

## How it works

- As you learned in activities four and five, a **cryptographic hash** function is a one-way calculation; the original password can't be reverse-calculated from its hash. When a hacker breaks into a computer and steals the password hashes, they can't use them to log in, but they could be used in a **rainbow table** or a **brute-force** attack.
- A brute-force attack generates all possible **permutations** of cleartext passwords, calculates their hashes, and checks them against a stolen hash. It is a prolonged process requiring much **computation time**.
- One reason this attack requires so much computation time is the number of permutations for a given **password rule**, which requires the password to be composed of a certain number and type of characters.
- The permutations of a particular password rule represent all the ways the password could be formed based on the number and types of characters required.
- For example, a password rule with exactly four characters, using only uppercase letters, has 1 of the 26 characters at each position. The number of ways a password could be formed based on this rule is  $26 \times 26 \times 26 \times 26 = 456,976$ , or  $26^4$  different password permutations!
- The above example can be generalized as:

$$\text{permutations} = \text{number of possible characters}^{\text{length of password}}$$

- Rules that mandate long passwords comprising a wide range of possible characters present a significant hurdle for brute-force attacks. The more complex the password, the more difficult it is to crack. For instance, a rule requiring eight characters, including a lowercase letter, an uppercase letter, a number, and a special character (32 in total), results in a staggering 6,095,689,385,410,816 possible passwords! This complexity empowers you to enhance your cybersecurity practices.

## What will you do?

1. Practice with password strength (permutations):
  - a. Open 'prac\_1\_6.py', in the editor and run the code. Press the [var] key and select "p(chrs,len)" from the menu. This function takes the number of different characters and the length of the password rule and calculates the permutations. Test the example in the introduction, p(26+26+10+32, 8). Try calculating the permutations of different password rules.
  - b. Open 'prac\_2\_6.py' in the editor and run the code. This program creates a plot of password length on the horizontal axis and the number of permutations on the vertical axis. Edit the program to change the pswd\_chrs and pswd\_len. What does the shape of the curve imply about complex passwords?
  - c. Open 'prac\_3\_6.py', in the editor and review the Python code. Do you see how three nested loops generate every possible password based on the password rule requiring three uppercase characters? Run the program and observe how it works. Do you see how the three nested loops create the permutations? How many possible passwords can be made from three uppercase letters? **Note: This program takes about ten minutes to complete!** If you become impatient, press and hold the [on] key to interrupt the program.
  - d. Return to program 'prac\_1\_6.py' and calculate the permutations for the brute-force program you just ran in 1.c. How many different passwords were created? How long did it take on the calculator? Can you imagine how long it would take to calculate 6,095,689,385,410,816 permutations?

2. Set a password on your micro:bit.
  - a. Connect your micro:bit to your calculator.
  - b. Open "set\_pw\_6.py" in the editor and run. Create a password using the rule: three characters containing only lowercase letters. *Note:* To save runtime later in the 'prac\_4\_6.py' activity, **choose a password with a first letter near the beginning of the alphabet**. For example, "aab" or "aba" will be cracked MUCH sooner than "zaa" or "zba". Can you look at the Python code in 'brute\_6.py' and explain why? *Note:* if your calculator takes too long to finish, press and hold the [on] key to interrupt the program.
  
3. The brute-force attack:
  - a. Exchange your micro:bit with a group member. *Do not tell them your password!*
  - b. Open "brute\_6.py" in the editor and review the program. Notice there are no print statements within the three nested loops. Output of any kind, such as printing to the screen, is relatively slow. Eliminating print statements makes the program much faster. **Optimizing** extensively repeated code is essential when writing programs.
  - c. Open "prac\_4\_6.py" in the editor and run. The hash of the password is retrieved from the file "password.txt" on the micro:bit and displayed. This is the file and hash a hacker will try to steal to compromise the security of your micro:bit! Run the program and notice the number of attempts and the elapsed computation time. *Note:* the hash can't be reversed; however, the brute-force attack is a workaround that finds the plaintext password of the micro:bit's hashed password.
  
4. Authentication:
  - a. Open "authen\_6.py" in the editor and review the program. Run the program using the cracked plaintext password from 3.c to test the result of the brute-force attack. Did you hack it?
  - b. Rerun the program and enter the wrong password three times. After examining the Python code, can you explain how it prevents repeated authentication attempts? How does limiting attempts enhance security?
  
5. Remote login:
  - Ensure all group members use the same assigned group number.
  - The **receiver**
    - a. Change the password as outlined in activity 2. Practice good **password hygiene** and do not reuse the same password. Whisper your password to the sender so they can log in to your micro:bit. Be sure to *keep the password private* from the hacker. Open "recv\_6.py" in the editor and review the program. Run the program *before* the sender has run theirs.
  - The **sender**
    - a. Open "send\_6.py" in the editor and review the program. Run your program *after* the receiver and hacker have started theirs. Send the receiver's password to log in to their micro:bit remotely.
  - The **hacker**
    - a. Open 'hack\_6.py' in the editor and review the program. Run your program *before* the sender has run theirs. *Note – this program will do a man-in-the-middle attack on the sender transmission and steal the receiver's password hash.*

- b. Use the stolen hash and the imported brute\_6 module to crack the receiver's password from the intercepted hash. Use the module's "brute(stolen\_hash)" function to crack the hash. Type >>>brute(stolen\_hash). Remember, the variable "stolen\_hash" can be selected from the [var] key.
- Once the hacker has cracked the receiver's password, the receiver should run the 'student\_receiver.py' program again to test whether the hacker can break into their micro:bit. The hacker should open to the 'send\_6.py' and send the cracked cleartext password. Note – if the hacker is successful, they should be able to log in to the receiver's micro:bit without ever being told the password!

## Code it

### Sender role

```
EDITOR: SEND_6
PROGRAM LINE 0001
from microbit import *
from mb_radio import *
from hashing import *
from mb_disp import *
from mb_music import *

radio.on()
radio.config(length=250, channel
            =12,power=6,group=1)
disp_clr()
pswd = input("Enter password: ")
Fns... [a A #] Tools Run Files
```

### Receiver role

```
EDITOR: RECV_6
PROGRAM LINE 0001
from microbit import *
from mb_radio import *
from mb_file import *
from mb_disp import *
from mb_music import *

radio.on()
radio.config(length=250, channel
            =12,power=6,group=1)
disp_clr()
print("Waiting for password...")
Fns... [a A #] Tools Run Files
```

### Hacker role

```
EDITOR: HACK_6
PROGRAM LINE 0001
from microbit import *
from mb_radio import *
from hashing import *
from brute_6 import *

_chr_set_26 = "abcdefghijklmnopqrstuvwxyz"

stolen_hash=""
radio.on()
radio.config(length=250, channel
            =12,power=6,group=1)
Fns... [a A #] Tools Run Files
```

## Go further

- Each team member should play each role and try to hack the other's password.
- Set the micro:bit password to "aaa" and run "prac\_4\_6.py" and record how long it takes to crack the hash. Repeat the process for the password "aaf", "aak", "aap". Can you predict how long it would take to crack "aaz"? Looking at the code, can you explain this trend?

## Check your understanding

- A brute-force attack is a computation-intensive hack that computes the hash of all permutations of passwords for a given password rule and compares them with a stolen password hash. If the hashes are the same, the cleartext password for the stolen hash is discovered.
- Frequently changing your password, incorporating many infrequently used characters, not reusing the same password on different accounts, and not sharing or writing down passwords help protect your accounts from hackers.

## Help

- Check that everyone on the team is using their assigned group number.
- Ensure the receiver and hacker run their programs and wait before the sender transmits the message.
- Remember, the sender is trying to log in to the receiver's micro:bit and must send the password for the receiver's micro:bit.

### Files

- Transfer the activity files below to your calculator using the TI Connect CE Software. The link to download is [here](#). The best practice is to load all files for this cybersecurity activity and then delete them before loading the next set of activity files. This helps keep your calculator organized.

Name	Description
prac_1_6.py	Calculate the number of permutations of a given password rule.
prac_2_6.py	Plot a graph of the number of permutations as a function of password length.
prac_3_6.py	Displays all permutations of 3 lowercase letter passwords.
Prac_4_6.py	Imports brute_6.py to brute-force crack the password stored on the micro:bit.
set_pw_6.py	It prompts for a three-lowercase character password and saves the hash in the 'password.txt' file on the micro:bit.
authen_6.py	Compares the hash of the entered password with the hash stored on the micro:bit.
send_6.py	Sends hashed password to receiver for authentication.
recv_6.py	Receives hashed password and authenticates.
hack_6.py	A man-in-the-middle attack snatches the hash from the sender and imports "brute_6.py" to crack the stolen password hash.
brute_6.py	The program to brute-force crack a hash. It calculates all permutations of plaintext passwords, computes the hash, and tests it with the stolen hash to reveal the plaintext password.
HASHING.8XV	SHA-256 hashing module