Here are a bit of explanations of the codes written.

1.Password Manager

This is a Python program for a password manager that allows the user to store, retrieve, remove, save, and load passwords for different websites. Here is a brief explanation of the different parts of the code:

1. The **getpass** module is imported to allow the user to input passwords without them being displayed on the screen.
2. The **PasswordManager** class is defined with the following methods:
   * **\_\_init\_\_**: Initializes an empty dictionary to store the passwords.
   * **add\_password**: Adds a new password to the dictionary with the specified site, username, and password.
   * **remove\_password**: Removes the password for the specified site from the dictionary.
   * **get\_password**: Retrieves the username and password for the specified site from the dictionary.
   * **save\_passwords**: Saves the passwords to a binary file using the **pickle** module.
   * **load\_passwords**: Loads the passwords from a binary file using the **pickle** module.
3. The **main** function is defined, which creates an instance of the **PasswordManager** class and displays a menu of options for the user to choose from.
4. The user's choice is read from input, and the corresponding action is taken based on the choice:
   * If the choice is 1, the user is prompted to enter the site, username, and password, and the **add\_password** method is called to store the information.
   * If the choice is 2, the user is prompted to enter the site, and the **remove\_password** method is called to delete the information.
   * If the choice is 3, the user is prompted to enter the site, and the **get\_password** method is called to retrieve the information and display it.
   * If the choice is 4, the user is prompted to enter a filename, and the **save\_passwords** method is called to save the passwords to a file.
   * If the choice is 5, the user is prompted to enter a filename, and the **load\_passwords** method is called to load the passwords from a file.
   * If the choice is 6, the program ends.
5. The **\_\_name\_\_** variable is checked to ensure that the **main** function is only called if the program is run directly, not if it is imported as a module.

2.SHA-1

This is a Python program that demonstrates how to hash a message using the SHA-1 hash function from the hashlib library. Here is an explanation of the code:

1. The first line of code imports the hashlib library, which contains the SHA-1 hash function as well as other cryptographic hash functions.
2. The next line of code prompts the user to enter a message to be hashed, and stores the input in the variable **message**.
3. The **message** variable is then encoded as bytes using the UTF-8 encoding, which is a common encoding used for text.
4. The SHA-1 hash function is then applied to the **message\_bytes** variable using the **hashlib.sha1()** method. This creates a hash object that contains the SHA-1 hash of the message.
5. The **hexdigest()** method is called on the hash object to convert the binary hash value into a hexadecimal string representation.
6. Finally, the resulting hash value is printed to the console using the **print()** function.

Note that the SHA-1 hash function is no longer considered secure and has been deprecated in favor of newer hash functions such as SHA-256 and SHA-3.

3. check\_password\_strength

Let's go through the code line by line:

1. We import the **re** module, which provides support for regular expressions (regex).
2. We define a function **check\_password\_strength** that takes a password as input and returns a score between 0 and 10 based on how strong the password is.
3. Inside the function, we define a list of regex patterns to check for various password criteria, such as lowercase letters, uppercase letters, digits, and special characters.
4. We initialize a variable **score** to 0, which we will use to keep track of the password's strength score.
5. We loop through each pattern in the list, and use **re.search** to check if the password meets the pattern. If it does, we add 2 points to the score.
6. We check the length of the password using **len**, and assign a score between 0 and 2 based on its length.
7. We add up all the scores from steps 5 and 6 to get the final score between 0 and 10.
8. We return the final score from the function.
9. We prompt the user to enter a password, and call the **check\_password\_strength** function with the input as argument.
10. We print the final score to the console.

4.keylogger

This Python script uses the **pynput** library to monitor keyboard input and log the keys pressed to a file called **log.txt**.

The **pynput** library provides a cross-platform way to listen for and control input devices like keyboards and mice. In this case, it's used to listen for keyboard input.

The **on\_press** function is a callback function that is called every time a key is pressed. It takes a **key** parameter, which represents the key that was pressed. In this function, the key is written to the **log.txt** file.

The **Listener** class is used to create a listener object that will call the **on\_press** function for each key press. The **with** statement is used to create a context for the listener object. This ensures that the listener is stopped and resources are cleaned up properly when the program exits.

The **join** method is called on the listener object to start the listener. This method will block the program and wait for input until the program is terminated.

Top of Form

5.packet sniffer

1. **import socket**: This imports the Python socket module, which provides access to low-level networking interfaces.
2. **import struct**: This imports the Python struct module, which provides functions to pack and unpack binary data.
3. **s = socket.socket(socket.AF\_INET, socket.SOCK\_RAW, socket.IPPROTO\_IP)**: This creates a raw socket using the AF\_INET address family and the SOCK\_RAW socket type. The IPPROTO\_IP protocol constant indicates that we want to receive all IP packets.
4. **s.setsockopt(socket.IPPROTO\_IP, socket.IP\_HDRINCL, 1)**: This sets the IP\_HDRINCL option on the socket to 1, which tells the kernel to include the IP header in the received packet.
5. **s.bind(("0.0.0.0", 0))**: This binds the socket to all available network interfaces and a random port number.
6. **while True: ...**: This starts an infinite loop to receive and process packets.
7. **packet, address = s.recvfrom(65535)**: This receives a packet from the socket and returns the packet data and the address of the sender as a tuple.
8. **ip\_header = packet[0:20]**: This extracts the first 20 bytes of the packet, which contains the IP header.
9. **iph = struct.unpack('!BBHHHBBH4s4s', ip\_header)**: This unpacks the binary IP header data into a tuple of individual fields using the '!' byte order specifier and the specified format string. The format string specifies the sizes and types of the fields in the header.
10. **version\_ihl = iph[0]**: This extracts the first byte of the IP header, which contains the version and header length fields.
11. **version = version\_ihl >> 4**: This extracts the version field by shifting the first byte right by 4 bits.
12. **ihl = version\_ihl & 0xF**: This extracts the header length field by masking the first byte with 0xF (binary 1111).
13. **iph\_length = ihl \* 4**: This calculates the total length of the IP header in bytes.
14. **ttl = iph[5]**: This extracts the time to live (TTL) field from the IP header.
15. **protocol = iph[6]**: This extracts the protocol field from the IP header, which indicates the type of the payload.
16. **s\_addr = socket.inet\_ntoa(iph[8])**: This converts the source IP address from binary format to dotted-decimal notation using the inet\_ntoa function.
17. **d\_addr = socket.inet\_ntoa(iph[9])**: This converts the destination IP address from binary format to dotted-decimal notation using the inet\_ntoa function.
18. **print(f"Packet from {address[0]} to {d\_addr} ({protocol})")**: This prints some information about the received packet, including the source and destination IP addresses and the protocol.
19. **# do something with the packet ...**: This is a placeholder for any additional processing or analysis that you want to perform on the packet data.
20. **s.close()**: This closes the socket when the program is terminated.

6.network scanner

This is a Python script that uses the Scapy library to perform an ARP scan on a target IP address or range of IP addresses. The ARP (Address Resolution Protocol) is used to map a network address (such as an IP address) to a physical address (such as a MAC address).

The script first defines a function **get\_arguments()** that uses the **argparse** library to parse command-line arguments. It expects a single argument **-t** or **--target** that specifies the target IP address or range of IP addresses to scan.

The **scan()** function takes an IP address or range as input and creates an ARP request packet using Scapy. It then combines the ARP request packet with an Ethernet frame and sends it to the network using the **srp()** function of Scapy. This function sends the packet to the network and waits for a response for a maximum of 3 seconds, then returns a list of packets that were received. The function then parses the response packets and extracts the IP and MAC addresses of each responding device. The function returns a list of dictionaries that contain the IP and MAC addresses of the devices that responded to the ARP request.

The **print\_result()** function takes the list of dictionaries returned by **scan()** and formats the output in a table with two columns: IP address and MAC address.

Finally, the script uses the **get\_arguments()** function to retrieve the target IP address or range from the command-line arguments, calls the **scan()** function to perform the ARP scan, and displays the results using the **print\_result()** function.

7.Digital forensic tool

This code extracts metadata from an image file using the Python Imaging Library (Pillow).

Here's how it works:

* First, the path of the image file is specified using a string and assigned to the variable **image\_path**.
* Then, the image file is opened using the **Image.open()** method of the **Image** class from the **PIL** module. The **with** statement is used to ensure that the file is properly closed after it has been used.
* Next, the **info** attribute of the **Image** object is used to retrieve the metadata of the image file.
* Finally, the metadata is printed using the **print()** function.

Note that this code only retrieves the metadata that is stored in the **info** attribute of the **Image** object. Some image files may store metadata in other locations, such as the EXIF data. In those cases, additional steps may be required to retrieve the full metadata.

8.file path traversal

Path Traversal, also known as Directory Traversal, is a vulnerability that occurs when user input is not properly validated or sanitized, allowing an attacker to access files or directories outside the intended scope. Here are the steps involved in a path traversal attack:

1. User input: The attacker identifies an input field or parameter where they can supply a file path or directory.
2. Input manipulation: The attacker modifies the input by adding special characters or sequences that can bypass the validation or sanitization mechanisms in place.
3. Directory traversal: The attacker uses special characters, such as "../" or "..", to navigate up the directory tree and access files or directories outside the intended scope.
4. File access: The attacker specifies the desired file or directory they want to access by traversing the directory structure. They may target sensitive files containing passwords, configuration files, or other valuable information.
5. Unauthorized access: If the path traversal is successful, the attacker gains unauthorized access to files or directories they shouldn't have access to.
6. Exploitation: The attacker can use the accessed files or directories for various purposes, such as extracting sensitive information, modifying files, executing arbitrary code, or further exploiting the system.

**To prevent path traversal vulnerabilities, it's important to implement proper input validation and sanitization techniques:**

1. Implement input validation: Validate and sanitize user input by checking for allowed characters and patterns. Whitelist validation is recommended, where only known safe inputs are accepted.
2. Use secure file access APIs: Utilize file access APIs that provide built-in protection against path traversal attacks. These APIs handle path resolution and ensure that file operations are limited to the intended directories.
3. Normalize file paths: Normalize and canonicalize user-supplied file paths to remove any relative path components and ensure they resolve to the correct absolute paths.
4. Apply access controls: Implement appropriate access controls and permissions to restrict file system access based on user roles and privileges. Users should only be able to access files and directories they are authorized to access.
5. Implement server-side input validation: Even if client-side validation is in place, server-side validation is crucial to prevent bypassing client-side restrictions.
6. By following these practices, you can significantly reduce the risk of path traversal vulnerabilities and enhance the security of your application or system.

**code explanation:**

1. The import statements at the beginning of the code import necessary modules: os, re, and sys.
2. The main() function is defined as the entry point of the script.
3. filename = sys.argv[1] retrieves the filename specified as a command-line argument when running the script. sys.argv is a list that contains command-line arguments, where sys.argv[0] is the script name itself.
4. re.findall('[\\/]\*([a-zA-Z0-9]+[.][a-zA-Z0-9]+)$', filename) uses regular expressions to extract the filename from the provided path. It matches alphanumeric characters with a dot in the filename part of the path.
5. filename = re.findall(...)[0] retrieves the first matched filename from the list returned by re.findall().
6. path = os.path.join(os.getcwd(), filename) creates the full path to the file by joining the current working directory (os.getcwd()) with the extracted filename.
7. The code enters a try block to handle potential exceptions.
8. print('Reading:', os.path.abspath(path)) displays the absolute path of the file that is being read.
9. with open(path, 'r') as f: opens the file in read mode using a with statement, ensuring that the file is properly closed after reading.
10. file\_data = f.read() reads the contents of the file into the file\_data variable.
11. print('File data:', file\_data) displays the contents of the file.
12. If the file is not found, a FileNotFoundError is raised and caught in the except block. The error message "Error - file not found" along with the exception details is printed.
13. Finally, the main() function is called to start the execution of the script.