PASSWORD CRACKING

1) One Unix password is an eleven character long hexadecimal number (with no leading 0x and all letters in upper case).

To try and crack this password we started with a forward search using the command: *nice -19 ./john --session=forward_orig --mask=?H?H?H?H?H?H?H?H?H?H?H?H fifth_hash.txt.* The forward search by itself will search through 16^11 combinations at worst and (16^11) / 2 combinations in expectation.

Later, we added a backward search with the command : *nice -19 ./john --session=first_back --mask=?1?1?1?1?1?1?1?1?1?1?1 -1=[F-A9-0] fifth_hash.txt.* The forward and backward search combined will search through (16^11) / 4 combinations in expectation.

As the search size for this password is extremely large, we were not able to crack the password. Being an eleven-character-long hexadecimal password in uppercase, still involves 16^11 possible combinations, making brute force computationally expensive. While combining forward and backward search reduces the expected search space, it remains infeasible without further hints.

2) One Unix password is a twenty character long random ASCII string (using any printable ASCII character).

To try and crack this password we used an exhaustive search with the command: *nice -19 ./john --session=ascii_orig --mask=?a --min-length=20 --max-length=20 first_hash.txt.* This will search through 95^20 combinations at worst and (95^20) / 2 combinations in expectation

Similar to hash one, the search size for this password was also extremely large. A twenty-character-long random ASCII string is even more impractical to crack, as it spans 95^20 possible combinations. Without additional constraints, such as known character patterns or partial leaks, cracking these hashes purely through brute force is computationally infeasible and would take an impractical amount of time.

3) One Unix password is the thirteen character long leet version of a thirteen character long English word.

For this password, we first got a list of all english words from a public source file on GitHub: words.txt. Using a Python script (filter.py), we filtered out only the words that were exactly 13 letters long. The output was saved in words13.txt.

Next, we wrote a Python program (leet.py) to generate all possible leet-style transformations for each 13-letter word. This program created a set of rules in the format used by John the Ripper. To run the script and save the rules, we used the following command: python3 leet.py > leetRules.txt The generated rules were then saved to leetRule.txt The text file is also provided.

Then, we copied the content of the file to the john.config file and used this command in John the ripper: *nice -19 ./john --wordlist=words13.txt --rules=Leet --session=leetU hash3.txt*

Please Note: With this process, we were able to crack 16/17 of all group leet hashes, only excluding our own. As a result, we believe there may be an error in the hash we were given. Here is a list of the other leet passcodes we were able to crack. The file with all those hashes and their equivalent passwords is also provided names: cracked_leet_hashes.txt

3x4gg3r4tions
4ccl1m4t1z1ng
met411urgic41
[ompr3h3nsiv3
[0ntaminati0n
b3n3f1c1ar13s
313ctromagn3t
c0ns3qu3ntial
r3v0luti0nary
di5agr33m3nt5
ga5tro3nt3ric
m1sunderst00d
h0rticu1tura1
4rch3ologic4l
C0un7erac7ive
1n7rospec71on

4) One Unix password is a thirteen character long English word starting with an uppercase letter (the rest are lower case)

To crack this password, we first got a list of all english words from a public source file on GitHub: words.txt. Then, in John the Ripper, we added this custom rule to the john.conf file, which capitalizes the first letter of each word in the wordlist:

```
[List.Rules:CapitalizeFirst]
c
```

Then, the password was cracked using the following command: *nice -19 ./john --wordlist=words.txt --rules=CapitalizeFirst all_hashes.txt*

The cracked password was "Anticipations", and the process took less than 10 minutes.

5) One Unix password is a mangled name of a Hockey Player.

To crack this password, we got the list of all NHL players from a publicly available Google Sheet. We found this at: https://github.com/Zmalski/NHL-API-Reference. Then, a python script (hockey.py) was used to extract all player names and save them to hockey_players.txt. The script can be run with: *python3 hockey.py*.

Initially, we used the following command to attempt cracking the hash: *nice -19 ./john --wordlist=mangled_names.txt*. The password took less than a minute to crack. However, later, we added the following rule to john.conf:

```
[List.Rules:Mangle]
T0 /  ip\- @  /- vj01vjpj Tj /- Dp Ap"[\-_~]"
T0 /  ip\- @  /- vj01vjpj /- Dp Ap"[\-_~]"
/  ip\- @  /- vj01vjpj /- Dp Ap"[\-_~]"
/  ip\- @  /- vj01vjpj Tj /- Dp Ap"[\-_~]"
```

We used John the Ripper and the above rule to crack the password again with the command : nice -19 ./john --wordlist=hockey_names.txt --rules=Mangle fourth_hash.txt.

The cracked password was “nathan~macKinnon” and the process took ~ one minute.

RANK OF PASSWORDS

(From easiest to hardest based on time taken to crack and counting runtime of the commands)

1. Number 4 (<10 minutes)
2. Number 5 (10-20 seconds)
3. Number 3 (N/A): For other groups' passwords, 5-10 minutes for one password cracking.
4. Number 1 (N/A)
5. Number 2 (N/A)

LAMPORT ONE TIME SIGNATURE

Lots_genkeys:

- How to run: python3 lots_genkeys.py
- The keys are stored as bytes.
    - For private-keys:
      Every even-indexed entry (i % 2 == 0) is X_i.
      Every odd-indexed entry (i % 2 == 1) is Y_i.
      Total: 512 pairs → (X_1, Y_1), (X_2, Y_2), …
    - public_key.lots
      Every even-indexed entry (i % 2 == 0) is A_i = H(X_i).
      Every odd-indexed entry (i % 2 == 1) is B_i = H(Y_i).
      Total: 512 pairs → (A_1, B_1), (A_2, B_2), …
      *Assuming the index starts at 0

1) How did you validate your code? That is, how did you test that it worked correctly? List any and all tests you performed. Make sure you include test cases you created to test the ability of the verifier to come up with an INVALID response.

Test to validate lot_genkeys size of output files is correct :

2 files: private_key.lots and public_key.lots are created in the current directory

Checking they are created properly: ls -l private_key.lots public_key.lots
Shows if the files are created and This should also verify the right number of pairs generated because the size of the files should be 65536 (512 *64*2) 512 different pairs each 64 bytes

The program checks to see if the same random pair is generated twice.

Test for ensuring that lots_genkeys produces a new, unique public/private key pair every time :

The program was executed three times, each time saving the files under different names with the following sequence of commands

```
./lots_genkeys
mv private_key.lots private_key1.lots
mv public_key.lots public_key1.lots

./lots_genkeys
mv private_key.lots private_key2.lots
```

```
mv public_key.lots public_key2.lots

./lots_genkeys.py
mv private_key.lots private_key3.lots
mv public_key.lots public_key3.lots
```

The difference was checked with the diff command:

```
diff private_key1.lots private_key2.lots
diff private_key2.lots private_key3.lots
diff public_key1.lots public_key2.lots
diff public_key2.lots public_key3.lots
```

They should all differ.

Test for testcases-lamport-ots-UDP dataset :

testcase1: Use `pubkey-test-1.lots` to verify signature `sig-1.lots` for the message `m1.text`.

```
./lots_verify m1.text pubkey-test-1.lots sig-1.lots
VALID
```

testcase2: Use `privatekey-test-2.lots` and `pubkey-test-2.lots` to sign the message you produce and verify.

```
echo "my message" > message.txt

./lots_sign message.txt privatekey-test-2.lots
success

./lots_verify message.txt pubkey-test-2.lots signature.lots
VALID
```

testcase3:

```
./lots_verify m3.text pubkey-test-3-correct.lots sig-3.lots
VALID

./lots_verify m3.text pubkey-test-3-wrong.lots sig-3.lots
INVALID
```

2) How did you ensure that each invocation of lots_genkeys produces a new, unique public/private key pair? List all inputs you combined to ensure this uniqueness.

A combination of large enough key length and element size ensures that each invocation of lots_genkeys.py produces a new, unique key pair without using persistent memory to keep

track of previous keys. Key length, in this case, represents the number of pairs in our private key which is 512. Key element size refers to the number of bits of an element of our private key which is also 512. There are 2^(512*2+512) = 2^(3*512) private keys possible. While we do not have 100% guarantee that lots_genkeys.py will not produce duplicate keys, the probability of such an event is astronomically low.

3) The LOTS is used for a single signature, i.e. signature of a single message. While practically speaking this is true, theoretically, we could sign one more message (for a total of two) using the same public/private key pair. What is the relation of those two messages? Why is such a scenario unlikely?

The second message would be the message where the hash is the bitwise XOR of the hash of the first message and the 512 bits of all 1's. In other words, the bits of the hash of the first message are flipped. The bad guys have a very low chance of correctly signing such a message because we have not yet exposed that part of the private key yet. However, a scenario where we sign such a message is unlikely because that second message is highly likely to be just a meaningless combination of characters. Furthermore, we now expose all parts of our private key lowering the scheme's security level.