Password Based Encryption (RFC 2898)

Ryan Dearing
Passwords Are Everywhere!

• Computers
• Bank
• Websites
• ATM
• Phones
Password Weaknesses

• Small Space (ASCII)
• Poorly Chosen
• Reused
• Stored Insecurely
• Humans!
Password Attacks

• Dictionary
• Personal Info
• Rainbow Table
• Brute-Force
• Weak Storage (Reusing passwords)
Password Space
(US Keyboard)

• Single Case: 26
• Mixed Case: 52
• Mixed Alphanumeric: 62
• All characters on US Keyboard: 88
Password Strength
More Characters vs Longer

• Single Case, 8 characters: $26^8=208827064576$
• Mixed Case: $52^8=53459728531456$
• Single Case, 10 characters: $26^{10}=141167095653376$
• Use more characters, but be careful of dictionary attack!
Brute-Force/Rainbow Attack

• Computers can brute-force passwords very quickly!
• Rainbow attack even quicker (seconds)
• Solutions:
  – Add time complexity (Iterations)
  – Add salt (prevents Rainbow and Dictionary attacks)
Salt

• Hash function: \( h(v) \rightarrow R \)
• Just using hash function is bad because \( h(v) \rightarrow R \) for same \( v \)! Imagine \( n \) users with the same password.
• Instead, add a random and public salt:
  • \( h(v+\text{salt}) \rightarrow R1 \)
• Better, each user has different salt:
  • \( h(v+\text{salt2}) \rightarrow R2 \)
• Hard to build Rainbow Table!
Multiple Iterations

• Hash function: $h(v) \rightarrow R$

• Just using hash function is bad because $h(v_1) \ldots h(v_N)$ can be calculated quickly for brute-force.

• Instead, hash many times:
  $h(h(h(h\ldots h(v)))) \rightarrow R'$

• Better, Use salt too: $h(h(\ldots h(v+salt)+salt))$

• Takes much longer to brute-force.
Recommendations

• Salt: 64 bits, which is a string of length 10 if using 88 printable characters $88^{10}$
• Iterations: At least 1000
Creating Encryption Key From Password

- PBKDF2 (P, S, c, dkLen) → O
- P – password
- S – salt
- c – iteration count
- dkLen – length of key
- O – key of dkLen length
Creating Encryption Key From Password

• Password-Based Key Derivation Function
• PBKDF2 (P, S, c, dkLen) → O
• P – password
• S – salt
• c – iteration count
• dkLen – length of key
• O – key of dkLen length
Creating Encryption Key From Password

- \( l = \text{CEIL}(dk\text{Len} / h\text{Len}) \)
- \( r = dk\text{Len} - (l - 1) * h\text{Len} \)
- \( h\text{Len} \) is output length of hash function
- \( T1 = F(P, S, c, 1), T2 = F(P, S, c, 2), Tl = F(P, S, c, l) \)
- \( U_1 = h(P+S) \)
- \( U_2...U_c = h(P + U_{c-1}) \)
- \( F(P, S, c, i) = U_1 \oplus U_2 \oplus ... \oplus U_c \)
- \( \text{Key} = T1 || T2 || ... || Tl \) (concatenation)
JavaSE

• Don't worry, Java makes it easy!
• Java has built-in support for security and cryptography
• Supports many algorithms through Java Cryptography Architecture (JCA) and Java Cryptographic Extensions
Conclusions

• Java makes it easy to hash password and to use password based encryption.
• Store passwords hashed with salts and recompute the hash to verify users at login.
• Make sure to use PBKDF2 when created encryption keys from passwords.
• If designing requirements for passwords, remember that longer passwords are often more secure and easier to remember.
More Information and References

- http://seclists.org/basics/2008/Jul/207
Thank You!
Questions?