Password Privkey Pummeling Power – The Math and Words of Dictionary Cracking (Multi Code Companion)

Author	josh
Date	2024-07-14

Overview

There's a lot that goes into the cracking of passwords and private keys. These secrets defend our valued exchange accounts, cryptocurrency, and other assets. There's various attacks that hackers can use to crunch numbers and crack passwords or private keys. But there's not just brute force, as the math behind that means that most secrets can't be compromised in that way. Instead, dictionary-based attacks can be used to compromise weak secrets. This article will dive into the math behind brute-force cracking, why it's difficult to do so as the size of the secrets increase. Next, we'll discuss how dictionary attacks can be conducted against weaker passphrases and cryptocurrency keys. All these examples will be discussed with some fun and simple code written in several programming languages. Let's talk password and privkey pummeling!

The Math of Password and Key Cracking

Why Brute-Force Fails: The Exponential Math of Passwords and Keys

Brute-force is a term used when talking about cracking passwords or private keys. This type of attack means trying *every possible combination* for a given keyspace. For passwords, that means trying every possible character combination for a given password length – for example, all 8 character passwords consisting of uppercase and lowercase letters and numbers. For private keys, that means trying every possible combination of bits for a given keysize – for example all possible 20 bit keys. In other words, every possible number from 0 to $2^{20} – 1$.

This attack may seem simple and effective, and it is...to an extent. The problem with brute-force is that it quickly becomes impossible to conduct an attack due to the exponentially increasing number of total combinations. Let's first look at the math of cracking passwords with some simple code. This program called PassPerms has a Fortran and Python version, and shows how increasing password length effects brute-force combinations and crack times.

```
$ ./bin/passperms.exe --lengths --fast
 Combinations for constant complexity 62, variable length
 Cracking w/ 180 GH/s, MD5 - 25 GPU professional cluster
 Length           Combinations            Crack Time
    8.                  2.E+14             20.21668 minutes
    9.                  1.E+16             20.89057 hours
   10.                  8.E+17             53.96729 days
   12.                  3.E+21           567.96791 years
   14.                  1.E+25         2.18327E+06 years
   16.                  5.E+28         8.39248E+09 years
   20.                  7.E+35         1.24010E+17 years
```

```
      ! This subroutine calculates the results array for the given lengths
      ! It will calculate an entry for one sample-complexity to the length
power

      SUBROUTINE calc_lengths(results)
            ! Declare function params and returns, and data
            IMPLICIT none
            REAL(8), DIMENSION(ASIZE), INTENT(OUT)::results
            INTEGER::i

            ! Loop on each element and exponentiate; array shorthand does not
work in reverse

            DO i = 1, ASIZE, 1
                  results(i) = COMPLEXITY_SAMPLE ** LENGTHS(i)
            END DO

            RETURN

      END SUBROUTINE calc_lengths
```

```
# Get permutations based on lengths for a given complexity
# Returns a list of total password permutations

def compute_lengths(complexity):
    results = [ complexity ** length for length in LENGTHS ]
    return results
```

This code calculates the possible combinations of passwords given complexity (the character space) and length. This can be expressed as c^l, complexity to the length power. The longer our password is, the more possible combinations increase exponentially. Beyond about 10 characters, it quickly becomes untenable to brute-force alphanumeric passwords.

Private keys operate similarly in terms of exponential growth. My program PkTime, written in C, shows how the increasing number of bits in a cryptographic key (such as those used in Bitcoin, Ethereum, and other cryptocurrencies) exponentially increases the number of possible keys:

```
$ ./bin/pktime.exe 100000
Average optime for one keypair check: 0.00001000
Est. time to bruteforce 8 bit keysize: 0.00256000 seconds with 256 iterations
Est. time to bruteforce 12 bit keysize: 0.04096000 seconds with 4096
iterations
Est. time to bruteforce 16 bit keysize: 0.65536000 seconds with 65536
iterations
Est. time to bruteforce 18 bit keysize: 2.62144000 seconds with 262144
iterations
Est. time to bruteforce 20 bit keysize: 10.48576000 seconds with 1048576
iterations
Est. time to bruteforce 24 bit keysize: 2.79620267 minutes with 16777216
iterations
Est. time to bruteforce 28 bit keysize: 44.73924267 minutes with 268435456
iterations
Est. time to bruteforce 32 bit keysize: 11.93046471 hours with 4294967296
iterations
Est. time to bruteforce 36 bit keysize: 7.95364314 days with 68719476736
iterations
Est. time to bruteforce 40 bit keysize: 127.25829025 days with 1099511627776
iterations
Est. time to bruteforce 44 bit keysize: 5.57462736 years with 1.759219E+013
iterations
Est. time to bruteforce 48 bit keysize: 89.19403779 years with 2.814750E+014
iterations
Est. time to bruteforce 52 bit keysize: 1427.10460471 years with 4.503600E+015
iterations
Est. time to bruteforce 64 bit keysize: 5.845420E+006 years with 1.844674E+019
iterations
Est. time to bruteforce 128 bit keysize: 1.078290E+026 years with
3.402824E+038 iterations
Est. time to bruteforce 192 bit keysize: 1.989094E+045 years with
6.277102E+057 iterations
Est. time to bruteforce 256 bit keysize: 3.669230E+064 years with
1.157921E+077 iterations

* It iterates over an array of keysizes, and provides estimated CPU time and
number of iterations
* needed to brute-force the keyspace. Data is returned back in the array
arguments for times and iterations.
*/
void calc_esttimes(const short keysizes[], double times[], double
iterations[], const int length, double avg_time)
{
    for (int i = 0; i < length; i++)
    {
        iterations[i] = pow_two_double(keysizes[i]);
        times[i] = time_bruteforce_est(iterations[i], avg_time);
    }
}
```

This program iterates over given keysizes and calculates the total possible keys as 2^{l}, with l being the key length. Private keys don't use alphanumeric characters, but instead bits – which is why our base is always 2 in this case.

Again, above a relatively small keysize, brute-force becomes impractical or impossible. Above 40 or so bits. Most cryptographic keys nowadays are 256 bits (the keysize used in Bitcoin), which would take several universe lifetimes to break!

Alternative Attacks

Dictionary Based Attacks on Passwords and Keys

If brute-forcing is impractical or impossible for most passwords and keys, then how do attackers often crack longer passwords or even Bitcoin keys? The answer lies in *entropy*, the overall randomness used to generate the passphrase or key. Most people don't choose passwords that are truly random, generated from a secure source. Instead, they use common words in combination to create a password. Bitcoin keys are usually generated at random, but there's a type of user-generated key called a brainwallet. For these, a user chooses a passphrase and hashes that into a 256 bit number used as the key (usually via SHA-256). This means that the wallet essentially is backed by an alphanumeric passphrase.

Both of these types of passphrases can be cracked using what's called a *dictionary attack*, where the cracker tries combinations of words, common leaked passwords, and variations on those to crack password hashes or key hashes (in the case of Bitcoin addresses). The attacker either has a password hash from a database leak, or an address on the blockchain they may suspect belongs to a brainwallet. In both cases, the attacker does not try *all* possible combinations, but rather *likely* combinations using words.

My project DictionaryDriller, written in Rust, shows what a simple dictionary attack looks like on a password hash.

```
$ dictionarydriller.exe wordlist_sample.txt
$argon2i$v=19$m=16,t=2,p=1$Zm9vYmFyYmF6$YMoljsb76DqkMZExF1AZZg
Hash: $argon2i$v=19$m=16,t=2,p=1$Zm9vYmFyYmF6$YMoljsb76DqkMZExF1AZZg
Salt: foobarbaz
Running attack
Attack finished
Password found: rockyou
```

```
/* This function conducts the dictionary attack */
pub fn attack(argon2_hash: argon2::PasswordHash, wordlist_filename: &String) -
> (bool, String)
      {
            let mut found = false;
            let mut found_password = String::from("");
            for word_line in
std::fs::read_to_string(wordlist_filename).unwrap().lines()
            {
                  // Get the word from file
                  let word = word_line.to_string();

                  // Use the argon2 libraries verify_password functionality to
check
                  // the word against the supplied password hash
```

```
                  // Print information about the match if found
                  let hash_match =
argon2::Argon2::default().verify_password(word.as_bytes(),
&argon2_hash).is_ok();
                  if hash_match
                  {
                        found_password = word;
                        found = true;
                  }
            }

            return (found, found_password);
      }
```

This code takes a wordlist (for example a list of previously breached passwords), and hashes each possible word using the password hashing algorithm Argon2. Given the target password hash, it then compares the hash for each attempt against that target. If there's a match, that means our password is found, because a cryptographic hash always gives the same hash output for a given input.

For Bitcoin brainwallets, I wrote a project called BrainBrawler with both a C++ and Python version. This code conducts a dictionary attack on a Bitcoin address instead of a password.

```
$ ./bin/brainbrawler.exe 167pV6UnXfYe6DFyxubM45rWVeW1uA6bF7
Password found: rockyou
```

```
// Conduct a dictionary attack on possible brainwallets
AttackResult attack(std::string address)
{
      AttackResult result;

      std::string word;
      std::ifstream wordlist_file(WORDLIST_FILENAME);
      while (std::getline(wordlist_file, word))
      {
            std::string candidate_address = generate_brainwallet(word);

            if (address == candidate_address)
            {
                  result.found = true;
                  result.passphrase = word;
                  result.address = candidate_address;

                  return result;
            }
      }

      result.found = false;
      result.passphrase = "";
      result.address = "";

      return result;
```

```
}

# Run a dictionary attack on possible brainwallets
def attack(address):

    bitaddr = bitaddrlib.BitAddr()

    with open(WORDLIST_FILENAME) as f:
        for word in f:
            word = word.strip()
            candidate_address = bitaddr.generate_brainwallet(word)
            if candidate_address == address:
                result = AttackResult(True, word, candidate_address)

                return result

        result = AttackResult(False, "", "")

        return result
```

This code takes a wordlist and generates a Bitcoin address using each possible word. It first hashes the word using SHA-256, then uses that as a Bitcoin private key. The private key is used to derive the final address. For each generated address, it is compared to the target address. If they match, then the brainwallet password has been found.

Big ‘Ol Brute-Force Math and Dictionary Cracking for Passwords & Privkeys

These code projects demonstrate the interesting math of password and keysizes, and how larger passphrases and keys can quickly thwart brute-force attacks. As the secret gets longer, the more the total possible combinations exponentially increase! Instead, we can use dictionary attacks to target poor-entropy passphrases and keys that use common words and phrases for generation. Most users generate passphrases from words, and some Bitcoin users try to use such memorable phrases to derive their keys. In both cases, a dictionary attack can crack that – leading to account compromise or the theft of funds. In all cases, we can explore this interesting math and possible attacks with some fun code!