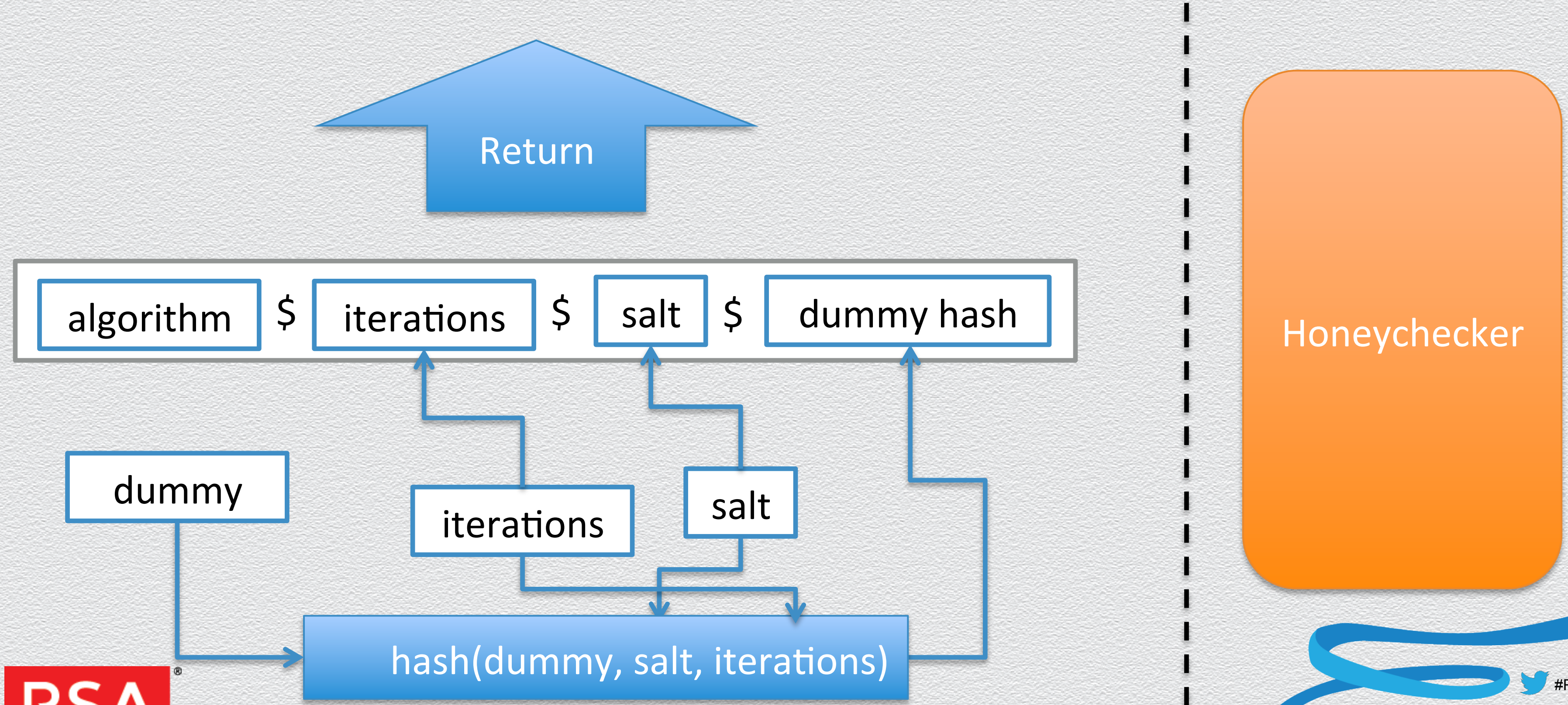
Encode

- The other half of implementing honeywords is creating them and storing them in the databases
- When a user submits a new password (or upgrades an old password) the encode function must:
  - Create the honeywords
  - Combine them with the real password to form the sweetword list
  - Randomly order that list
  - Store the hashes of all sweetwords in the Sweetword database
  - Inform the honeychecker of the new index associated with the user
  - Return something of the correct form to be stored in the User database

encode(new_password, salt, iterations)



encode(new_password, salt, iterations)



Helpers

- Base password generation
  - Download generation script from Ron’s webpage:
    - http://people.csail.mit.edu/rivest/honeywords/gen.py
  - Edit the file to ensure unique generation and inclusion of at least one digit (to allow tweaking)
- Tweaking
  - Tweak your base password as many times as you like (or can)
  - Need to ensure tweaks are unique
- Reordering
  - Base and tweaks are then randomly ordered
- Salt generation
  - Because salts are used as key, we need to ensure they are unique

Reviewing our checklist

- ☑ Modify the password verification function to implement new logic
- ☑ Enable communication with a remote system (honeychecker)
- ☑ Change what is stored as the user's password
- ☑ Build the honeychecker to store indices and verify them
- ☑ Modify the encoding function to generate honeywords and store their hashes, as well as notifying the honeychecker of the correct index

- The full code implementing everything on this list is included at the end of these slides.

Discussion and Conclusions

The larger landscape

- Honeywords are a kind of poor-man's distributed security system
- There are other, practical approaches to password-breach protection
  - Hashing (see Password Hashing Competition)
  - [Y82] (and many others), Dyadic Security
- Honeywords strike attractive balance between ease of deployment and security
  - Little modification to computer system
  - Honeychecker is minimalist
  - Conceptually simple

Code

HoneywordHasher

```
from django.contrib.auth.hashers import PBKDF2PasswordHasher
import xmlrpclib

# Define HoneywordHasher derived from PBKDF2PasswordHasher
class HoneywordHasher(PBKDF2PasswordHasher):
    # Give our hasher a unique algorithm name to later identify
    algorithm = “honeyword_base9_tweak3_pbdkf2_sha256”
    # Setup the honeychecker
    honeychecker = xmlrpclib.ServerProxy(<uri>)
```

HoneywordHasher.hash(self, password, salt, iterations)

```
# Compute pbkdf2 over password
hash = pbkdf2(password, salt, iterations, digest=self.digest)
# Base64 encode the result
return base64.b64encode(hash).decode(‘ascii’).strip()
```

HoneywordHasher.salt(self)

```
from django.utils.crypto import get_random_string

def salt(self)
    salt = get_random_string()  # Generate a candidate salt
    # Check if the salt already exists, if so, create another one
    while Honeywords.objects.filter(salt=salt).exists():
      salt = get_random_string()
    return salt   # Return the unique salt
```

HoneywordHasher.verify(self, password, encoded)

```
# Pull apart the encoded password that was stored in the database
algorithm, iterations, salt, dummy= encoded.split(‘$’, 3)
# Grab the honeyword hashes from the database
hashes = pickle.loads(Sweetwords.objects.get(salt = salt).sweetwords)
# Use a helper function to hash the provided password
hash = self.hash(password, salt, int(iterations))
if hash in hashes: # Make sure the submitted hash is in the local database
  #Check with the honeychecker to see if the index is correct
  return honeychecker.check_index(salt, hashes.index(hash))
return False  #Return false if the hash isn’t even in the local database
```

HoneywordHasher.encode(self, password, salt, iterations)

```
#Put the real password in the list
sweetwords = [password]
# Add generated honeywords to the list as well
sweetwords.extend(honeywordgen.gen(password, <bases>,
    [<pwfiles>]))
# Add tweaks of all the sweetwords to the list
for i in range(<bases+1>):
    sweetwords.extend(honeywordtweak.tweak(passwords[i], <tweaks>))
# Randomly permute the sweetword order
random.shuffle(sweetwords)
```

HoneywordHasher.encode(self, password, salt, iterations)

```
hashes = [ ]
for swd in sweetwords:   # Hash all of the passwords
    hashes.append(self.hash(swd, salt, iterations))
# Update the honeychecker with a new salt and index
self.honeychecker.update_index(salt, sweetwords.index(password))
# Create a new honeyword entry for the local database
h = Sweetwords(salt = salt, sweetwords = pickle.dumps(hashes))
h.save()  #Write to the database
# Return what is expected for storage in the User database
return “%s$%d$%s$%s” % (self.algorithm, iterations, salt, hashes[0])
```

honeywordgen.py
Modifying generation parameters

- Downloaded from: http://people.csail.mit.edu/rivest/honeywords/gen.py
- Black = existing code

  Blue = additions

  Red = deletions

```
#######################################################################
 #### PARAMETERS CONTROLLING PASSWORD GENERATION
nL = 8   # password must have at least nL letters
nD = 1  # password must have at least nD digit
nS = 0  # password must have at least nS special (non-letter non-digit)
```

honeywordgen.py (cont)
Ensure generated passwords are unique

```
def generate_passwords( n, pw_list ):
""" print n passwords and return list of them """
ans = [ ]
for t in range( n ):
    pw = make_password(pw_list)
    while pw in ans:
      pw = make_password(pw_list)
    ans.append( pw )
return ans
```

honeywordgen.py
Make a generation function, remove system parameters

```
def ~~main()~~gen(password, n, filenames):
    # get number of passwords desired
    ~~if len(sys.argv) > 1:~~
        ~~n = int(sys.argv[1])~~
    ~~else:~~
        ~~n = 19~~
    ~~# read password files~~
    ~~filenames = sys.argv[2:]          # skip "gen.py" and n~~
    pw_list = read_password_files(filenames)
    …
# import cProfile
# cProfile.run("main()")
~~main()~~
```

Tweaking function - pseudocode

- Identify the piece of the password you will tweak (input, length)
  - If that piece is numeric, replace with different digits of same length

```
str(random.randrange(pow(10, length))).zfill(length)
```

  - If symbols, create a translation table

```
symbolchars = [‘!’, ‘@’, ‘#’, ‘$’, ‘%’, ‘^’, ‘&’, ‘*’, ‘(‘, ‘)’, ‘_’, ‘+’, ‘=‘, ‘-’, ‘`’, ‘~’, ‘<‘, ‘>’,
   ‘?’, ‘/’, ‘\\’, ‘\”, ‘”’, ‘;’, ‘:’, ‘{‘, ‘}’, ‘[‘, ‘]’, ‘|’, ‘.’, ‘\,’, ‘ ‘]
shuffled = random.shuffle(copy.deepcopy(symbolchars))
translation = str.maketrans(symbolchars, shuffled)
input.translate(translation)
```

Sweetwords Database

```
from django.db import models

class Sweetwords(models.Model)
    # Our index is the salt value.
    salt = models.CharField(max_length=128)
    # Allow the sweetwords field to store a huge number of hashes
    sweetwords = models.CharField(max_length = 65536)
```

Honeychecker

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
indices = { } # Maps the salt to the correct index for that salt

def check_index(salt, index):
    if salt in indices: # User exists
      #If index matches, user is authenticated
      # Otherwise a honeyword was submitted – should probably alert
      return indices[salt] == index
    return False
```

Honeychecker (cont)

```
def update_index(salt, index):
    indices[salt] = index   #Add new salt/index pairing to dictionary

def main():  # Setup server, register functions and then start running
    honeychecker = SimpleXMLRPCServer((“<ip_addr>”, <port>))
    honeychecker.register_function(check_index, ‘check_index’)
    honeychecker.register_function(update_index, ‘update_index’)
    honeychecker.server_forever()

main()   # Call main to get things going once everything is setup
```

settings.py
Change the settings file

```
INSTALLED_APPS = (
…
‘django.contrib.staticfiles’,
‘honeywords’,
)
PASSWORD_HASHERS = (
‘honeywords.hashers.HoneywordHasher’,
‘django.contrib.auth.hashers.PBKDF2PasswordHasher’,
…
)
```

Create the tables and go!

- Now you need to make those settings take effect

```
python manage.py sql honeywords

python manage.py syncdb
```

- That's it. Your up and running!
- As users log in their passwords will be converted to honeywords, the honeychecker will be notified of the new mapping, and their password will be better protected in case you are ever breached.

References

- http://people.csail.mit.edu/rivest/honeywords/
- https://docs.djangoproject.com/en/dev/topics/auth/passwords/
- https://docs.djangoproject.com/en/1.6/intro/tutorial01/