Assignment 2 Rubric

1. Tor, Anonymity and Usability [30]
    a. Download and install the Tor browser on your respective systems. [0]
    b. What are some onion links that you visited using the Tor browser? What was your experience like browsing through these sites? [5]
    c. In your opinion, how usable is the Tor browser compared to other popular browsers like Chrome or Firefox? What are some advantages and disadvantages of using Tor? [5]
    d. Why do you think onion links use such unusual and seemingly random URLs? What is the reasoning behind choosing these names? [5]
    e. How do you think the unique URL naming convention impacts the usability of .onion links? Do you find it more difficult or confusing to navigate these sites compared to regular URLs? [5]
    f. In your opinion, how can one balance the need for anonymity and security provided by the Tor browser and .onion links, with the usability and ease of navigation for users? Are there any potential solutions to this challenge? [10].

Ans.

(b) Onion links visited & browsing experience [5 marks]

Mark on:

- **Relevance** (did they talk about using Tor / onion sites?)
- **Concreteness** (do they give some concrete description instead of only “it was weird”?)
- **Reflection on experience** (speed, comfort, trust, difficulty, etc.)
- **Usability angle** (do they mention things like navigation, ease of finding info, trust cues, etc.?)

Suggested scale:

- **4–5 marks** – Clear description of what they did, with **specific observations** (e.g., loading time, broken pages, how they felt, how easy/hard it was to use) and

some reflection on usability.

- **2–3 marks** – Basic description with a few observations; shows they used it but reflection is shallow or brief.

- **0–1 mark** – Very vague (“I visited some sites”) and almost no experience/reflection; or clearly off-topic.

(c) Usability of Tor vs Chrome/Firefox; advantages & disadvantages [5 marks]

Mark on:

- **Comparison** – Do they actually compare Tor with a regular browser?

- **Multiple aspects** – Speed, reliability, site compatibility, ease of use, interface, etc.

- **Trade-off thinking** – Do they see that Tor’s privacy features impact usability?

Suggested scale:

- **4–5 marks** – Compares Tor vs normal browsers on **several aspects** (e.g., speed, blocking, comfort, setup, extensions) and discusses **both advantages and disadvantages** with some reasoning.

- **2–3 marks** – Mentions some pros and cons and a basic comparison, but only on 1–2 aspects or with limited explanation.

- **0–1 mark** – Only generic statements (“Tor is secure”, “Chrome is faster”) with no real comparison or reasoning.

(d) Why onion URLs look random/strange [5 marks]

Mark on:

- **Awareness of purpose** – Do they realise it’s intentional, not “just ugly”?

- **Anonymity/security idea** – Hard to guess, hide location, etc.

- **Connection to a threat/goal** – Avoid easy discovery, censorship, tracking, etc.

Suggested scale:

- **4–5 marks** – Gives a **clear explanation** that these addresses are designed this way for **security/anonymity reasons** (e.g., hard to guess, protect service location, resisting blocking/scanning) with some reasoning.

- **2–3 marks** – Mentions that it is for "security/anonymity" but only briefly; partial reasoning.

- **0–1 mark** – Explanation unrelated (e.g., "to look cool") or purely aesthetic with no security angle.

(e) Usability impact of onion naming scheme [5 marks]

Mark on:

- **Identification of usability issues** – memorability, error-prone typing, difficulty returning to sites, etc.

- **User perspective** – Do they put themselves in the shoes of a typical user?

- **Link to UX concepts** – cognitive load, navigation, trust, etc. (even in simple language).

Suggested scale:

- **4–5 marks** – Describes **multiple concrete usability problems** caused by random-looking URLs (e.g., hard to remember, easy to mistype, difficult to know if a site is "legit", hard to share) and reflects on how this affects everyday use.

- **2–3 marks** – Identifies that it is "confusing / hard to remember" and mentions one or two consequences; explanation modest but on-topic.

- **0–1 mark** – Just says "bad usability" with almost no explanation.

(f) Balancing anonymity/security with usability + suggestions [10 marks]

You want **maximum variation** here — students can argue for different design directions, and all can be valid if argued well.

Mark on **three components**:

1. Understanding the trade-off [0–3 marks]

- **3 marks** – Clearly recognises that improving anonymity/security can reduce usability (and vice versa), and gives at least one concrete example.
- **1–2 marks** – Mentions a trade-off in general terms ("more secure is less convenient") but without much detail.
- **0 marks** – No trade-off; argues only one side without acknowledging any cost.

2. Quality of ideas / suggestions [0–3 marks]

- **3 marks** – Proposes **2–3 concrete ideas** to improve the situation, such as:
    - better onboarding/help without weakening anonymity
    - curated directories or search tools
    - safer bookmarking/address management
    - clearer warnings, better UX, privacy-by-default configs, etc.
      Ideas can be very different from student to student; reward **specific, feasible thinking**.
- **1–2 marks** – At least one idea, but vague ("improve UI") or not very connected to anonymity/usability.
- **0 marks** – No real suggestion or completely off-topic.

3. Justification & reflection [0–4 marks]

- **4 marks** – Explains *why* their ideas make sense, shows some empathy for different users (novice vs expert, activists vs casual users), maybe references

course concepts (usable security, mental models, risk communication, etc. in any language).

- **2–3 marks** – Some justification and reflection, but limited depth.
- **0–1 mark** – Mostly opinion with little reasoning ("Tor should just… because I think so").

2. Reverse Engineering [30]

Please download any executable (exe or APK), and utilize tools such as Appium, Apktool, JADX, dogbolt, to decompile executable files and provide outputs. With the help of de-compilation tools, attempt to answer the following questions by decompiling the executable.

a. What are the system calls being performed by the executable and what are their roles. [5]
b. In order to get the "right" output what should be given as input? [5]
c. Explain the working of the executable by executing it and then retracing it to the decompiled output. [10]
d. In terms of useability and readability would you rather prefer output [eg. from ghidra or hex-rays]? Please explain your answer. [10]

Ans.

a) System calls and their roles [5 marks]

Rubric:

- 5 marks (Excellent)
    - Lists the main relevant calls (not necessarily every single low-level call, but the important ones).
    - Roles are clear and correct ("changes window title", "pauses until keypress", "reads user input", etc.).
    - Shows they mapped code to behaviour (not just copy-pasted names).

- 3–4 marks (Adequate)
    - Identifies most of the important calls, with mostly correct roles.
    - Some minor omissions or slightly vague descriptions, but overall shows understanding.
- 1–2 marks (Weak)
    - Only a few calls identified; or roles are very vague/confused.
    - Suggests they saw the names but didn't really understand what they do.
- 0 marks (No real attempt / off-topic)
    - No meaningful calls listed, or roles are essentially random / unrelated.

Note: Different executables will naturally have different sets of calls; that's okay. You only care whether they correctly describe their chosen program.

(b) Input needed to get the "right" output [5 marks]

Rubric:

- 5 marks (Excellent)
    - Clearly states the correct input(s) (could be a word, number, pattern, etc.).
    - Explains how they figured it out (e.g., "I saw this constant / condition in the decompiled code" or "I ran it and matched with the check").
    - Shows a real link between decompiled logic and behaviour.
- 3–4 marks (Adequate)
    - Gives the correct input, but explanation is brief ("this input gives RIGHT") or only partially justified.

    - Or: minor inaccuracy (e.g., misses a corner case) but generally understands what triggers “right” output.

- 1–2 marks (Weak)
    - Input seems guessed or partially correct; little or no evidence they used the code.
    - Minimal explanation of how they arrived at it.
- 0 marks (Incorrect / no attempt)
    - Clearly wrong input with no reasoning, or answer doesn’t match their own program’s behaviour.

Again, the actual string/number will differ per student; you only check if their reasoning matches their program.

(c) Explain the working of the executable [10 marks]

.

Rubric (breakdown idea: 4 + 4 + 2):

1. High-level behaviour (0–4 marks)
    - 4 – Clear, accurate story of what the program does from a user’s perspective (what user sees, what happens internally).
    - 2–3 – Partially clear; main idea is right but misses some important steps or minor inaccuracies.
    - 0–1 – Very vague or mostly wrong.
2. Link to decompiled code (0–4 marks)
    - 4 – Explicitly connects observed behaviour to specific parts of the decompiled output (e.g., “this branch prints X”, “this loop re-prompts

input”).

- 2–3 – Some references to code, but not very systematic; still shows they looked at it.
- 0–1 – No real mapping; description could have been guessed from running only.

3. Clarity and structure (0–2 marks)
    - 2 – Well-structured, easy to follow; uses simple steps or bullet points.
    - 1 – Understandable but messy.
    - 0 – Hard to follow / incoherent.

Different programs → very different stories. Full marks if their story is correct for their program and clearly linked to the code.

(d) Preferred decompiler output & usability/readability comparison [10 marks]

Rubric (suggested: 3 + 4 + 3):

1. Comparison of tools (0–3 marks)
    - 3 – Clearly compares at least two tools/outputs (e.g., Ghidra vs Hex-Rays, different Dogbolt backends) on multiple aspects (layout, clarity, noise, etc.).
    - 1–2 – Some comparison, but only on one or two aspects or very brief.
    - 0 – No real comparison; just says “X is better” without context.
2. Usability & readability reasoning (0–4 marks)
    - 4 – Uses specific, concrete observations: “tool A grouped logic better”, “tool B had fewer irrelevant lines”, “this one’s pseudocode looked closer to C”, “variable naming / control flow was easier to follow”, etc. Ties them to usability concepts like cognitive load, readability, ease of navigation.

- 2–3 – Mentions some relevant factors (e.g., “more readable”, “easier to understand”) but with limited detail.

- 0–1 – Very generic (“nice UI”, “looks good”) with almost no connection to code readability.

3. Justification & reflection (0–3 marks)

    - 3 – Explains *why* their preferred output helps them understand the program; may mention their own experience/learning style (“this representation helped me reason about conditions better”).

    - 1–2 – Some justification but shallow; more like personal preference without much reflection.

    - 0 – Bare opinion without reasoning.

---

3. Authentication [15 points]

Suggest ways to make the IIIT-Delhi Authentication System more usable. Address the topics that we discussed in class. Answer the question in less than 450 words. In your proposed system, please discuss:

- What usability issues you feel need to be addressed? [7.5]
- Include some suggestions to improve the system. [7.5]

Remember there is a trade-off, the scheme should provide reasonable security at least.

Ans.

IIIT-Delhi uses multiple independent authentication schemes across its services: the ERP portal (https://erp.iiitd.edu.in), LDAP-based authentication for WiFi and device registration (https://it.iiitd.edu.in), GitLab (https://git.iiitd.edu.in), SSH access to research servers, FortiClient/GlobalProtect VPN, the Library’s EZProxy remote access service (https://ezproxy.iiitd.edu.in), and Google-based authentication for Moodle (https://moodle.iiitd.ac.in). While each system works independently, the combined experience presents several usability problems reflected in the concepts discussed in the course.

1. Usability Issues (7.5 marks)

- A major issue is the fragmentation of credentials. ERP, LDAP, Google accounts, and EZProxy all require distinct passwords or flows. This inconsistency makes it difficult for users to build stable mental models, a problem highlighted in the Mental Models material (Lecture #7), and increases cognitive load during authentication.
- Password practices create a second challenge. LDAP imposes strict complexity rules, and ERP maintains a separate password. As noted in the Authentication and Password slides (e.g., password entropy issues, coping behaviors such as pattern reuse) (Lecture #2 - Lecture #8), such designs often lead users to adopt predictable or insecure passwords in an attempt to cope.
- Third, the lack of actionable error feedback impairs usability. WiFi failures display only “Authentication Error,” ERP reports “Invalid Credentials,” and VPN clients commonly fail without specifying the source of the failure. Poor feedback, a known usability barrier, is addressed in the Usability Studies section (e.g., the importance of clear error information and recovery paths) (Lecture #2 - Lecture #3) .
- Additionally, the inconsistency of user interfaces across authentication points (ERP, EZProxy, GitLab, WiFi portals, VPN pop-ups) reduces user confidence and increases susceptibility to interface-based phishing attacks. The Social Engineering and Phishing material (e.g., risks created by inconsistent trust cues) directly discusses this issue (Lecture #6).
- Finally, account recovery is fragmented: ERP, LDAP, and Google each require separate reset mechanisms, making the process time-consuming and confusing.

2. Suggested Improvements (7.5 marks)

- A unified Single Sign-On (SSO) system using SAML or OIDC should be adopted so that ERP, LDAP-based services, EZProxy, VPN, and GitLab share a single identity. This reduces cognitive load and supports consistent interaction patterns.
- Introducing modern authentication options—such as WebAuthn for passwordless login, optional TOTP-based MFA, and SSH key–based access for research servers—would reduce password fatigue while maintaining strong security, consistent with principles of usable authentication (Lecture #2 - Lecture #5).
- Error messages should be redesigned to provide clear, actionable guidance, such as indicating whether a password has expired or linking to the appropriate reset mechanism. This aligns with principles of effective feedback and error recovery (Lecture #2 - Lecture #3).
- A unified IIIT-D login interface template should be applied across ERP, EZProxy, GitLab, and other portals to establish consistent visual identity and reduce phishing risk (Lecture #6).
- Finally, IIIT-Delhi should offer a centralized account-management dashboard for password updates, MFA configuration, SSH key management, and device registration, improving transparency and reducing user reliance on IT support.

Conclusion (not needed)

Streamlining authentication, improving feedback, reducing password burden, and standardizing interfaces would significantly enhance both usability and security across IIIT-Delhi's ecosystem.

Rubric (15 marks)

1. Identifying Usability Issues — 7.5 marks

Marks	Criterion
3	Mentions real IIITD authentication problems (multiple passwords, WiFi errors, ERP/LDAP confusion, inconsistent UI, VPN failures, etc.)
2.5	Shows conceptual understanding (e.g., mental models, cognitive load, password burden, poor feedback, phishing risk). Precise terminology not required.
2	Provides brief reasoning for why these issues affect usability.

2. Suggested Improvements — 7.5 marks

Marks	Criterion
3	Proposes reasonable, practical improvements (SSO, clearer error messages, unified interface, MFA, better recovery).
2.5	Connects improvements to good design principles (consistency, clarity, lower cognitive load, reducing friction).
2	Acknowledges that security must remain robust.

General Rubric Notes

- Students do not need to match the model answer.
- Partial credit should be awarded for any reasonable IIITD-specific issue or improvement.

---

4. Hashing [25 points]

Shiv, an IIIT-D Student needs to implement a simple OTP generation program. He cannot use any OTP generation library directly but is allowed to use hash library (hashlib) in python for generating hash values for any input. He thinks of an idea to generate and verify the OTP involving below steps:

- He plans to retrieve the current system time and date to be taken as input for generating hash as the value would be unique each time.
- He then plans to generate a hash (say H) of the string of date-time using SHA-256 function (using hashlib) and plans on storing the Hash H thus generated in a text file.
- Now he thinks of implementing a function (say F) which takes a hash value H as input to process specific characters/numbers from H to generate an OTP (say O) such that OTP is of length 4 and is completely numeric.
- He then plans to send the OTP O to the user and keep the value H with him (stored in the text file).
- For verification, he plans to build a simple program which takes O as input and H as input. It then performs function F on hash value H to give output O'. Now he'll check if O matches O'. If it matches, OTP verification will be "Successful", otherwise, it will be "Unsuccessful". Once an H matches, he'll remove that H from the text file. For simplicity, it is assumed that only one hash value H is generated and stored at a time

a. You need to implement both of the OTP generations and OTP verification programs and also describe the details of how did you implement function F. Also, Reason why you chose a specific function F. [12.5]
b. Could you think of any vulnerabilities in this OTP generation approach and in your implementation of F? If yes, mention them and explain how could they be exploited. If not, reason how is this approach and your F implementation secure. [12.5]

Note - You are not allowed to directly use pyotp or any other direct OTP generation libraries. If you're using Java / C then you can use an SHA-256 hash generation library but no direct OTP generation libraries allowed.

Ans.

A. OTP Generation & Verification (with Function F + alternatives) (12.5 marks)

1. System Overview

Shiv builds two programs: OTP Generation and OTP Verification, using only `hashlib` for SHA-256 hashing.

Generation Steps

1. Retrieve current date-time as string (`dt_str`), preferably in UTC for consistency. Example: `"2025-11-30T14:32:10Z"`.
2. Compute `H = SHA256(dt_str)` using `hashlib.sha256(dt_str.encode()).hexdigest()`.
3. Store `H` (the 64-character hex digest) in a text file (`stored_hash.txt`). Only one H exists at a time.
4. Compute OTP: `O = F(H)`, where F deterministically transforms H into a 4-digit numeric code.
5. Send `O` to the user.

Verification Steps

1. Read stored H from file.
2. Recompute `O' = F(H)` using the same function.
3. Compare user input `O` with computed `O'`.
4. If they match: successful; delete H from file. Otherwise: unsuccessful.

2. My Function F (Primary Implementation)

Chosen F — 8-Hex-Window + mod-10000

Idea:

- SHA256 digest is 64 hex characters.
- Extract an 8-hex-character slice (32 bits).
- Convert to integer.
- Reduce modulo 10000.
- Zero-pad to 4 digits.

Why this F?

- 32-bit window → good entropy distribution → uniform-ish OTP space.
- No bias from early digest bytes.
- Simple, deterministic, and easy to implement in Python.
- Uses only integer arithmetic + substring operations.

Pseudocode (Python-like)

```
import hashlib

def sha256_hex(s):
    return hashlib.sha256(s.encode()).hexdigest()  # 64 hex chars

def F(H_hex):
    segment = H_hex[16:24]     # choose middle window of 8 hex chars
    val = int(segment, 16)     # convert to integer
    otp = val % 10000          # constrain to 4 digits
    return f"{otp:04d}"        # zero-pad
```

3. Other Possible Valid Fs (for easy TA evaluation)

TAs should expect many variations. Any deterministic, well-explained mapping is valid.

(i) Digit-Sum Method

Extract digits from hex digest, sum them, apply `% 10000`.

- Convert hex digest to decimal string.
- Sum all digits, take `% 10000`.
- Zero-pad.

Pros: Very simple.

Cons: Lower entropy; more predictable.

(ii) XOR-Based Method

Split H into blocks of equal length (e.g., 16 hex chars each). Convert each to int, XOR all blocks, then `% 10000`.

- Split H into 4 blocks of 16 hex chars.
- Convert each block to integer.
- XOR all integers.
- Take `% 10000`.

Pros: Good mixing; more uniform.

Cons: Slightly more computation.

(iii) Even-Odd Index Mixing

Concatenate hex characters at even indices, convert to int, apply `%10000`.

- Take alternating hex digits, e.g., `H[0] + H[2] + H[4] + ...`.
- Convert concatenation to int → `%10000`.

Pros: Simple, unique to each student.

Cons: Not the highest entropy.

(iv) Hash-the-Hash

Compute `H2 = SHA256(H)` and use first N hex chars → int → `%10000`.

- Compute a second SHA-256: `H2 = SHA256(H)`.
- Take the first N hex digits of H2 → convert → mod → 4-digit OTP.

Pros: Adds mixing; prevents reliance on digest structure.

Cons: Extra hash call.

(v) ASCII Code Transformation

Take ASCII codes of several hex characters, sum them or multiply modulo 10000.

- Take selected ASCII codes of characters (e.g., `[ord(c) for c in H_hex[10:20]]`).
- Sum or multiply modulo some number.
- Reduce to 4 digits.

Pros: Acceptable if deterministic.

Cons: Usually lower entropy.

TA Note: Any F that:

1. Uses H deterministically,
2. Produces a 4-digit number,
3. Is clearly explained,
   must be considered valid.

B. Vulnerabilities & Security Analysis (12.5 marks)

This OTP scheme has several weaknesses due to the lack of a secret, predictability of datetime, and the structure of the OTP space.

Primary vulnerabilities

1. **No secret: scheme is entirely deterministic and public**
    - Because OTP = `F(H)` and `H` = SHA256(datetime), everything is predictable if an attacker learns stored `H`.
        i. If an attacker obtains `H` (file read or backup leak), `F(H)` is trivial to compute → the attacker can generate OTP.
        ii. Exploit: read stored file or intercept backup; compute OTP offline.

    - **Mitigation:** never store raw `H` or protect storage with strong file permissions. Better: incorporate a server-side secret key and compute `H = SHA256(secret || dt_str)` (i.e., a simple HMAC-like construct). Only the server knows the `secret`, so an attacker with dt or H cannot compute OTP.

2. **Datetime predictability / brute-force preimage**
    - Datetime values are highly predictable to within seconds/minutes.
        i. Because `H` is SHA256(datetime), an attacker who guesses the time window (e.g., within ±2 minutes) can brute-force possible `dt_str` values, compute `H` and then `O`.
        ii. Exploit: if OTP is issued recently, attacker enumerates plausible datetimes and derives OTPs.

    - **Mitigation:** include randomness/nonce or server secret (as above). Also use coarse time resolution and short expiry (e.g., OTP valid 30s), but that alone doesn't prevent guessing if the attacker knows exact time.

3. **Short OTP space (0000-9999)**
    - Only 10,000 possibilities.
        i. 4-digit OTP allows easy online brute-force with no rate-limiting.
        ii. Exploit: automated guess attempts at login form until success.

    - **Mitigation:** rate-limit attempts, lock after N failures, use longer OTP (6 digits) or alphanumeric tokens.

4. **Replay attacks**
    - OTP travels over email/SMS, and may be intercepted.

i. If OTP is intercepted in transit (SMS/email), the attacker can reuse it. Storing single-use H reduces the window, but transit must be protected.

- **Mitigation:** one-time use + very short expiry (30 seconds) + TLS and authenticated channels for delivery (or use push/secure app).

5. **Storage & file-system attacks**
   - H stored in a plaintext file is easily readable.
     i. Plaintext `stored_hash.txt` can be read by other users/processes.
   - **Mitigation:** secure file perms, avoid persistent storage; store only `SHA256(H || secret2)` or an ID mapping to ephemeral memory.
6. **F-specific Weaknesses: Collision/format assumptions**
   - Converting sub-hex and `%10000` is okay, but choosing the same window always could bias—minor issue vs security.
     i. Digit-sum reduces entropy drastically.
     ii. Fixed window may exhibit bias.
     iii. ASCII methods may have predictable structure.
   - **Mitigation:** can use multiple disjoint windows XORed, or feed entire H to a KDF (but only hashlib allowed).
     i. Use a large sample of H (≥ 32 bits).
     ii. Mix multiple blocks, apply XOR.
     iii. Add server secret for keyed hashing.

Rubric (25 marks)

Part A — Implementation & F (12.5)

Criterion	Max	TA Notes
Correct OTP generation + verification flow	3	Steps including date-time → SHA256 → store H → O=F(H) → verify.
Clear definition of F	3	Any deterministic H→4-digit mapping is acceptable if explained.

Justification for F choice	2	The student explains why F is simple, uniform, secure-ish, or convenient.
Code or pseudocode for F	2	Python or similar pseudocode demonstrating feasibility.
Edge-case handling	2.5	Zero-padding, single-use H, file read/write, deletion after success.

Part B — Vulnerabilities & Security (12.5)

Criterion	Max	TA Notes
Identifies ≥3 vulnerabilities clearly	4	Predictability, no secret, small OTP space, replay, file exposure, F bias.
Explains exploits	3	Attacker reads file, brute-forces time, guesses OTP.
Proposes strong mitigations	3	Secret-based hashing, expiry, rate-limiting, randomness, secure storage.
Discusses F-specific risks	1.5	Bias, entropy loss, predictable transformations.
Completeness & clarity	1	Reasonably structured explanation.

General Rubric Notes

- Multiple valid F exist. Accept any deterministic, well-justified mapping from H→4-digit numeric, with code/pseudocode and explanation.
- Penalize only if F is ambiguous, non-deterministic, or impossible to implement with `hashlib`.
- Major deduction if the student fails to discuss the `no-secret` issue or the ease of brute-forcing datetime-derived hashes.