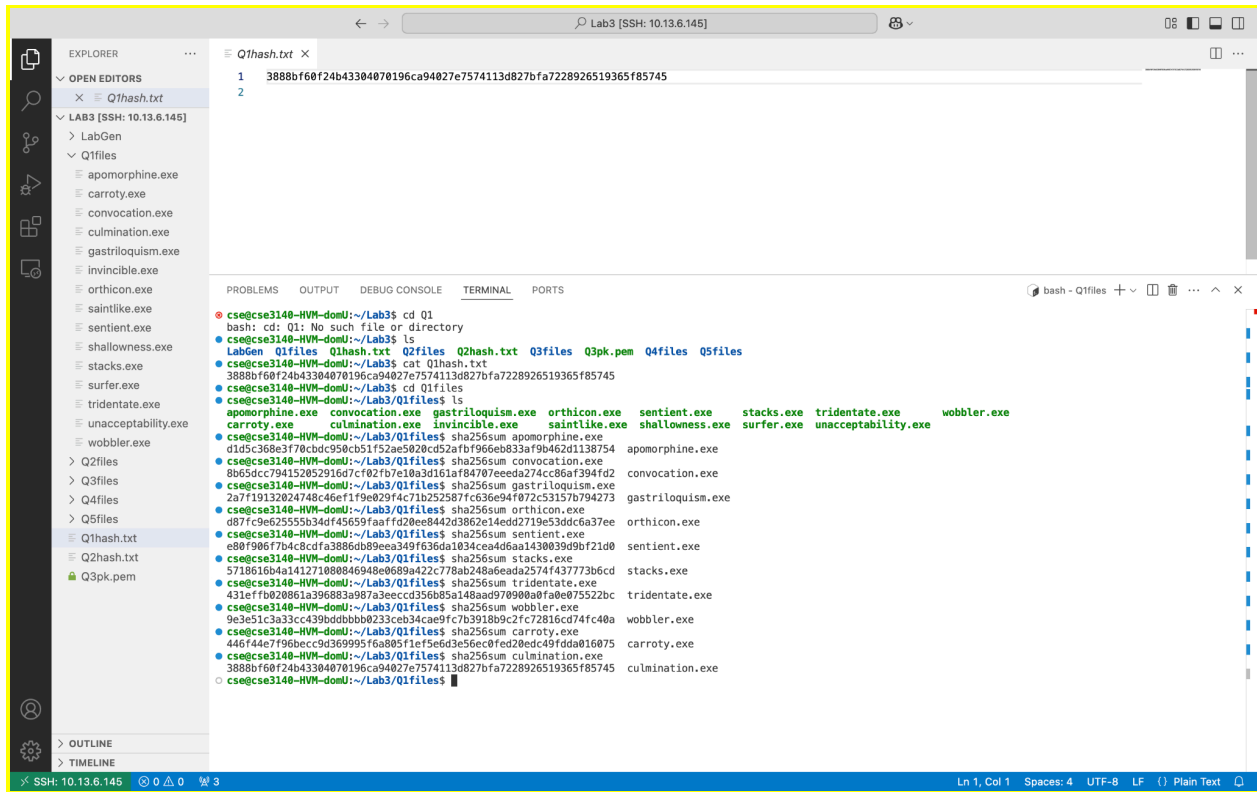


Layla Nassar
Section 003
CSE 3140
IP Address: 10.13.6.145
NetID: LTN22001
March 11, 2025

CSE 3140 Lab 3 Report: Cryptography, Malware, and Ransomware

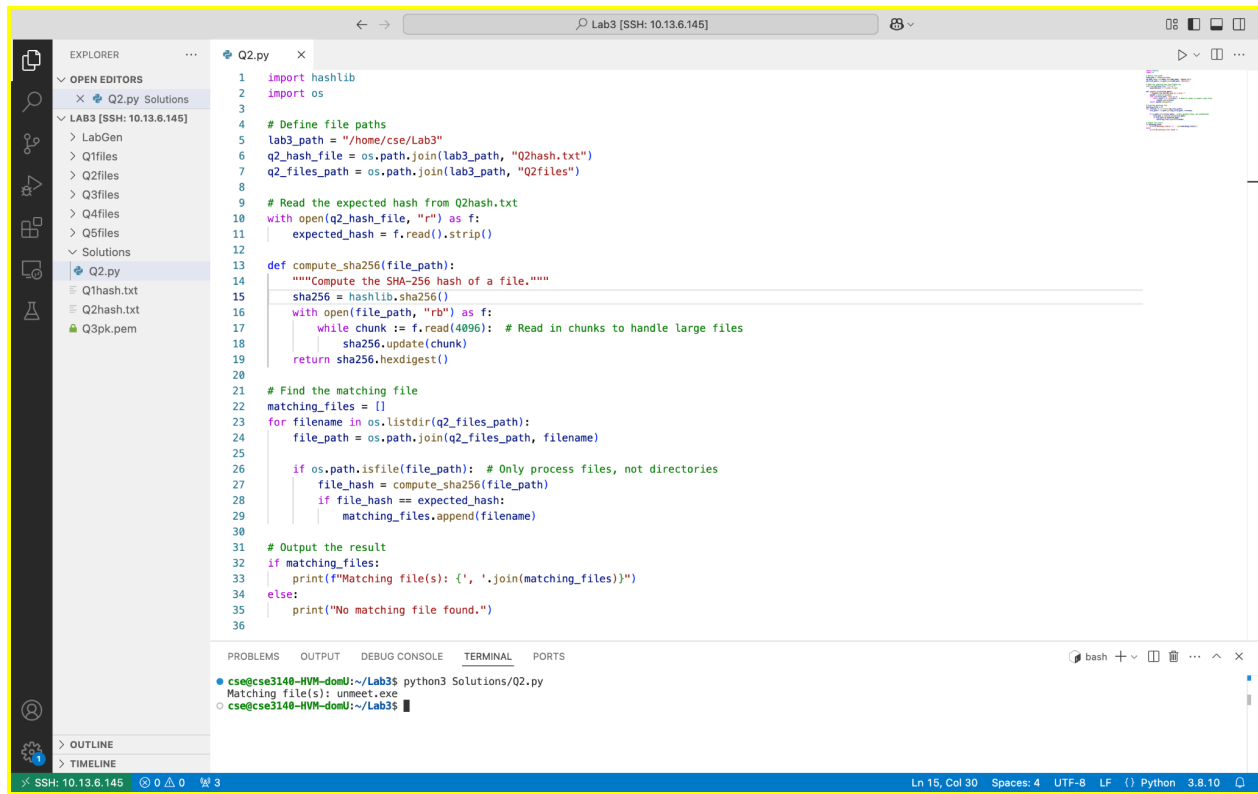
Question #1: culmination.exe



The screenshot shows a VS Code editor interface with a terminal window open. The terminal displays the following commands and output:

```
bash: cd: Q1: No such file or directory
cse@cse3140-HVM-domu:~/Lab3$ ls
LabGen  Q1files  Q1hash.txt  Q2files  Q2hash.txt  Q3files  Q3pk.pem  Q4files  Q5files
cse@cse3140-HVM-domu:~/Lab3$ cat Q1hash.txt
3888bf60f24b43304070196ca94027e7574113d827bfa7228926519365f85745
cse@cse3140-HVM-domu:~/Lab3$ cd Q1files
cse@cse3140-HVM-domu:~/Lab3/Q1files$ ls
apomorphine.exe  convocation.exe  gastriloquism.exe  orthicon.exe  sentient.exe  stacks.exe  tridentate.exe  wobbler.exe
carroty.exe      culmination.exe  invincible.exe     saintlike.exe  shallowness.exe  surfer.exe  unacceptability.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum apomorphine.exe
d1d5c368e3f70cbdc950cb51f52ae5020d52afbf966eb833af9b462d1138754  apomorphine.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum convocation.exe
8b65dccc794152052916d7cf02fb7e10a3d161af84707eeda274cc86af394fd2  convocation.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum gastriloquism.exe
2a7f19132024748c46ef1f9e029f4c71b252587fc636e94f072c53157b794273  gastriloquism.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum orthicon.exe
d07fc9e62555b34df45659faaf7d20ee8442d3862e14edd2719e53ddc8a37ee  orthicon.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum sentient.exe
e80f906f7b4c8cdfa3886db89eca349f636da1034cea4d6aa1430039d9bf21d0  sentient.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum stacks.exe
5718616b4a1412718080846948e8689a422c778ab248a6eada2574f437773b6cd  stacks.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum tridentate.exe
431effb020861a396883a987a3eeccdd356b85a148aad970900a0fa0e075522bc  tridentate.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum wobbler.exe
9e3e51c3a33cc439b0dbbb0233ceb34cae9fc7b3918b9c2f72816cd74fc40a  wobbler.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum carroty.exe
446f44e7f96bec95d369995f6a805f1ef56d3e56ec0fed20edc49fada816075  carroty.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$ sha256sum culmination.exe
3888bf60f24b43304070196ca94027e7574113d827bfa7228926519365f85745  culmination.exe
cse@cse3140-HVM-domu:~/Lab3/Q1files$
```

Question #2: unmeet.exe



The screenshot shows a VS Code editor window with a file explorer on the left and a terminal at the bottom. The file explorer shows a project structure with folders like LabGen, Q1files, Q2files, Q3files, Q4files, Q5files, and a Solutions folder containing Q2.py, Q1hash.txt, Q2hash.txt, and Q3pk.pem. The main editor displays the code for Q2.py, which is a Python script that reads a hash from Q2hash.txt, computes SHA-256 hashes for files in Q2files, and prints the matching file(s). The terminal shows the command `python3 Solutions/Q2.py` being executed, resulting in the output `Matching file(s): unmeet.exe`.

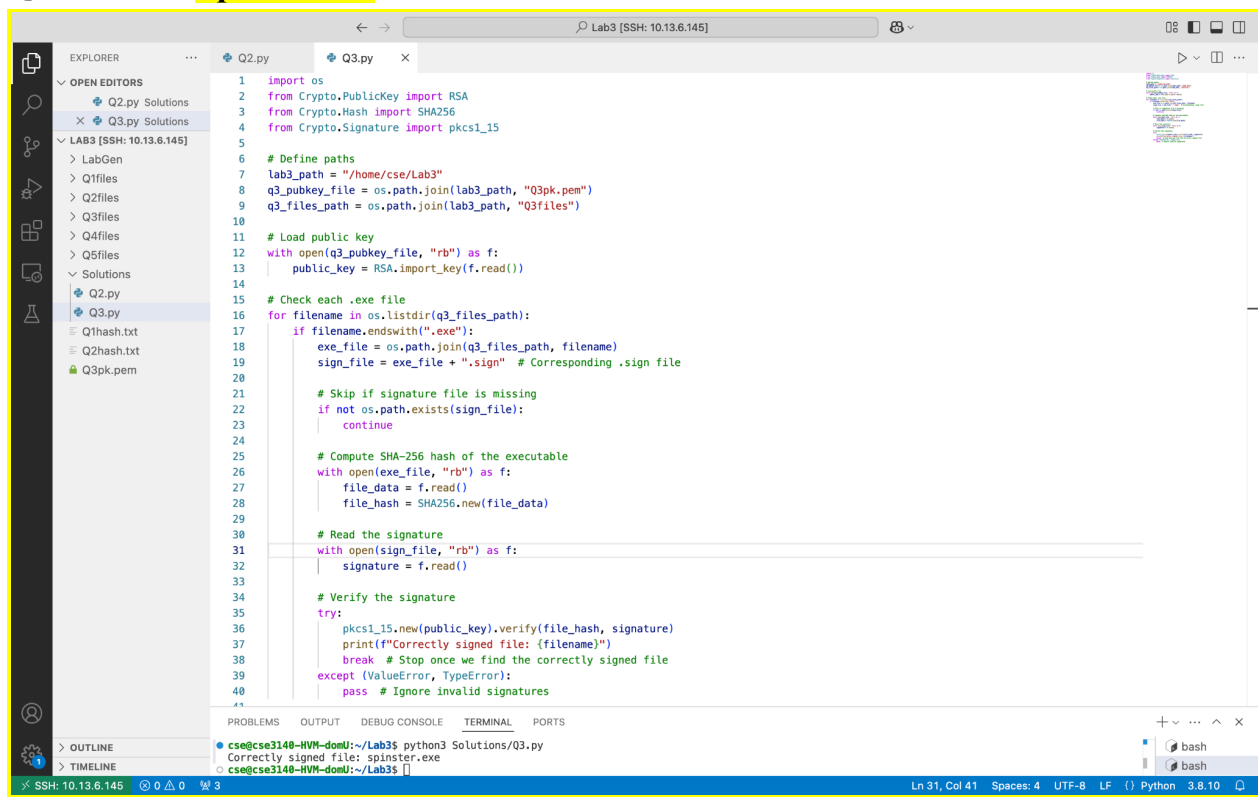
```
1 import hashlib
2 import os
3
4 # Define file paths
5 lab3_path = "/home/cse/Lab3"
6 q2_hash_file = os.path.join(lab3_path, "Q2hash.txt")
7 q2_files_path = os.path.join(lab3_path, "Q2files")
8
9 # Read the expected hash from Q2hash.txt
10 with open(q2_hash_file, "r") as f:
11     expected_hash = f.read().strip()
12
13 def compute_sha256(file_path):
14     """Compute the SHA-256 hash of a file."""
15     sha256 = hashlib.sha256()
16     with open(file_path, "rb") as f:
17         while chunk := f.read(4096): # Read in chunks to handle large files
18             sha256.update(chunk)
19     return sha256.hexdigest()
20
21 # Find the matching file
22 matching_files = []
23 for filename in os.listdir(q2_files_path):
24     file_path = os.path.join(q2_files_path, filename)
25
26     if os.path.isfile(file_path): # Only process files, not directories
27         file_hash = compute_sha256(file_path)
28         if file_hash == expected_hash:
29             matching_files.append(filename)
30
31 # Output the result
32 if matching_files:
33     print(f"Matching file(s): {' '.join(matching_files)}")
34 else:
35     print("No matching file found.")
36
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
● cse@cse3140-HVM-domU:~/Lab3$ python3 Solutions/Q2.py
Matching file(s): unmeet.exe
○ cse@cse3140-HVM-domU:~/Lab3$
```

SSH: 10.13.6.145 Ln 15, Col 30 Spaces: 4 UTF-8 LF Python 3.8.10

Question #3: spinstex.exe



The screenshot shows a VS Code editor window with a Python script named `Q3.py` open. The script is designed to verify the authenticity of executable files (.exe) in a directory named `Q3files` using RSA digital signatures and SHA-256 hashing. The script imports `os`, `Crypto.PublicKey.RSA`, `Crypto.Hash.SHA256`, and `Crypto.Signature.pkcs1_15`. It defines paths for the lab directory, public key file (`Q3pk.pem`), and the directory containing the files to be verified (`Q3files`). The script then loads the public key, iterates through the files in `Q3files`, and for each .exe file, it computes its SHA-256 hash and compares it with the hash stored in the corresponding .sign file. The verification is performed using `pkcs1_15.new(public_key).verify(file_hash, signature)`. The output of the script is shown in the terminal at the bottom, which displays the message: `Correctly signed file: spinstex.exe`.

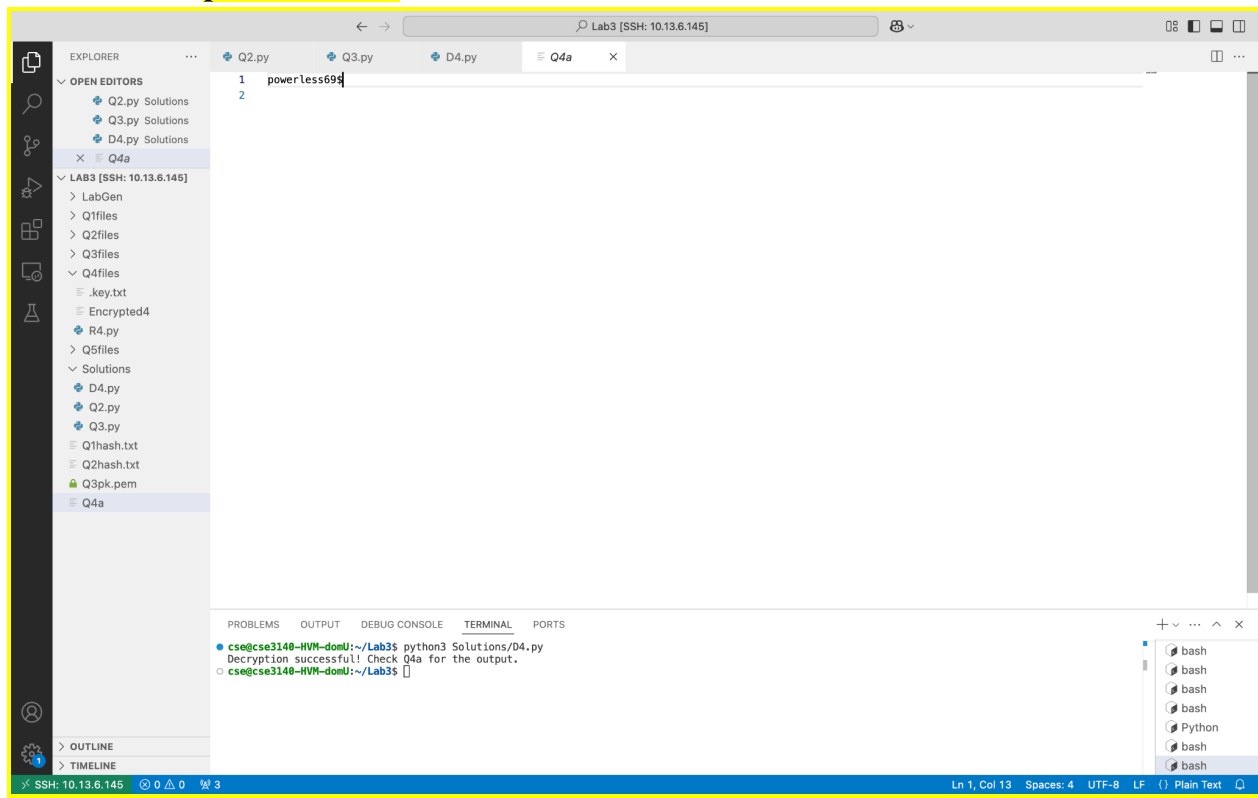
```
1 import os
2 from Crypto.PublicKey import RSA
3 from Crypto.Hash import SHA256
4 from Crypto.Signature import pkcs1_15
5
6 # Define paths
7 lab3_path = "/home/cse/Lab3"
8 q3_pubkey_file = os.path.join(lab3_path, "Q3pk.pem")
9 q3_files_path = os.path.join(lab3_path, "Q3files")
10
11 # Load public key
12 with open(q3_pubkey_file, "rb") as f:
13     public_key = RSA.import_key(f.read())
14
15 # Check each .exe file
16 for filename in os.listdir(q3_files_path):
17     if filename.endswith(".exe"):
18         exe_file = os.path.join(q3_files_path, filename)
19         sign_file = exe_file + ".sign" # Corresponding .sign file
20
21         # Skip if signature file is missing
22         if not os.path.exists(sign_file):
23             continue
24
25         # Compute SHA-256 hash of the executable
26         with open(exe_file, "rb") as f:
27             file_data = f.read()
28             file_hash = SHA256.new(file_data)
29
30         # Read the signature
31         with open(sign_file, "rb") as f:
32             signature = f.read()
33
34         # Verify the signature
35         try:
36             pkcs1_15.new(public_key).verify(file_hash, signature)
37             print(f"Correctly signed file: {filename}")
38             break # Stop once we find the correctly signed file
39         except (ValueError, TypeError):
40             pass # Ignore invalid signatures
```

Terminal Output:

```
cse@cse3140-HVM-domU:~/Lab3$ python3 Solutions/Q3.py
Correctly signed file: spinstex.exe
cse@cse3140-HVM-domU:~/Lab3$
```

In this experiment, we used RSA digital signatures with SHA-256 hashing to verify the authenticity of `.exe` files in the `Q3files` directory. The public key from `Q3pk.pem` was used to check the validity of each file's signature. The process involved computing the SHA-256 hash of each `.exe` file and comparing it with its corresponding `.sign` file using PKCS#1 v1.5 verification. The correctly signed file was identified as `spinstex.exe`, confirming that it was signed by the legitimate software vendor. This experiment highlighted the role of digital signatures in ensuring both integrity and authenticity, as the verification process successfully filtered out improperly signed or modified files.

Question #4: powerless69\$



The screenshot shows a VS Code editor window with the following components:

- EXPLORER:** Displays the file structure of the 'LAB3 [SSH: 10.13.6.145]' project. The 'Q4a' file is selected under the 'Q4files' directory.
- CODE EDITOR:** Shows the content of the 'Q4a' file, which contains two lines of text: '1 powerless69\$' and '2'.
- TERMINAL:** Displays the output of a command executed in the terminal. The command is 'python3 Solutions/D4.py', and the output is 'Decryption successful! Check Q4a for the output.'

```
1 powerless69$
2
```

```
cse@cse3148-HVM-domU:~/Lab3$ python3 Solutions/D4.py
Decryption successful! Check Q4a for the output.
cse@cse3148-HVM-domU:~/Lab3$
```

Question #5: applewood92&

The screenshot shows a VS Code editor window with the following details:

- Explorer Sidebar:** Displays the project structure for 'LAB3 [SSH: 10.13.6.145]'. It includes folders like 'LabGen', 'Q1files', 'Q2files', 'Q3files', 'Q4files', and 'Solutions'. The 'Q5a' file is selected under the 'Solutions' folder.
- Editor Window:** The file 'Q5a' is open, showing the content 'applewood92&' on two lines.
- Terminal Panel:** Shows the following output:

```
cse@cse3140-HVM-domU:~/Lab3$ python3 D5.py
python3: can't open file 'D5.py': [Errno 2] No such file or directory
cse@cse3140-HVM-domU:~/Lab3$ cd Solutions
cse@cse3140-HVM-domU:~/Lab3/Solutions$ python3 D5.py
cse@cse3140-HVM-domU:~/Lab3/Solutions$
```
- Status Bar:** Indicates the current file is 'Q5a' at line 1, column 13, using UTF-8 encoding.

Question #6: All files and screen recording located in ZipFile.

For this lab project, I chose to use RSA as the public key cryptosystem because it is widely used for secure encryption and key exchange. RSA ensures that the symmetric key, which encrypts the actual files, can only be decrypted by the private key, making it an effective method for ransomware-like encryption. I selected a key size of 2048 bits because it provides strong security while maintaining reasonable computational efficiency. A smaller key, such as 1024 bits, would be vulnerable to modern cryptographic attacks, while larger keys would increase processing time without significant security benefits.

To test my implementation, I first generated a public and private key pair using my key generation program. Then, I ran my ransomware program, which hardcoded the public key and used it to encrypt a randomly generated symmetric key. This symmetric key was stored in EncryptedSharedKey and was used to encrypt all **.txt** files in the working directory, renaming them with the **.txt.encrypted** extension.

Next, I acted as the attacker and used my decryption script to take EncryptedSharedKey and decrypt it using the private key, recovering the original symmetric key and saving it as DecryptedSharedKey. Finally, I ran the victim's decryption script, which used this decrypted key to restore all encrypted files to their original **.txt** format.

The process worked as expected, demonstrating how ransomware typically operates by encrypting files in a way that requires the attacker's private key for decryption. I recorded my screen while testing the program to document each step, ensuring that the key generation, encryption, and decryption processes functioned correctly.