Password Vulnerability

Access Control

- Two parts to access control…
- **Authentication:** Are you who you say you are?
  - Determine whether access is allowed or not
  - Authenticate human to machine
  - Or, possibly, machine to machine
- **Authorization:** Are you allowed to do that?
  - Once you have access, what can you do?
  - Enforces limits on actions
- Note: “access control” often used as synonym for authorization

Are You Who You Say You Are?

- Authenticate a human to a machine?
- Can be based on…
  - Something you **know**
    - For example, a password
  - Something you **have**
    - For example, a smartcard
  - Something you **are**
    - For example, your fingerprint

Something You Know

- Passwords
- Lots of things act as passwords!
  - PIN
  - Social security number
  - Mother's maiden name
  - Date of birth
  - Name of your pet, etc.

Trouble with Passwords

- “Passwords are one of the biggest practical problems facing security engineers today.”
- “Humans are incapable of securely storing high-quality cryptographic keys, and they have unacceptable speed and accuracy when performing cryptographic operations.”

Why Passwords?

- Why is “something you know” more popular than “something you have” and “something you are”?
- **Cost**: passwords are free
- **Convenience**: easier for sysadmin to reset pwd than to issue a new thumb

Keys vs Passwords

- **Crypto keys**
- Suppose key is 64 bits
- Then 2^{64} keys
- Choose key at random…
- …then attacker must try about 2^{63} keys

- **Passwords**
- Suppose passwords are 8 characters, and 256 different characters
- Then $256^8 = 2^{64}$ pwds
- Users do not select passwords at random
- Attacker has far less than 2^{63} pwds to try **(dictionary attack)**

Good and Bad Passwords

- Bad passwords
  - frank
  - Fido
  - Password
  - incorrect
  - Pikachu
  - 102560
  - AustinStamp

- Good Passwords?
  - jfIej,43j-EmmL+y
  - 09864376537263
  - P0kem0N
  - FSa7Yago
  - 0nceuP0nAt1m8
  - PokeGCTall150

Password Experiment

- Three groups of users — each group advised to select passwords as follows
  - **Group A:** At least 6 chars, 1 non-letter
  - **Group B:** Password based on passphrase
  - winner → **Group C:** 8 random characters
- Results
  - **Group A:** About 30% of pwds easy to crack
  - **Group B:** About 10% cracked
    - Passwords easy to remember
  - **Group C:** About 10% cracked
    - Passwords hard to remember

Password Experiment

- User compliance hard to achieve
- In each case, 1/3rd did not comply
  - And about 1/3rd of those easy to crack!
- Assigned passwords sometimes best
- If passwords not assigned, best advice is…
  - Choose passwords based on passphrase
  - Use pwd cracking tool to test for weak pwds
- Require periodic password changes?

Attacks on Passwords

- Attacker could…
  - Target one particular account
  - Target any account on system
  - Target any account on any system
  - Attempt denial of service (DoS) attack
- Common attack path
  - Outsider $\rightarrow$ normal user $\rightarrow$ administrator
  - May only require **one** weak password!

Password Retry

- Suppose system locks after 3 bad passwords. How long should it lock?
  - 5 seconds
  - 5 minutes
  - Until SA restores service
- What are +’s and -’s of each?

Password File?

- Bad idea to store passwords in a file
- But we need to verify passwords
- Solution? **Hash** passwords
  - Store y = h(password)
  - Can verify entered password by hashing

- If Trudy obtains the password file, she does not (directly) obtain passwords
- But Trudy can try a *forward search*
  - Guess x and check whether y = h(x)

passw0rd

$\downarrow h_{MD5}$

BED128365216C019988915ED3ADD75FB

Dictionary Attack

- Trudy pre-computes h(x) for all x in a **dictionary** of common passwords
- Suppose Trudy gets access to password file containing hashed passwords
  – She only needs to compare hashes to her pre-computed dictionary
  – After one-time work of computing hashes in dictionary, actual attack is trivial
- Can we prevent this forward search attack? Or at least make it more difficult?

- Hash password with **salt**
- Choose random salt s and compute

  $$y = h(\text{password}, s)$$

  and store (s,y) in the password file
- Note that the salt s is not secret
- Still easy to verify salted password
- But lots more work for Trudy
  - Why?

Password Cracking: Do the Math

- Assumptions:
- Pwds are 8 chars, 128 choices per character
  - Then $128^8 = 2^{56}$ possible passwords
- There is a **password file** with 2^{10} pwds
- Attacker has **dictionary** of 2^{20} common pwds
- **Probability** 1/4 that password is in dictionary
- **Work** is measured by number of hashes

Password Cracking: Case I

- Attack 1: specific password ***without*** using a dictionary
  - E.g., Alice’s password
  - Must try $2^{56}/2 = 2^{55}$ on average
  - Like exhaustive key search
- Does **salt** help in this case?

Password Cracking: Case II

- Attack 1 specific password ***with*** dictionary
- With **salt**
  - Expected work: 1/4 (2^{19}) + 3/4 (2^{55}) ≈ $2^{54.6}$
  - In practice, try all pwds in dictionary…
  - …then work is at most 2^{20} and probability of success is 1/4
- What if **no salt** is used?
  - One-time work to compute dictionary: 2^{20}
  - Expected work is of same order as above

Password Cracking: Case III

- Attack3: Any of 1024 pwds in file, ***without*** dictionary
  - Assume all 2^{10} passwords are distinct
  - Need 2^{55} **comparisons** before expect to find pwd
- If **no salt** is used
  - Each computed hash yields 2^{10} comparisons
  - So expected work (hashes) is $2^{55}/2^{10} = 2^{45}$
- If **salt** is used
  - Expected work is 2^{55}
  - Each comparison requires a hash computation

Password Cracking: Case IV

- Attack 4: Any of 1024 pwds in file, ***with*** dictionary
  - Prob. one or more pwd in dict.: $1 – (3/4)^{1024} \approx 1$
  - So, we ignore case where no pwd is in dictionary
- What if **no salt** is used?
  - If dictionary hashes not precomputed, work is about $2^{19}/2^{10} = 2^9$
- If **salt** is used, expected work less than 2^{22}
  - Work ≈ size of dictionary / P(pwd in dictionary)

$$\frac{1}{4}(2^{19}) + \frac{3}{4}.\frac{1}{4}(2^{20} + 2^{19}) + \left(\frac{3}{4}\right)^2 \frac{1}{4}(2.\,2^{20} + 2^{19}) + \cdots + \left(\frac{3}{4}\right)^{1023} \frac{1}{4}(1023.\,2^{20} + 2^{19})$$

$<2^{22}$

Other Password Issues

- Too many passwords to remember
  - Results in password reuse
  - Why is this a problem?
- Who suffers from bad password?
  - Login password vs ATM PIN
- Failure to change default passwords
- Social Engineering
- Bugs, keystroke logging, spyware, etc.

Passwords

- The bottom line…
- **Password attacks are too easy**
  - Often, one weak password will break security
  - Users choose bad passwords
  - Social engineering attacks, etc.
- Trudy has (almost) all of the advantages
- All of the math favors bad guys
- Passwords are a **BIG** security problem
  - And will continue to be a problem

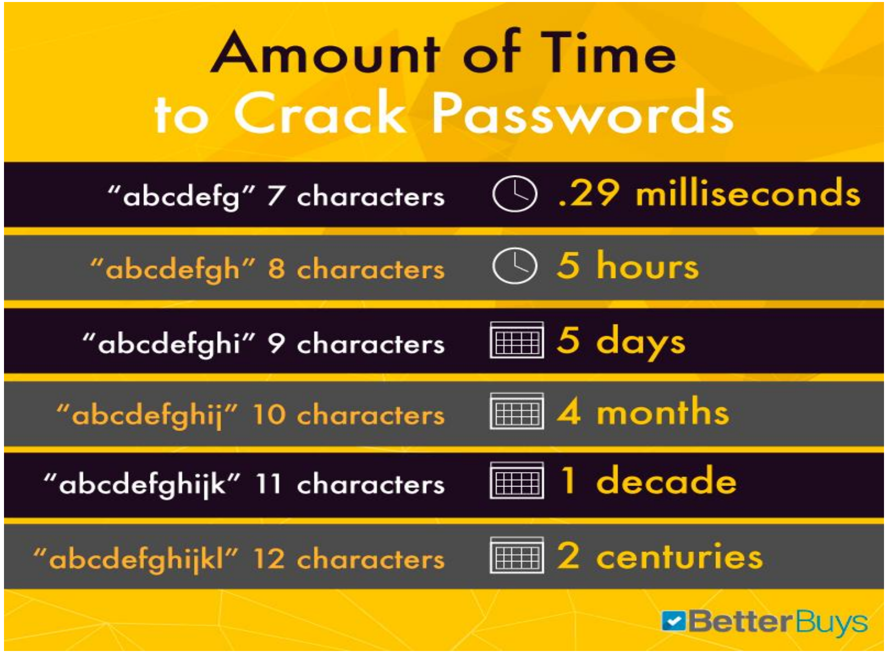
Password Cracking Tools

- Popular password cracking tools
  - Password Crackers
    - http://www.pwcrack.com/index.shtml
  - L0phtCrack (Windows)
    - https://l0phtcrack.gitlab.io/
  - John the Ripper (Unix)
    - http://www.openwall.com/john/
- System Admin should use these tools to test for weak passwords since attackers will
- Good articles on password cracking
  - Various password research articles are maintained in
    - http://passwordresearch.com/
  - Passwords revealed by sweet deal
    - http://news.bbc.co.uk/2/hi/technology/3639679.stm

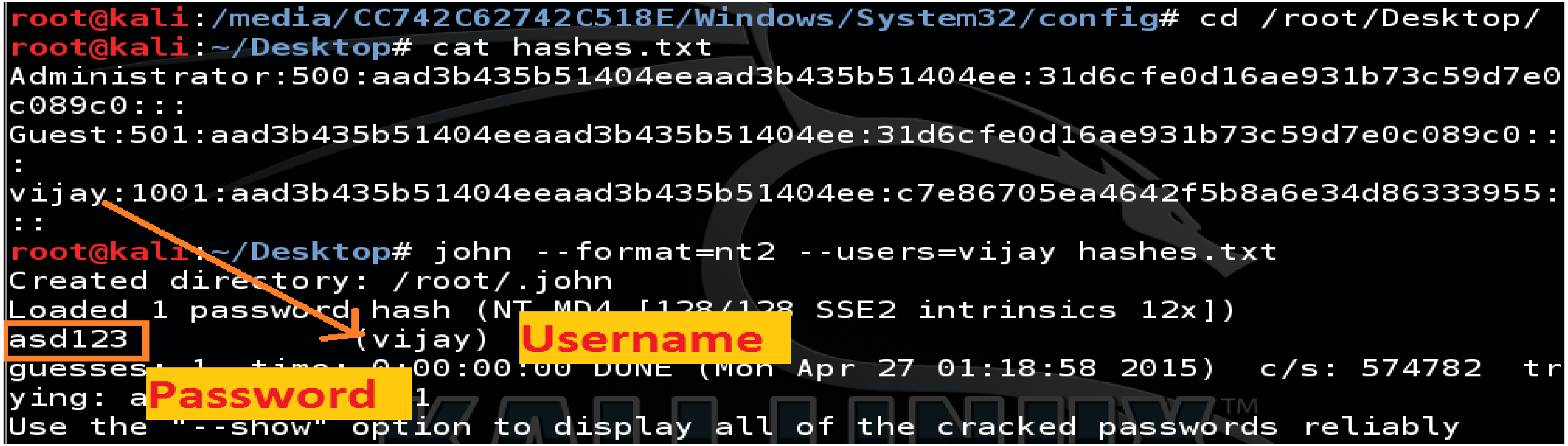
https://www.betterbuys.com/estimating-password-cracking-times/

Password Cracking

- Initially for password cracking, brute force attack was conducted
- However, the time complexity of cracking passwords of a reasonably large length is quite high
- Now the availability of password cracking algorithms have reduced this high time requirement



Password Cracking



A snapshot using john the ripper cracking tool

Some Interesting Facts

- Password files of many giant web-based organizations have been compromised
  - Adobe (150 million)
  - Evernote (50 million)
  - Anthem (40 million)
  - RockYou (32 million)
  - Tianya (28 million)
  - Dodonew (16 million)
  - Gmail (4.9 million)
  - …
- Interestingly, some of the breaches came into notice after a long time of the actual leakage
  - Yahoo: leakage 2013 and revealed in 2017
  - Dropbox: between leakage and revealed 4 years
  - Myspace: between leakage and revealed 8 years

Detection of timely password breach is important

How to detect password breach?

Honeywords: Making Password-Cracking Detectable

Ari Juels
RSA Laboratories
Cambridge, MA, USA
ari.juels@rsa.com

Ronald L. Rivest
MIT CSAIL
Cambridge, MA, USA
rivest@mit.edu

ABSTRACT

We propose a simple method for improving the security of hashed passwords: the maintenance of additional "honeywords" (false passwords) associated with each user's account. An adversary who steals a file of hashed passwords and inverts the hash function cannot tell if he has found the password or a honeyword. The attempted use of a honeyword for login sets off an alarm. An auxiliary server (the "honeychecker") can distinguish the user password from honeywords for the login routine, and will set off an alarm if a honeyword is submitted.

Times [32]. The past year has also seen numerous high-profile thefts of files containing consumers' passwords; the hashed passwords of Evernote's 50 million users were exposed [20] as were those of users at Yahoo, LinkedIn, and eHarmony, among others [19].

One approach to improving the situation is to make password hashing more complex and time-consuming. This is the idea behind the "Password Hashing Competition."[2] This approach can help, but also slows down the authentication process for legitimate users, and doesn't make successful password cracking easier to detect.

Sometimes administrators set up fake user accounts ("hon-

ACM CCS 2013

Assumption

- The paper assumes a computer system with n users $u_1, u_2, \ldots, u_n$
- p_i denotes the password of i^{th} user u_i
- This p_i is the legitimate password that user uses for login purpose
- Traditional systems store
  - Unsalted: $(u_i, H(p_i))$ or Salted: $(u_i, H(p_i, s_i))$

To setup the honeyword based system

- For each user u_i, a list W_i of distinct words (called "potential passwords" or more briefly, "sweetwords") is represented: $W_i = (w_{i,1}, w_{i,2} \ldots, w_{i,k})$
  - The list W_i of sweetwords thus contains one sugarword (the password) and $(k - 1)$ honeywords (the chaff)
  - An auxiliary secure server, called the "honeychecker", is used to store the index of the sugarwords

Honeyword System

- The system's login routine needs to determine whether a proffered password g is
  - user's password or not
  - a honeyword or not
  - anything other than password or honeywords
- If the adversary has entered one of the user's honeywords,
  - then an appropriate action takes place (determined by policy)

Honeyword System (cntd)

- When user u_i changes her password, or sets it up when her account is first initialized, procedure Gen(k) is used to obtain
  - a new list W_i of k sweetwords,
  - the list H_i of their hashes, and
  - the value c(i) of the index of the correct password p_i in W_i
- Then, securely notify the honeychecker of the new value of c(i), and update the user's entry in the file F to (u_i, H_i)

Honeyword Generation

- Based on the impact on the user interface (UI) for password change
  - With legacy-UI procedures
    - the password-change UI is unchanged.
  - With modified-UI procedures
    - the password-change UI is modified to allow for better password/honeyword generation.

Legacy-UI password changes

- With a legacy-UI method,
  - the password-change procedure asks the user for the new password
  - The UI does not tell the user about the use of honeywords, nor interact with user to influence the password choice

Honeyword generation procedure can be changed without needing to notify anyone or to change the UI

Honeyword Generation

- We start with a password p_i supplied by user u_i
- The system then generates a set of $k-1$ honeywords "similar in style" to the password p_i, or at least plausible as legitimate passwords, so that an adversary will have difficulty in identifying p_i in the list W_i of all sweetwords for user u_i

Chaffing

- The password p_i is picked, and then the honeyword generation procedure $Gen(k, p_i)$ or "chaff procedure" generates a set of k − 1 additional distinct honeywords ("chaff")
- Note that the honeywords may depend upon the password p_i
- The password and the honeywords are placed into a list W_i, in random order
- The value $c(i)$ is set equal to the index of p_i in this list

Chaffing by tweaking

- "tweak" selected character positions of the password to obtain the honeywords
- Let *t* denote the desired number of positions to tweak (such as t = 2 or t = 3).
  - If password is "BG+7y45", then the list W_i might be (for tail-tweaking with t = 3 and k = 4):
  - BG+7q03, BG+7m55, BG+7y45, BG+7o92
- Another example
  - chaffing-by tweaking-digits for t = 2:
  - 42*flavors, 57*flavors, 18*flavors

Chaffing-with-a-password-model

- Generates honeywords using a probabilistic model of real passwords
- Unlike the previous chaffing methods, this method does not necessarily need the password in order to generate the honeywords, and it can generate honeywords of widely varying strength

An Example of 19 generated honeywords

```
kebrton1        02123dia
a71ger          forlinux
1erapc          sbgo864959
aiwkme523       aj1aob12
9,50PEe]KV.0?RI0tc&L-:IJ"b+Wol<*[!NWT/pb
xyqi3tbato      a3915
#NDYRODD_!!     venlorhan
pizzhemix01     dfdhusZ2
svenіresly      'Sb123
mobopy          WORFmgthness
```

Modeling Syntax

- [Bojinov et al., ESORICS 2010] propose an interesting approach based on the work of [Weir et al., IEEE SP, 2009] to chaffing-with-a-passwordmodel in which honeywords are generated using the same syntax as the password
- Here, honeywords do depend on the password

Modeling Syntax

- The password is parsed into a sequence of "tokens," each representing a distinct syntactic element—a word, number, or set of special characters
  - For example, the password mice3blind might be decomposed into the token sequence W4 | D1 |W5
  - meaning a 4-letter word followed by a 1-digit number and then a 5-letter word

- Honeywords are then generated by replacing tokens with randomly selected values that match
  - For example, the choice W4 ←“gold,” D1 ← ‘5" , W5 ← “rings” would yield the honeyword gold5rings
  - Replacements for word tokens are selected from a dictionary provided as input to the generation algorithm

Chaffing with “tough nuts”

- One might also like to have some honeywords that are much harder to crack than the average
- So, the adversary would not then (as we have been assuming) be faced with a completely broken list of sweetwords, but rather only a partial list
- There may possibly be some uncracked hashes (represented by ‘ ? ’ here) still to work on, with the correct password possibly among them

Chaffing with “tough nuts”

- For example, what should the adversary do with the following list?

`gt79, tom@yahoo,` $\boxed{?}$`, g*7rn45, rabid/30frogs!,` $\boxed{?}$

- Having some “tough nuts” among the honeywords might give the adversary additional reason to pause before diving in and trying to log in with one of the cracked ones
- “Tough nuts” represent potentially correct passwords whose plausibility the adversary cannot evaluate

Modified UI Password Changes

- Take-a-tail
- Utilizes a modified UI for password changes
- Similar to chaffing by tail tweaking method
- Except that the tail is now randomly chosen by the system

That is, the password-change UI is changed from:

```
Enter a new password:
```

to something like:

```
Propose a password: ●●●●●●●
Append '413' to make your new password.
Enter your new password: ●●●●●●●●●●
```

Password: RedEye2

New Password: RedEye2413

Honeywords can now be generated using chaffing by tail tweaking method

Other ways of generating honeywords

- Random pick honeyword generation
- A modified UI procedure
- Flat honeywords

- Generate a list W_i of k distinct sweetwords in some arbitrary manner
  - May involve interaction with the users
- Also, pick randomly an element of this as the password
- The other elements become honeywords

Random-Pick Honeyword Generation

User may supply following six sweetwords-

```
4Tniners     all41&14all   i8apickle
sin(pi/2)    \{1,2,3\}     AB12:YZ90
```

System may say pick any one of them as password

Some Issues

- User will have to create k number of sweetwords
- Users may remember a sweetword and may mistakenly enter that as a password
- System may treat that as honeyword

It may be better if the sweetwords are generated algorithmically

Next Paper

284 IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 13, NO. 2, MARCH/APRIL 2016

Achieving Flatness: Selecting the Honeywords from Existing User Passwords

Imran Erguler

Abstract—Recently, Juels and Rivest proposed honeywords (decoy passwords) to detect attacks against hashed password databases. For each user account, the legitimate password is stored with several honeywords in order to sense impersonation. If honeywords are selected properly, a cyber-attacker who steals a file of hashed passwords cannot be sure if it is the real password or a honeyword for any account. Moreover, entering with a honeyword to login will trigger an alarm notifying the administrator about a password file breach. At the expense of increasing the storage requirement by 20 times, the authors introduce a simple and effective solution to the detection of password file disclosure events. In this study, we scrutinize the honeyword system and present some remarks to highlight possible weak points. Also, we suggest an alternative approach that selects the honeywords from existing user passwords in the system in order to provide realistic honeywords—a perfectly flat honeyword generation method—and also to reduce storage cost of the honeyword scheme.

Index Terms—Authentication, honeypot, honeywords, login, passwords, password cracking

Few Remarks by Erguler

- Remarks on Take-a-tail
  - “Although this method strengthens the password, to our point of view, it is impractical—some users even forget the passwords that they determined”
- Remarks on Chaffing-by-tweaking
  - “Many users have the propensity to choose the numbers included in passwords related to a special date, e.g. birthday, anniversary or an important historical event”

Remarks by Erguler

A potential value of a
valid year

$alex6323$	$alex9058$	$\boxed{\textbf{alex1992}}$
$alex1270$	$alex0976$	$alex2785$
$alex5469$	$alex8147$	$alex9705.$

The digits in the honeywords don't make sense. Thus,
alex1992 makes sense for an adversary

Remarks by Erguler

- Remarks on Chaffing-by-tweaking
  - Apart from the use of a date in passwords, many users prefer to append consecutive numbers to their password heads, like ’123’, ’1234’, due to the tendency of users to choose rememberable number patterns

A randomly replacing technique like this model leads an adversary to make a natural selection

Remarks by Erguler

- Remarks on Chaffing-with-a-Password-Model
  - Leaked password database shows that some of the passwords have a well known pattern

    bond007 james007
    007bond 007007.

  - Even modeling syntax approach looses its effectiveness against such passwords

Remarks by Erguler

- Remarks on correlation
  - If there is a correlation between username and password then the password can be easily distinguishable
  - Example:
    - username/password- alice/alice123
    - username/password- peterparker/spiderman1992

Remarks by Erguler

- Remarks on tough-nuts
  - “an adversary may suppose that most of the passwords made up of simple words and digit combinations, not a tough nut. Hence, it is reasonable for this adversary to conduct her classic attack with skipping tough nuts contrarily to authors’ expectations”

Remarks by Erguler

- Remarks on hybrid method
  - "… previous remarks are also valid for this case, e.g. in the above example an adversary may make plausible guesses"

$happy9679$	$apple1422$	$angel2656$
$happy9757$	**apple1903**	$angel2036$
$happy9743$	$apple1172$	$angel2849$
$happy9392$	$apple1792$	$angel2562.$

Remarks by Erguler

- Remarks on DoS Attack
  - “the system limits for unsuccessful login attempts as n, i.e., after n consecutive wrong password trials the account will be blocked. Nonetheless, if the correct password is entered before n is reached, then system resets the wrong password counter”
  - “the adversary logins with the correct password at each nth attempt to avoid blocking of the account”

Two major issues

- Flatness of honeywords
  - The main purpose of honeywords will not serve
- Chance of hitting a honeyword
  - Leading to the DoS attack

Erguler's Proposal

- Use of existing passwords to simulate the honeywords
- For each account k-1 existing password indexes, which we call honeyindexes, are randomly assigned to a newly created account of u_i, where k>=2
- A random index number is given to this account and hash of the correct password is kept with the correct index in a list
- In another list u_i is stored with an integer set which consists of the honeyindexes and the correct index

Erguler's Proposal

Username	Honeyindex Set
agent-lisa	$(93, 16, 626, \ldots, 94, 931)$
alexius	$(15, 476, 51, 443, \ldots, 88, 429)$
baba13	$(3, 62107, \ldots, 91, 233)$
⋮	⋮
zack_tayland	$(1, 009, 23, 471, \ldots, 47, 623)$
zoom42	$(63, 51234, \ldots, 72, 382)$

Each username is paired with k numbers as sweetindexes and each of which points to real passwords in the system

Two major advantages

- Less storage
- Achieving flatness

Initialization

- Firstly, T fake accounts (honeypots) are created with their passwords
- An unique index value between [1,N] is assigned to each created account randomly
- Then k-1 numbers are randomly selected from the index list for each account as honeyindex set
  - $X_i=\{x_{i,1},x_{i,2},\ldots,x_{i,k}\}$
  - One of the index of X_i is the correct index

Intialization

- Two passwords files F1 and F2 are used in the main server
- F1 stores username and honeyindex set $<hu_i, X_i>$, here hu_i denotes honeypot account
  - F1 is sorted alphabetically
- F2 keeps the index number and corresponding hash of the password $<c_i, H(p_i)>$
  - F2 is sorted according to the index values
  - Let S_I denote the index column and S_H represents the corresponding hash value

Example

Let's create honeypot account <macbeth, master2014>
An index number 1008 is assigned to it randomly

Index No	Hash of Password
$\vdots$	$\vdots$
1,008	$H(master2014)$
$\vdots$	$\vdots$

S_I	S_H
3	$H(p_3)$
7	$H(p_7)$
85	$H(p_{85})$
$\vdots$	$\vdots$
100,000	$H(p_{100000})$
100,004	$H(p_{100004})$

Contents of F2

- Then k-1 numbers are randomly chosen from S_l of F_2
- F1 is generated by combining with the correct index

Username	Honeyindex Set
⋮	⋮
macbeth	(42, 96, 104, 1,008, 7,201, 23,008)
⋮	⋮

Contents of F1

Registration

- A legacy-UI is preferred
- The user as $<u_i, p_i>$ registers to the system
- The honeyindex generator algorithm $Gen(k,S_l) \rightarrow c_i, X_i$ which outputs c_i as the correct index for u_i and the honeyindexes $X_i=\{x_{i,1},x_{i,2},\ldots,x_{i,k}\}$
- $Gen(k,S_l)$ produces X_i randomly picking a number c_i not in S_l
- After c_i and X_i are obtained, $<u_i, c_i>$ pair is delivered to the honeychecker

After the Registration Process, Change of F_2 is Illustrated on the Left, while Update of F_1 is Shown on the Right

S_I	S_H
3	$H(p_3)$
$\vdots$	$\vdots$
c_i	$H(p_i)$
$\vdots$	$\vdots$
$\vdots$	$\vdots$
100000	$H(p_{100000})$
100004	$H(p_{100004})$

Username	Honeyindex Set
agent-lisa	$(93, 16626, \ldots, 94931)$
alexius	$(15476, 51443, \ldots, 88429)$
baba13	$(3, 62107, \ldots, 91233)$
$\vdots$	$\vdots$
u_i	X_i
$\vdots$	$\vdots$
zack_tayland	$(1009, 23471, \ldots, 47623)$
zoom42	$(63, 51234, \ldots, 72382)$

Honeychecker

- Store the correct indexes of each account
- Communicates with the main server through secure channel in an authenticated manner
- The honeychecker executes two commands
- Set: <c_i, u_i >
  - Sets correct password index c_i for the user u_i
- Check: <u_i , j>
  - Checks whether c_i for u_i is equal to given j
  - Returns the result and if equality does not hold, notifies system a honeyword situation

Login process

- Need to check whether the entered password, g, is correct for the corresponding username u_i
- First the X_i of the corresponding u_i is attained from the F1 file
- The hash values stored in F2 file for the respective indices in X_i are compared with $H(g)$ to find a match
- If a match is not obtained, then it means that g is neither the correct password, nor one of the honeywords, i.e., login fails
- If $H(g)$ is found in the list, then the main server checks whether the account is a honeypot
- If it is a honeypot, then it follows a predefined security policy against the password disclosure scenario
- If, however, $H(g)$ is in the list and it is not a honeypot, the corresponding j from X_i is delivered to honeychecker with username as $< u_i, j >$ to verify it is the correct index
- Honeychecker controls whether $j=c_i$ and returns the result to the main server
- If it is not equal, then it assured that the proffered password is a honeyword and adequate actions should be taken depending on the policy
- Otherwise, login is successful

Assumptions

- We suppose that the adversary can invert most or many of the password hashes in file F2
- If user hits a password of another account in the system and the same user hits this situation more than once (trying with other passwords in F2), the system should turn on additional logging of the user's activities to detect a possible DoS attack and to attribute the adversary, besides the incorrect login attempt case proceeds as usual.

Assumptions (2)

- If a password, whose hash value is in the S_H of the F_2, is entered in wrong login attempts for more than once, the system should take actions against a possible DoS alarm
- In order to increase the number of unique passwords in the system, i.e., reduce common passwords, users should be forced to adhere to a password-composition policy

Assumptions (3)

- A username should not be correlated with its password
- To avoid occurrence of a high number of common passwords in the system, the user should be driven to choose another password when the created password is in the list of 1,000 most common passwords

DoS Attack

- Adversary has knowledge $m + 1$ username and respective passwords in the system
  - as $(<u_a, p_a>, \ldots, <u_{a+m}, p_{a+m}>)$
- Create m accounts with the same password as p_z
- If p_z is assigned by the system as a honeyword, then the adversary mounts a DoS attack by entering with the system $<u_y, p_z>$ pair

DoS Attack Analysis

- Let $Pr(p_z \varepsilon W_y)$ denote the probability that p_z is assigned as one of the honeywords for u_y
- It is also the success probability of the adversary for DoS attack

$$\Pr(p_z \in W_y) = 1 - \left(\frac{N-m}{N}\right)^k.$$

DoS Attack Analysis

- If $N = 1{,}000{,}000$, $k = 20$ and $m = 100$,
  - then adversary succeeds in realizing the described attack with a probability of 0.002
- If $N = 1{,}000$, $k = 20$ and $m = 10$,
  - then adversary succeeds in realizing the described attack with a probability of 0.18

The success of the adversary directly depends on (m/N)

Password guessing

- If the adversary randomly picks an account from the list in F1 and then tries to login with a guessed password, then her success will depend on
  - Firstly, the selected account is not a honeypot (decoy) account
  - Secondly, guessing the correct password p_i out of k sweetwords

- Let Prob(success) represents the probability that the adversary makes a correct guess for a randomly picked username
- $Prob(success) = \frac{N-T}{N} . \frac{1}{k}$

T: the number of honeypot accounts in the system

A convenient choice for T could be $N^{1/2}$

For k =20 and N = 1,000,000, she picks the correct password p_i with 5 percent probability

Storage Cost

- A typical password file system requires
  - hN plus storage for usernames,
  - where N number of users in the system
  - h denotes length of password hash in bytes
- Honeyword-based system requires khN storage,
  - where k the number of the sweetwords assigned to each account

Gain in storage cost

- If each index requires 4 bytes and the storage cost becomes:
  - 4kN + hN + 4N
- Gain in storage cost compared to the original scheme
  - $\frac{4kN+hN+4N}{khN} = \frac{4k+h+4}{kh}$