|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Information Security (116U01L602)** | **Semester:** | **VI** |
| **Date of Performance:** | **05 / 02 / 2025** | **DIV/ Batch No:** | **A-3** |
| **Student Name:** | **Kashish Mamania** | **Roll No:** | **16010122104** |

|  |
| --- |
| **Title: Implementation of the program , Exploring other forms / types of CAPTCHAs** |

|  |
| --- |
| **Objectives:** |
| To write a program to convert plain text into cipher text using Caesar cipher and Transposition cipher |

|  |
| --- |
| **Expected Outcome of Experiment:** |
| **CO1** |

|  |
| --- |
| **Books/ Journals/ Websites referred:** |
|  |

|  |
| --- |
| **Pre Lab/ Prior Concepts:** |
|  |

|  |
| --- |
| **New Concepts to be learned:** |
|  |

|  |
| --- |
| **Abstract:** |
| CAPTCHA stands for Completely Automated Public Turing test to tell Computers and Humans Apart. CAPTCHAs are tools used to differentiate between real users and automated bots. They provide challenges that are difficult for computers to solve but relatively easy for humans. Examples include identifying stretched letters or numbers or clicking in specific areas. CAPTCHAs are widely used by websites to restrict usage by bots. |

|  |
| --- |
| **Related Theory:** |
| Specific Uses of CAPTCHA  Maintaining Poll Accuracy  CAPTCHAs prevent poll skewing by ensuring each vote is entered by a human. Although this does not limit the total number of votes, it increases the time required for each vote, discouraging multiple votes.  Limiting Registration for Services  CAPTCHAs protect registration systems from bots creating fake accounts, reducing fraud and conserving resources.  Preventing Ticket Inflation  Ticketing systems use CAPTCHAs to limit scalpers from purchasing large numbers of tickets for resale or false registrations for free events.  Preventing False Comments  CAPTCHAs prevent bots from spamming message boards, contact forms, or review sites. The extra step required also helps reduce online harassment by adding inconvenience. |

|  |
| --- |
| **Implementation Details:** |
| # Import required libraries  import random  import string  from captcha.image import ImageCaptcha  # Function to generate randomized CAPTCHA  def generateCaptcha():      R1 = random.randint(1, 9)  # CAPTCHA length between 1 and 9      captcha = []      for i in range(R1):          R2 = random.randint(0, 9)          if R2 < 6:              R3 = random.randint(0, 9)              captcha.append(str(R3))          else:              R3 = random.choice(string.ascii\_letters)              captcha.append(R3)      return ''.join(captcha)  # Function to create image-based CAPTCHA  def createImageCaptcha(captcha\_text, image\_path='CAPTCHA.png'):      image = ImageCaptcha(width=280, height=90)      data = image.generate(captcha\_text)      image.write(captcha\_text, image\_path)      return image\_path  # Main authentication logic  def captcha\_authentication():      print("\n--- CAPTCHA Authentication System ---\n")        while True:          # Choose CAPTCHA type          captcha\_type = input("Choose CAPTCHA type:\n[T]ext-based\n[I]mage-based\n[Q]uit\nEnter choice: ").strip().lower()            if captcha\_type == 'q':              print("\n🔒 Session terminated.")              break            captcha\_text = generateCaptcha()          if captcha\_type == 't':              # Text-based CAPTCHA              print(f"\nGenerated Text CAPTCHA: {captcha\_text}")            elif captcha\_type == 'i':              # Image-based CAPTCHA              image\_path = createImageCaptcha(captcha\_text)              print("\nImage-based CAPTCHA generated.")              print(f"Please open the image at '{image\_path}' to view the CAPTCHA.")            else:              print("⚠️ Invalid choice. Please select again.\n")              continue            # User verification loop          while True:              user\_input = input("\nEnter the CAPTCHA: ").strip()                if user\_input == captcha\_text:                  print("✅ Captcha matches. Authentication successful!\n")                  return                else:                  print("❌ Captcha didn't match.")                  retry\_choice = input("[R]etry | [N]ew CAPTCHA | [Q]uit: ").strip().lower()                    if retry\_choice == 'q':                      print("\n🔒 Session terminated.")                      return                    elif retry\_choice == 'n':                      break  # Generate new CAPTCHA                    elif retry\_choice != 'r':                      print("⚠️ Invalid option. Defaulting to retry.")  # Run the authentication system  if \_\_name\_\_ == "\_\_main\_\_":      captcha\_authentication() |

|  |
| --- |
| **Results/Output:** |
|  |

|  |
| --- |
| **Conclusion:** |
| In this experiment, we explored the core concepts of CAPTCHA, its role in enhancing security for  digital platforms, and its implementation through both text-based and image-based methods. By  designing a program to generate CAPTCHA challenges and validate user responses, we gained  practical experience with the algorithms used to distinguish human users from automated bots.  This hands-on approach provided valuable insights into how CAPTCHAs help protect online  services from spam, automated attacks, and fraudulent activities. Additionally, this exercise  strengthened our programming skills and deepened our understanding of the critical role  cybersecurity measures play in modern computing. |

|  |
| --- |
| **Post-Lab Questions:** |
| **3.1 Difference Between Malicious and Non-Malicious Program Flaws**   | **Aspect** | | **Malicious Program Flaws** | | **Non-Malicious Program Flaws** | | --- | --- | --- | --- | --- | | **Definition** | | Intentional flaws introduced to exploit or harm systems, steal data, or disrupt operations. | | Unintentional flaws caused by programming errors or oversight during development. | | **Intent** | | Deliberate and harmful. | | Accidental and unintended. | | **Examples** | | - Viruses, worms, trojans. | | - Buffer overflows, null pointer dereferences, logic errors. | | **Impact** | | Can lead to data theft, system crashes, unauthorized access, or financial loss. | | Can cause system instability, crashes, or incorrect program behavior. | | **Example Programs** | - A trojan horse that steals user credentials. | | - A program that crashes due to improper handling of user input. | |   **3.2 Types of Buffer Overflow Attacks**   1. **Stack-Based Buffer Overflow:**    * Occurs when a program writes more data to a stack buffer than it can hold.    * Overwrites the return address of a function, allowing attackers to execute arbitrary code.    * Example: Exploiting a vulnerable gets() function in C. 2. **Heap-Based Buffer Overflow:**    * Happens when data written to a heap buffer exceeds its allocated size.    * Allows attackers to overwrite adjacent memory areas on the heap.    * Example: Exploiting dynamic memory allocation flaws (e.g., malloc()). 3. **Integer Overflow Leading to Buffer Overflow:**    * Occurs when an arithmetic operation exceeds the maximum value of an integer type.    * Can lead to allocating insufficient memory and subsequent overflow.    * Example: Using an unsigned integer for buffer size calculations. 4. **Format String Vulnerabilities:**    * Exploits improper use of formatting functions (e.g., printf()).    * Allows attackers to write arbitrary data to memory locations. 5. **Off-by-One Error:**    * A subtle overflow where one extra byte is written beyond the allocated buffer.    * May lead to corruption of adjacent memory.   **3.3 Mechanisms to Prevent Buffer Overflow Attacks**   1. **Input Validation:**    * Validate all user inputs to ensure they do not exceed expected sizes. 2. **Use Safe Functions:**    * Replace unsafe functions like gets() and strcpy() with safer alternatives like fgets() and strncpy(). 3. **Address Space Layout Randomization (ASLR):**    * Randomizes the memory address space of processes, making it harder for attackers to predict memory locations. 4. **Stack Canaries:**    * Places a "canary value" before the return address on the stack.    * If the value is altered, the program detects the overflow and terminates. 5. **Non-Executable Stack (NX Bit):**    * Marks stack memory as non-executable, preventing execution of injected code. 6. **Compiler Protections:**    * Use compiler flags like -fstack-protector in GCC to enable stack protection mechanisms. 7. **Code Auditing and Static Analysis:**    * Regularly audit code for vulnerabilities and use static analysis tools to detect potential buffer overflows. 8. **Use Modern Programming Languages:**    * Languages like Python and Java manage memory automatically, reducing the risk of buffer overflows.   **3.4 How Canary Values Are Used to Overcome Stack Buffer Overflow**   * A **canary value** is a small piece of data placed between a buffer and control data (e.g., return address) on the stack. * The canary value is initialized with a random number during program execution. * Before returning from a function, the program checks if the canary value has been altered:   1. If the canary is intact, the program continues execution normally.   2. If the canary has been modified (indicating a buffer overflow), the program terminates immediately to prevent exploitation. * This mechanism prevents attackers from overwriting critical control data without detection.   **Example Workflow:**   1. Before a function call:   text  [Buffer] [Canary Value] [Return Address]   1. After an overflow attempt:   text  [Overflow Data] [Modified Canary] [Modified Return Address]   1. Canary check fails → Program halts execution. |