

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 creates all **permutations** of possible **cleartext** passwords, calculates the hash, and checks it against the stolen hash. A brute force attack is a prolonged process requiring much **computation time**.
- One reason this attack requires so much computation time is the number of permutations for a **password rule**.
- The permutations of a password rule represent all the ways the password could be formed based on the number and types of characters required by the rule.
- For example, a password rule with exactly four characters, using only uppercase letters, has 1 of 26 different 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:

$$\text{possible characters}^{\text{length of password}}$$

- Rules requiring long passwords formed from many possible characters are more challenging to brute-force attack simply because there are many more possible passwords to test. For example, a rule requiring eight characters composed of a lowercase letter, an uppercase letter, a number, and a special character (there are 32) has $(26 + 26 + 10 + 32)^8 = 6,095,689,385,410,816$ possible passwords!
- Long passwords composed of many types of characters that are changed frequently reduce your chances of suffering a brute-force attack.

What will you do?

1. Practice with password strength (permutations):
 - a. Advance to page 1.3, "possible_passwords.py," 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.
 - b. Advance to the calculator page and review the example calculation. Do you see the password rule requiring three uppercase characters yields 17,576 possible passwords? Did this calculation yield the number of passwords the previous Python program generated?
 - c. Advance to the next calculator page with a problem prompt and complete the practice calculation. Place the cursor below the problem prompt, and use the calculator keys to enter the calculation. Did you get 3,226,266,762,397,899,821,056? Can you think of how you would modify the nested loops in the Python program to generate every possible password based on the new rule? Can you imagine how long it would take to run the program?
 - d. Advance to the following graph page. Use the trace function to explore the graph of the number of password permutations (y) as a function of password length (x). What do you notice

TI-Nspire CX II

about the shape of this curve? Since many permutations require more computation time, what does the graph imply about password length and vulnerability to brute-force attacks?

2. Set a password on your micro:bit.
 - a. Connect your micro:bit to your calculator.
 - b. Advance to page 2.1 and run the program. Create a password using the rule: four characters containing only letters. *Note:* To save runtime later in the “brute_force.py” activity, **choose a password with a first letter near the beginning of the alphabet**. For example, “Abba” or “Abby” will be cracked MUCH sooner than “Zeno” or “Zola.” Also, “ABBA” is faster than “abba”. Can you look at the Python code 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. Advance to page 3.1 and run the program. The hash of the password is retrieved from the file “password.txt” and displayed. This is the file and hash a hacker will try to steal to compromise the security of your micro:bit! *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.
 - c. After running the program in the Python shell on page 3.2, press the [var] key. Do you notice the variable named “hacked_hash”? Select the variable from the menu and see it returns the 256-bit hash of your password to the **Python REPL prompt >>> in the shell**. *Note:* you will use this variable in step 5, the remote login activity.
 - d. Advance to the next page with “brute_force.py” and review the program. Notice there are no print statements within the four 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. Run the program and notice the number of iterations and elapsed time. Compare your elapsed time with others within your group. The program will return the plaintext password with a hash that matches the hacked hash. Remember the plaintext password to test in the next activity.
4. Authentication:
 - a. Advance to page 4.1, “authentication.py” and run the program. 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?
 - b. Rerun the program using the cracked plaintext password to test the result of the brute-force attack. Did you hack it?
5. Remote login:
 - The **receiver**
 - Change the password as outlined in activity 2. Practice good **password hygiene** and do not reuse the same password. Select a password toward the beginning of the alphabet. Whisper your password to the sender so the sender can log in to your micro:bit. Be sure to *keep the password private* from hackers. Advance to ‘student_receiver.py,’ change the group to their assigned number, and run the program *before* the sender has run theirs.
 - The **sender**
 - Advance to ‘student_sender.py,’ change the group to their assigned number, and run your program *after* the receiver and hacker have started theirs. Send the receiver’s password to log in to their micro:bit remotely.

- advance to the 'student_hacker.py,' change the group to their assigned number, and then run the program *before* the sender has run theirs. *Note* – this program *will receive the password hash* sent by the sender to log in to the receiver's micro:bit.
- Use the stolen hash and the brute_force_cracking module to crack the receiver's password from the intercepted hash. Use the module's "brute(hacked_hash)" function to hack the hash. Type >>>brute(hacked_hash). Remember, the variable "hacked_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 your micro:bit. The hacker should advance to the 'student_sender.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

```

5.1 5.2 5.3 6 - Cyber...ker RAD X
student_sender.py 1/11
from microbit_radio import *
from hashing import *
# The sender must use the password of the
# receiver's micro:bit.
channel = 1
group = 1
clear_history()
password = input("Enter password: ")
password_hash = sha_hash(password)
tx(password_hash,channel,group)

```

Receiver role

```

5.1 5.2 5.3 6 - Cyber...ker RAD X
student_receiver.py 1/15
from microbit_radio import *
from hashing import *
# share your password with the sender
channel = 1
group = 1
clear_history()
test_hash = rx(channel,group)
authentic_hash = read_file("password.txt")
if test_hash == authentic_hash:
    display.show(Image.YES)
    music.play(music.BA_DING)

```

Hacker role

```

5.2 5.3 5.4 *6 - Cybe...ker RAD X
student_hacker.py saved successfully
from microbit_radio import *
from brute_force_cracking import *
# The hacker will use brute force to find
# the password that was sent to the receiver.
channel = 1
group = 1
clear_history()
hacked_hash = rx(channel,group)
print("hash string = {}".format(hacked_hash))
# On the next page type:
# brute(hacked_hash)

```

Go further

- Each team member should play each role and try to hack the other's password.
- Record the time required to crack a password starting with the letter "A", repeat for a password starting with "B". Continue for the letters through "E". Make a graph with letter position, e.g., A = 1, B, etc., on the horizontal axis and cracking time on the vertical axis. Can you predict the shape of the curve of this graph?

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.