LUG@UCLA TECH TALK

PASSWORD SECURITY

Presented by David Nguyen
AUTHENTICATION

- Passwords solve the problem of authentication: proving your identity
- Three ways of authentication
  - Something you know (e.g. passwords, pins, security questions)
  - Something you have (e.g. pgp keys, software tokens)
  - Something you are (e.g. biometrics, MAC address)
- All have drawbacks and strengths
HOW DO PASSWORDS WORK

• That's easy. You just tell the server your username and password and it authenticates you, right?
• The implementation is harder. You have to do this "securely".
  ▪ How do you send your username and password over.
  ▪ How does the server store their usernames and passwords.
QUICK RECAP FOR ENCRYPTION

- Symmetric Encryption uses 1 key; Asymmetric Encryption uses 2 keys
  - AES-256 is popular symmetric encryption standard
    - AES is the algorithm, 256 bits is the block size
- Hashing: Maps data of arbitrary size to data of fixed size
  - Cryptographic Hashing: Hashing function is one way which is easy to compute and hard to invert
- Hashing is not encryption!
HOW TO SEND PASSWORDS SECURELY

SSL Client

(3) Verify server certificate. Check cryptographic parameters

(1) "client hello"
Cryptographic information

(2) "server hello"
CipherSuite
Server certificate
"client certificate request" (optional)

(4) Client key exchange
Send secret key information
(encrypted with server public key)
(5) Send client certificate

(6) Verify client certificate (if required)

(7) Client "finished"

(8) Server "finished"

(9) Exchange messages
(encrypted with shared secret key)

SSL Server
HOW SHOULD SHOULDN'T SERVERS STORE PASSWORDS

• Obviously storing it in plain text is wrong
  ▪ Anyone who has access to the file can log in as anyone
• How about encrypting the file with some symmetric (or even asymmetric) cipher?
  ▪ Also bad! You're trusting the person with the password or private key
  ▪ In 2013 Adobe leaked their passwords this way
• Hash
  ▪ No! Rainbow tables will beat that.
• Hash and salt?
  ▪ Nope. For GPUs its easy to compute billions of hashes
THE "FINAL" ANSWER

- "If at first you don't succeed, try, try again."
- Repeatedly hash your password with that salt "thousands" of times
- PBKDF2, bcrypt, scrypt are three popular ones
- This solves the problem of:
  - Trust, rainbow table attacks, and brute force attempts
- However even this is pretty outdated (2013)
- New cryptohashing standards will be created, but the end goal is the same
  - Prevent brute-force attempts on collisions
LET'S TALK ABOUT PASSWORDS

• What makes a strong password?
  ▪ Is hunter2 really such a bad password?
• Password strength meters are misleading
  ▪ I used http://www.passwordmeter.com/
    ○ The password LUGatUCLA got 32/100 (weak)
    ○ LUGatUCLA2 got 61/100 (strong)
• We'll measure password strength with password entropy or bits of entropy
We measure a password's randomness in bits and say that it is the \( \log_2 \) of the number of guesses needed to brute force the password.

Equation: \( H = \log_2 (N^L) \)
- \( H \) is the number of bits of entropy, \( N \) is size of set you're choosing from, and \( L \) is the length of the password in terms of the elements in the set.

For example, if we looked into /dev/urandom and asked for a password of length 10 from the set of printable ASCII characters we might get:
- Z1-frlAtrl
- \( H = \log_2(95^{10}) = 10 \times \log_2(95) = 6.56 \times 10 = \sim 65 \text{ bits of entropy} \)
PASSWORD ENTROPY (CONTD.)

- correcthorsebatterystaple
  - each word: correct, horse, battery, staple comes RANDOMLY from a list of ~2048 words
    - Did I say RANDOMLY?! IT HAS TO BE RANDOM!
  - The length of the password is 4 in terms of these words
  - thus \( H = \log_2(\sim2048^4) = 4 \log_2(\sim2048) = 4 \times 11 = \sim44 \) bits
CORRECT HORSE BATTERY STAPLE

Tr0ub4dor & $3$

- **UNCOMMON (Non-Gibberish) BASE WORD**
- **ORDER UNKNOWN**
- **2^{19} = 3 DAYS AT 1000 GUESSES/SEC**
  - (Plausible attack on a weak remote web service. Yes, cracking a stolen hash is faster, but it's not what the average user should worry about.)
- **DIFFICULTY TO GUESS:** EASY
- **DIFFICULTY TO REMEMBER:** HARD

Correct horse battery staple

- **FOUR RANDOM COMMON WORDS**
- **2^{44} = 530 YEARS AT 1000 GUESSES/SEC**
- **DIFFICULTY TO GUESS:** HARD
- **DIFFICULTY TO REMEMBER:** YOU'VE ALREADY MEMORIZED IT

Through 20 years of effort, we've successfully trained everyone to use passwords that are hard for humans to remember, but easy for computers to guess.
HUMAN GENERATED PASSWORD ENTROPY

- Humans are predictable and have certain patterns and tendencies. We can't use our original definition.
- We look at the password process generation itself and break apart the password.
- For example: bananas2
- If we pretend that a computer pulled that out of all the printable ascii characters:
  - $H = \log_2(95^8) = 8 \times \log_2(95) = 8 \times 6.56 = \sim 52$ bits of entropy
  - According to RFC4086, you need $\sim 29$ bits of entropy. So not bad!
However look at it this way:
- It's a word prepended by a single digit number: "bananas" + "2"
- I'll give the benefit of the doubt and say that the word was found randomly out of the Oxford Dictionary (~600,000 words)
- There are 9 single digit numbers
- Thus \( H = \log_2(600000^1) + \log_2(9^1) = \sim 22 \) bits of entropy. Not good!
PASSWORD MANAGER

<I'm suppose to show you how to use keepass>
THANKS FOR COMING!