NASA HW11

B11901164 陳秉緯

Security Part I

1. 三角準則的侵略者!?

ref: 1, 2, 3

(a)

1. 2017 年 WannaCry 勒索病毒事件，違反 I與A：
    - 可用性：WannaCry 勒索病毒利用 Windows 系統的漏洞，感染全球數十萬台電腦和伺服器，使受感染的系統無法使用，嚴重影響企業的生產力
    - 完整性：勒索病毒還會對受感染的硬碟和檔案進行加密，除非支付贖金，否則檔案無法恢復，這也違反了完整性原則
2. 2021 年 Facebook 當機，違反 A：
    - 2021 年 10 月，Facebook、Instagram、WhatsApp 等服務因 BGP 設定錯誤導致全球性斷線，服務長時間無法使用。雖然資料本身並未外洩，但使用者無法正常存取服務，導致「可用性」遭到破壞。這反映出即使資料未被竄改或外洩，只要服務無法被合法使用者使用，就已構成資安問題

(b)

- Assumption:
    1. 筆電作業系統支援用戶身份管理與密碼保護
    2. 使用者經常關機或上鎖筆電
    3. 筆電不會被攻擊者物理拆解或改裝
    4. 使用者沒有將密碼隨意寫在可見處
- Threat model:

Threat Model	Countermeasure
攻擊者在受害者背後偷看他輸入密碼以取得登入資訊	使用 2FA 來加強登入保護
攻擊者利用 USB 開機或進入 BIOS 設定嘗試繞過登入系統	設定 BIOS 密碼並關閉 USB 開機功能

(c/)

- Assumption:
    1. 每支手機 SIM 卡需實名制

2. 系統依賴電信商轉發的簡訊作為身份驗證依據
3. 掃描 QR code 行為為自願且無強制驗證機制
4. 使用者無法輕易更換 SIM 卡匿名傳送訊息

- Threat model:

Threat Model	Countermeasure
有人使用他人手機或未授權設備進行實聯制掃描與簡訊傳送	要求簡訊內含手機號碼與時間戳記， 並由電信業者驗證身分
有人透過自動化工具大量假冒身份掃描	建立異常模式偵測機制， 阻擋大量重複或異常傳送行為

(d)

- Assumption:
  1. 考試為實體小組考試，每組坐在一起，可以口頭討論
  2. 每位同學可使用自己的筆電或教室電腦進行作答
  3. 禁止使用任何通訊軟體，但沒有技術性限制或監控機制
  4. 唯一的監督機制為老師或助教會在教室中巡邏查看學生畫面

- Threat model:

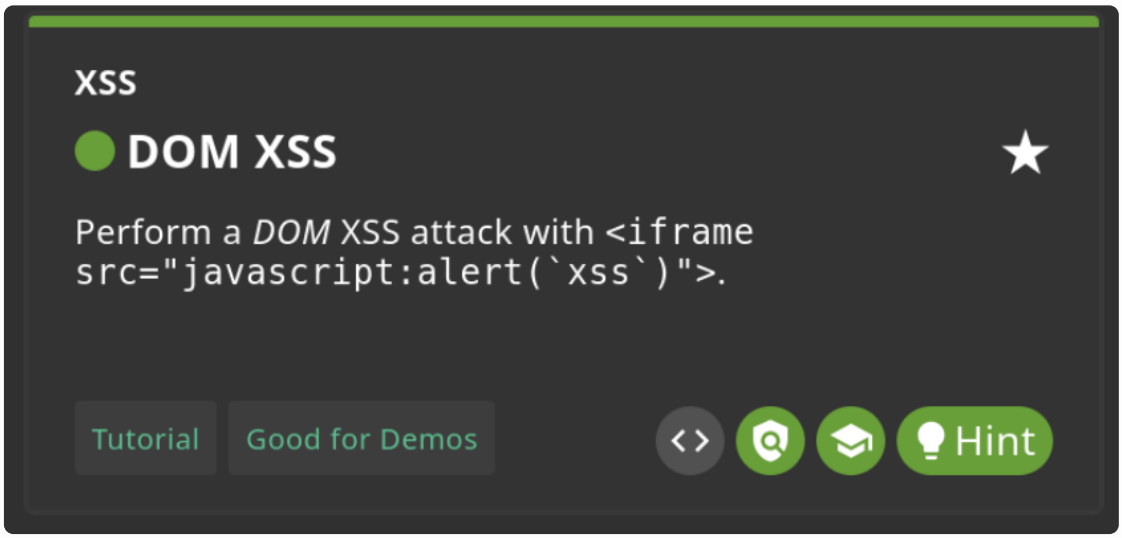
Threat Model	Countermeasure
使用背景通訊軟體偷偷傳訊息給外部人士	要求學生考試開始前開啟「任務管理員」或「活動監視器」， 由老師巡視時不定時要求展示背景執行程式， 或使用簡易監控工具列出開啟的通訊程式
使用個人手機偷偷的與外界聯絡	考試開始前統一規定所有手機需放入指定手機保管箱或集中收納 非經許可不得使用手機，避免偷偷通訊。

2. 果汁店也有洞！

ref: 1, 2, 3, 4, 5, 6, 7

(a)
(i) DOM XSS

- 截圖：



- 解題過程：

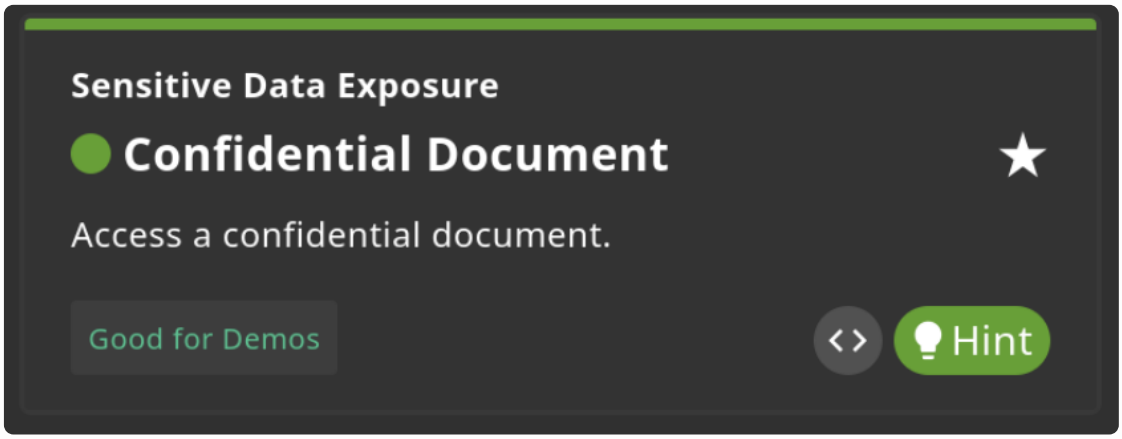
1. 在 `http://127.0.0.1:3000/#/search` 頁面上放搜尋框輸入：

```
<iframe src="javascript:alear(`xss`)">
```

- 漏洞類型：XSS
- 漏洞原理：DOM XSS 是一種跨站腳本攻擊，攻擊者利用網頁前端的 JavaScript 操作 URL 或頁面內容時，未正確過濾或轉義用戶輸入的資料，導致惡意腳本被執行。本題中，使用者輸入的字串直接被當作 HTML 或 JavaScript 片段插入到頁面中，例如在 iframe 的 src 裡，導致瀏覽器執行惡意程式碼，alert 彈跳視窗。此類漏洞不涉及伺服器端，而是純粹前端的 DOM 操作失誤。

(ii) Confidential Document

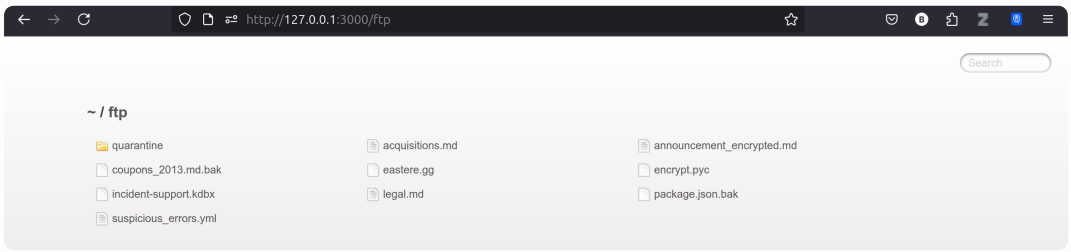
- 截圖：



- 解題過程：

1. Open side menu > About Us > Check out our boring terms of use if you are interested in such lame stuff，來到 `http://127.0.0.1:3000/ftp/legal.md` 頁面

2. 拿掉 URL 的 legal.md 並搜尋，看到很多檔案

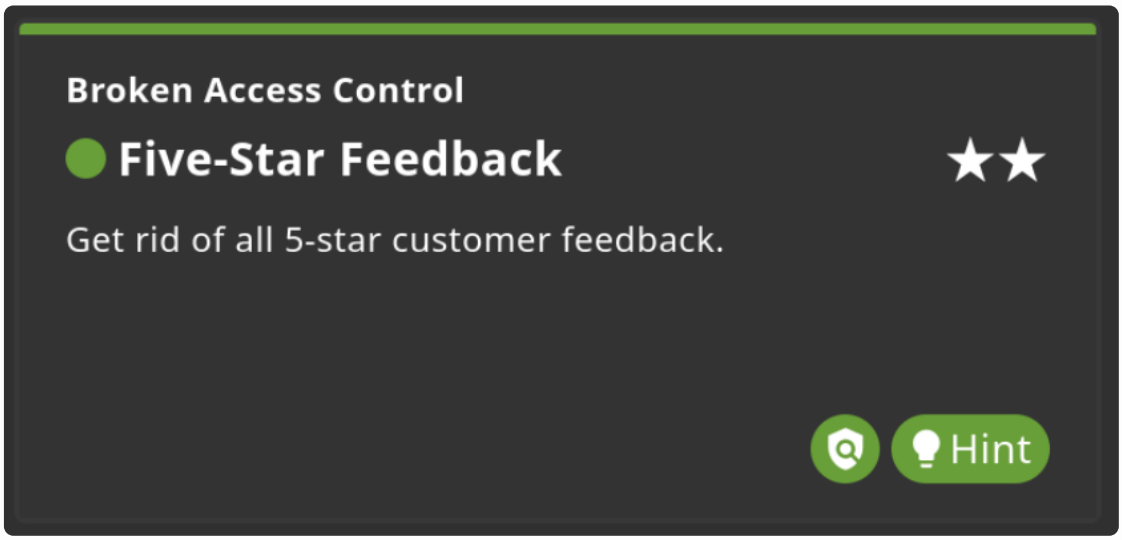


3. 點擊檔案 acquisitions.md 就完成此題

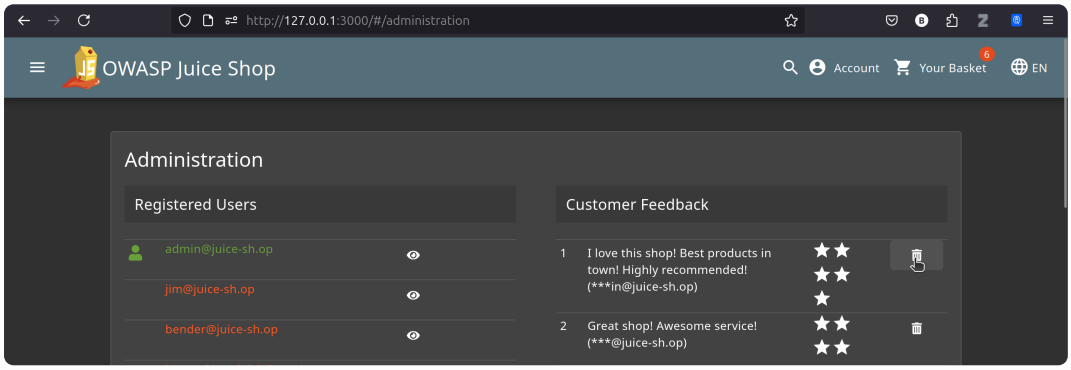
- 漏洞類型：Sensitive Data Exposure
- 漏洞原理：系統錯誤配置或設計缺陷，導致敏感資料如內部文件、帳戶資料等可被未授權用戶直接存取。本題利用 Juice Shop 內部 FTP 或靜態資源路徑可被直接瀏覽的弱點，使用者透過 URL 手動調整路徑，訪問到原本應保密的文件內容，造成機密文件外洩。

(iv) Five-Star Feedback

- 截圖：



- 解題過程：
  1. 利用助教上課教的 SQL Injection 登入 admin 帳號
  2. 將原本 URL 的 /search 改成 /administration 則能來到 Administration 頁面



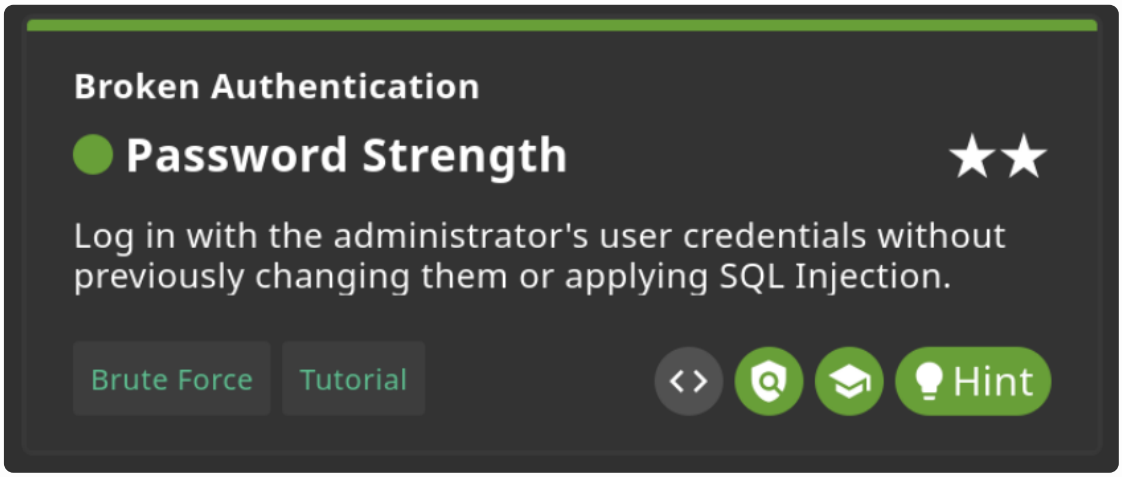
  3. 將右側五星 feedback 點擊垃圾桶圖示，則成功刪除 5-star customer feedback

- 漏洞類型：Broken Access Control

- 漏洞原理：使用者繞過系統的權限限制，取得非授權的操作能力。此題利用 SQL Injection 登入管理員帳號後，能進入管理員介面，執行刪除五星評論等管理行為，這表示系統未嚴格限制不同身份的操作權限，也沒有防範 SQL Injection 的攻擊。

(vi) Password Strength

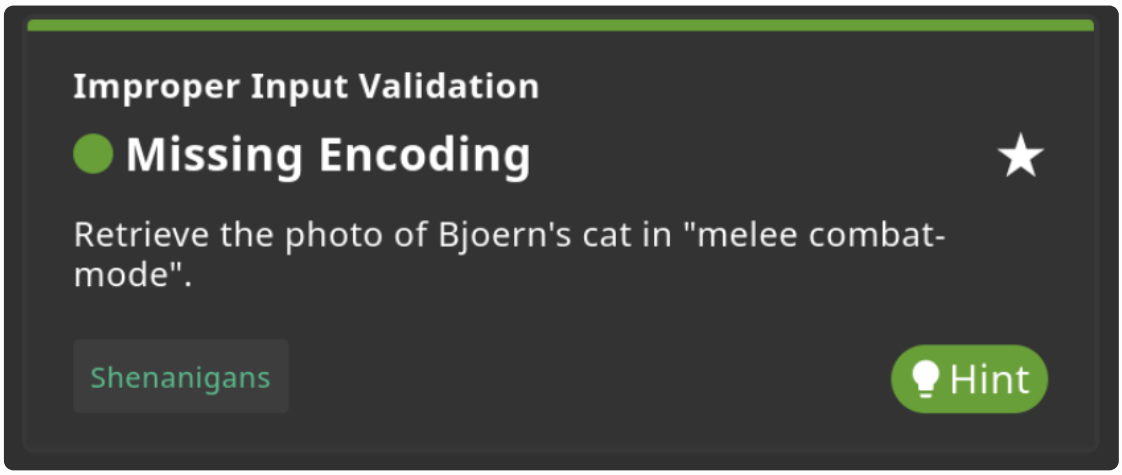
- 截圖：



- 解題過程：
  1. 參考 The top 20 admin passwords will have you facepalming hard 並試試看上面提到的 password
  2. 最後找到 admin 的 password 是 admin123
  3. 登入帳號 admin@juice-sh.op 密碼 admin123 則完成此題
- 漏洞類型：Broken Authentication
- 漏洞原理：系統在認證流程上存在缺陷，使得攻擊者能輕易取得或猜測帳號密碼，繞過身份驗證。此題中，管理員帳號使用弱密碼，如 admin123，未強制使用強密碼政策，導致可透過字典攻擊或常見弱密碼清單登入。

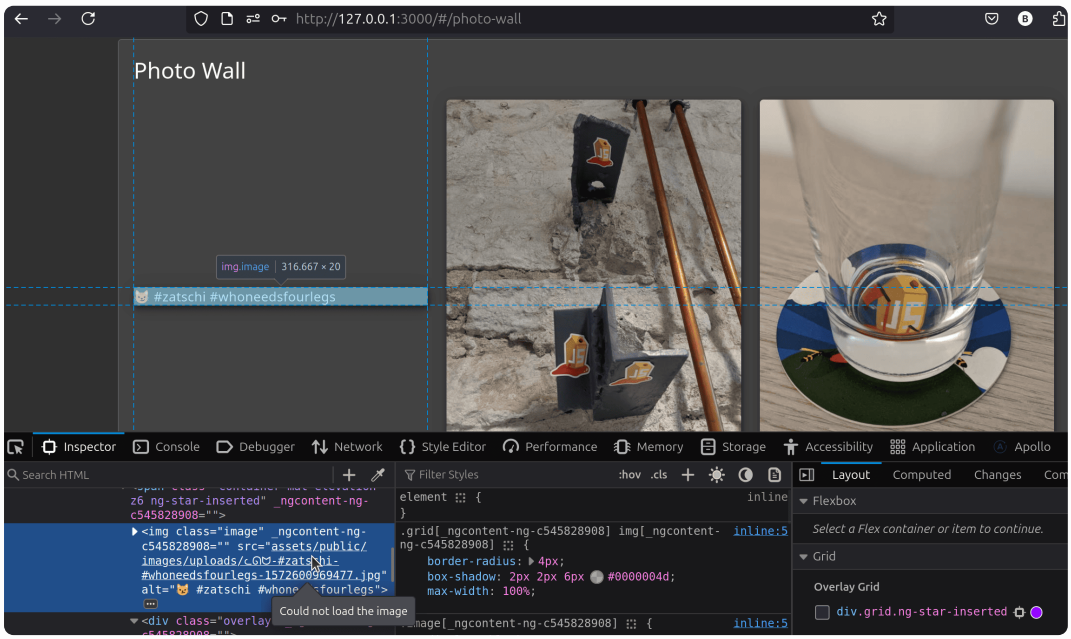
(viii) Missing Encoding

- 截圖：

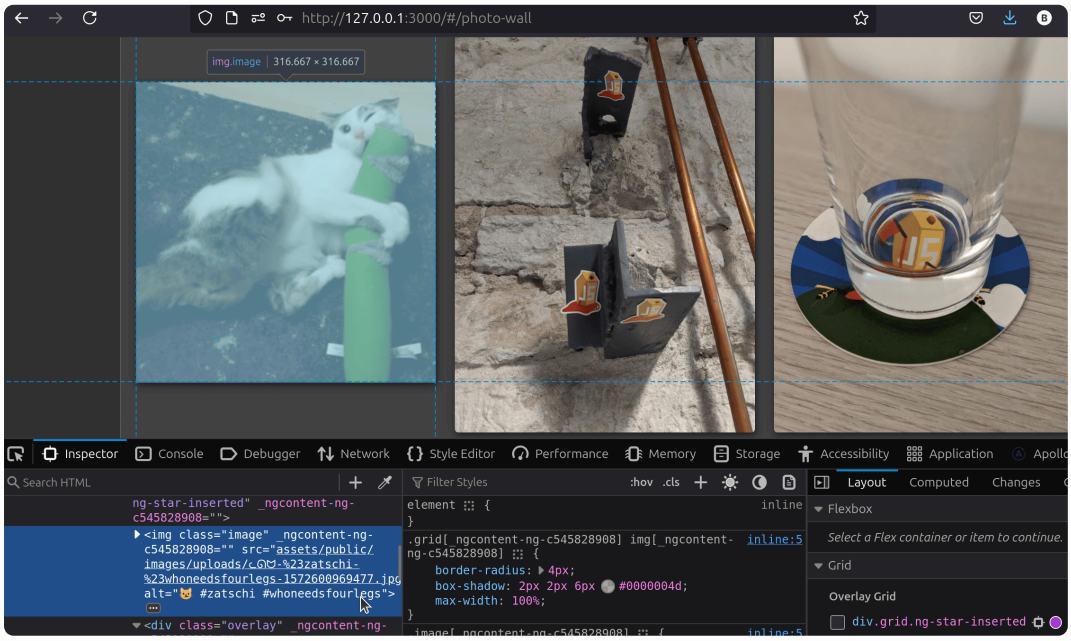


- 解題過程：
  1. Open side menu > Photo Wall 發現第一個圖片沒有正常顯示

2. f12 打開 Developer tools 去找 img



3. 修改 src，將 `#` 都改成 `%23` 後按 Enter 則出現圖片



- 漏洞類型：Improper Input Validation
- 漏洞原理：系統未正確處理輸入中的特殊字元，造成解析錯誤或無法正常顯示。此題中，圖片 URL 中的特殊字元 `#` 未經 URL Encoding，被瀏覽器視為 fragment identifier，導致圖片路徑錯誤，無法載入。透過將 `#` 改成 `%23`（URL encoded），圖片才正常顯示。

(b)

1. SSRF (Server-Side Request Forgery, 伺服器端請求偽造)：攻擊者誘使伺服器向內部或外部系統發送惡意請求，通常用於存取內部網路資源、繞過防火牆或掃描內網服務。
2. CSRF (Cross-Site Request Forgery, 跨站請求偽造)：利用使用者已登入網站上的身份，向該網站發送未授權請求，強迫使用者對目標網站執行動作，如改密碼、轉帳等。

3. XSS (Cross-Site Scripting, 跨站腳本攻擊)：攻擊者注入惡意腳本到網頁中，使其他使用者執行該腳本，竊取 Cookie 或篡改頁面內容。

- 差異：

特性	SSRF	CSRF	XSS
發生位置	伺服器端	使用者端觸發， 伺服器執行操作	瀏覽器端執行
攻擊目標	伺服器	使用者身份 （冒用其身份執行操作）	使用者 （通常是其他訪問者）
攻擊手段	操控伺服器請求內網/ 外部資源	利用使用者的 session 發送請求	插入並執行惡意 JavaScript
防禦方式	限制內部請求、 防火牆、URL 白名單	CSRF Token、SameSite cookie	輸入過濾、CSP 內容安全政策

3. R-SA！破密部

註：此題的 code 都寫在一個叫 `rsa.py` 的檔案內，在 `code` 資料夾內

ref: 1, 2, 3

(a)

- flag：NASA_HW11{blind_signing_is_dangerous}
- 攻擊過程：
  1. 從 `soyo.py` 拿到 RSA 公鑰 `(e, n)`
  2. 發現 soyo 拒絕簽署任何開頭為 `name=` 的訊息，但願意簽署其他任意訊息
  3. 建立目標訊息 `m_target = "name=soyo"`，並將其表示為兩段合法訊息相乘：`m_target = (m1 * m2) mod n`，其中 `m1` 是任意簡單的字串（如 `"hello"`），`m2` 為：`m2 = (m_target * inverse(m1, n)) mod n`
  4. 要求 soyo 分別簽署 `m1` 與 `m2`，取得簽章 `s1` 與 `s2`
  5. 利用 RSA 的 multiplicatively homomorphic 性質計算偽造簽章：`fake_signature = (s1 * s2) mod n`
  6. 使用 `ID = "name=soyo"` 和 `signature = fake_signature` 傳送給 anon 驗證成功，獲得flag: NASA_HW11{blind_signing_is_dangerous}
- 攻擊原理：
  - RSA 簽章為 `signature = m^d mod n` 而驗證者用：`m == signature^e mod n`，且 RSA 是 multiplicatively homomorphic：`sign((m1 * m2) mod n) == sign(m1) * sign(m2) mod n`
  - 因此，即使被禁止直接簽署 `name=soyo`，也可以透過將目標訊息分解為兩段合法訊息的乘積，再將兩個合法簽章相乘，偽造出對目標訊息的簽章

(b)

- flag: NASA_HW11{W0w_y0u_kNow_h@st@d'5_bro4dc@s7_47t@cK}
- 攻擊過程：
  1. 在 `anon.py` 內發現小指數漏洞，愛音使用小公鑰指數 `e = 7` 進行 RSA 加密，每次連接都產生新的 RSA 密鑰，但加密相同的日記內容
  2. 利用偽造簽名多次連接愛音系統來收集 8 個不同的 `(n, c)` 組合，且都使用相同的 `e = 7`
  3. 執行 Håstad's Broadcast Attack：使用中國剩餘定理從 7 個同餘方程式求解：`m^7 ≡ c_i (mod n_i)` 並計算得到 `m^7` 在整數域上的值（18299 bits），計算精確的 7 次方根，成功還原明文
  4. 將解密結果轉換為字串，發現愛音的秘密日記與其中的 flag
- 攻擊原理：Håstad's Broadcast Attack 利用了 RSA 在使用小指數時的數學弱點：只要收集到至少 `e` 個使用相同小指數的不同密文，當 `m^e < n_1 × n_2 × ... × n_e` 時，可使用中國剩餘定理在整數域上直接求解

4. TESTING in the FUZZ

ref: 1, 2, 3

(a)

面向	Mutation-based Fuzzing	Generation-based Fuzzing
測資產生方式	直接對現有合法輸入做隨機變異，如 bit flip、刪除、插入 byte 等，產生新測資	根據程式或協議的格式規則從零開始產生 通常需手動撰寫輸入格式
對資料格式的要求	不需要知道輸入格式， 不知道格式也可以	需要了解並定義輸入資料的語法，格式要
優點	自動化程度高、容易上手	對結構性輸入，如 JSON、XML 封包等測試效果佳

(b) greybox fuzzing 是一種利用部分程式內部資訊來指引 fuzzing 的方法，典型例子如 AFL。它會將目標程式透過特製的編譯器（如 afl-gcc）加入輕量級追蹤機制，執行每個測資後監控 coverage（執行路徑）。若某筆輸入產生新的程式路徑，便保留該輸入，進一步對其進行變異產生更多測資。這樣的回饋式循環設計，使 greybox Fuzzing 相較於 blackbox 更能有效探索程式空間，又不像 whitebox 那樣重度依賴符號執行與靜態分析。

(c) 先放棄

(d) 先放棄

5. 敗北協定太多了!

DNS Security

ref: 1, 2

(a)

- DNS amplification attack 是一種 DDoS 攻擊手法，攻擊者偽造受害者的 IP 向開放的 DNS 伺服器發送查詢，特別是會得到大回應的查詢，如 ANY 查詢，DNS 伺服器會把大量回應送給受害者，造成流量耗盡。
- 防範方法：
  - 關閉 Open Recursion：只允許可信賴的來源進行 DNS 查詢。
  - 啟用 Response Rate Limiting：限制單一來源查詢的回應頻率，降低被濫用的機會。

(b)

- 攻擊者向 DNS 伺服器注入偽造的回應記錄，例如錯誤的 IP，使得使用者查詢某個網域時會被導向攻擊者控制的伺服器。
- 防範方法：
  - 使用 DNSSEC：簽署 DNS 回應資料以驗證其真實性，防止偽造。
  - 增加不可預測性：如隨機化查詢的 source port 和 DNS ID，使得攻擊者更難猜中正確組合。

SMTP Security

ref: 1, 2

(c)

- SPF 是一種用於驗證寄信伺服器是否被授權發送該網域郵件的 DNS 機制。
- 如何防止 email spoofing：當接收郵件伺服器收到信時，會去查詢發信網域的 SPF 記錄，確認該 IP 是否被授權。如果不符合，根據網域設定，信件會被標記為失敗或拒收。

(d)

- DKIM 透過數位簽章驗證郵件內容與寄件者是否未被竄改。
- 如何防止 email spoofing：寄信伺服器使用私鑰對郵件某些 header 和 body 做簽名，接收者可從 DNS 拿公鑰驗證簽名是否有效。若有效，表示郵件在傳輸過程中未被更改，且來自該網域。

(e)

- 否
- email spoofing 手法：UI-mismatch Attacks：利用 email server 與 email client 對 From 標頭的解讀不一致，導致 client 顯示錯誤的寄件人。舉例來說，寄出含兩個 From 標頭的郵件，Server 驗證第一個但 Client 顯示第二個，則能達成 email spoofing。

TLS Security

ref: 1, 2, 3, 4

(f)

- certificate 裡會有什麼內容
    1. 網域名稱
    2. 憑證認證機構
    3. 憑證認證機構的數位簽章
    4. 簽發日期
    5. 到期日期
    6. 公開金鑰
    7. SSL/TLS 版本
- 什麼是 CA：是憑證授權機構，是值得信賴的第三方，負責簽發並驗證憑證。用戶端（如瀏覽器）內建信任的 CA 清單，若憑證由可信 CA 簽發，即視為合法。

(g)

- 攻擊者攔截初始 HTTP 請求並阻止其升級為 HTTPS，例如刪除 Strict-Transport-Security header，讓用戶停留在不安全的 HTTP 連線，便於竊聽或中間人攻擊。
- 防範方式：
    - 啟用 HSTS：讓瀏覽器記住此網站只能使用 HTTPS。
    - 自動 redirect HTTP -> HTTPS 並關閉 HTTP 功能。

6. 猫物語（赤）

註：此題的 code 都寫在一個叫 `main.py` 的檔案內，在 `code` 資料夾內

ref: 1, 2, 3, 4, 5

(a)

- FLAG1: NASA_HW11{pseudorandomness_does_not_guarantee_unpredictability}
    - 漏洞分析：

      ```
      def get_random(self):
          self.state = (self.state * self.a + self.c) % self.m
          return self.state
      ```

      在 `fatcat.py` 第 22 到第 24 行內，發先這個是 linear congruential generator，而他是是一種 pseudo 隨機數生成器，因此可以進行以下攻擊
    - 攻擊原理：
        - 由上面發現這題產生亂數的方法是用 linear congruential generator，而他是一種 pseudo 隨機數生成器，其公式為：`x_{n+1} = (a * x_n + c) mod m`
        - 因此如果我們能觀察到連續的幾個輸出值，就可以使用 modular inverse 推出 `a` 和 `c`，從而預測後續的所有隨機數。
    - 解題過程：
        1. 連續選擇選項 1 三次，透過連續猜錯三次來獲得三個連續的 LCG 狀態值

2. 使用已知的三個連續輸出 `s0`, `s1`, `s2` 與 modular inverse 求出 `a` 和 `c`：
    - 已知：`s1 = (a * s0 + c) mod m` 與 `s2 = (a * s1 + c) mod m`
    - 可以推導出 `s2 - s1 = a * (s1 - s0) (mod m)`
    - 因此 `a = (s2 - s1) / (s1 - s0) (mod m)` 與 `c = s1 - a * s0 (mod m)`
3. 解出 `a` 與 `c` 後就可以預測之後的續隨機數，一直重複做到 `trust` 等於 100
4. 最後選擇選項 2 拿到 FLAG1

- 程式實做關鍵的部份：

```
def crack_lcg(s0, s1, s2, m):
    numerator = (s2 - s1) % m
    denominator = (s1 - s0) % m

    if denominator == 0:
        return None, None

    try:
        a = (numerator * pow(denominator, -1, m)) % m  # modular inverse
        c = (s1 - a * s0) % m
        return a, c
    except:
        return None, None
```

- FLAG2: NASA_HW11{07p_k3y_mu57_b3_47_l3457_45_l0n6_45_pl41n73x7}
    - 漏洞分析：

      從 `fatcat.py` 中的 `OTPEncrypt` function 可以看到：

```
def OTPEncrypt(msg: bytes) -> bytes:
   key_len = 10  # 密鑰長度為 10 字節
   key = secrets.token_bytes(key_len)  # 每次生成新的 10 字節隨機密鑰
   enc = bytes(msg[i] ^ key[i % key_len] for i in range(len(msg)))  # 密
鑰循環使用
   return enc
```

      漏洞點：

      1. 密鑰長度固定為 10 字節且循環使用 (`key[i % key_len]`)
      2. 明文長度遠大於密鑰長度
      3. 作業開頭有說 flag 的格式，所以明文具有可預測的格式 (`NASA_HW11{...}`)

    - 攻擊原理：
        - XOR加密的可逆性：`K[i] = C[i] XOR P[i]` ，已知密文C和明文P可求密鑰K
        - 密鑰循環約束：對所有 `i = j (mod 10)` ，必須 `K[i mod 10] = K[j mod 10]`
        - 已知明文 `NASA_HW11{` 長度為 10 字節，恰好等於密鑰循環長度 10 字節，這意味著如果我們找到正確的起始位置，可以一次性恢復完整的 10 字節密鑰

    - 解題過程：

1. 連線後選擇選項 3 獲取加密的考試結果
2. 提取 16 進制格式的密文
3. 系統性地嘗試每個可能的 `NASA_HW11{` 起始位置（0-93，共 94 個位置）
4. 對每個候選位置通過 XOR 運算推導對應的密鑰片段
5. 檢查推導出的密鑰片段在 10 字節循環中是否一致
6. 所有 94 個位置都通過了一致性檢查並完整恢復了 10 字節密鑰
7. 對每個候選密鑰進行解密測試，要求 100% 可列印字符且包含有效 FLAG
8. 成功在位置 47 找到正確解密，得到完整明文和 FLAG2

- 程式實做關鍵的部份：

```
def bruteforce_repeating_key_xor_flag(encrypted_hex):
    encrypted = unhexlify(encrypted_hex)
    known_plaintext = b"NASA_HW11{"
    candidates = []

    for start_pos in range(len(encrypted) - len(known_plaintext) + 1):
        partial_key = {}
        consistent = True

        # 核心密鑰推導邏輯
        for i in range(len(known_plaintext)):
            cipher_pos = start_pos + i
            key_pos = cipher_pos % 10  # 計算在10字節循環中的位置
            key_byte = encrypted[cipher_pos] ^ known_plaintext[i]  # XOR
逆推密鑰

            # 嚴格一致性驗證
            if key_pos in partial_key:
                if partial_key[key_pos] != key_byte:
                    consistent = False
                    break
            else:
                partial_key[key_pos] = key_byte

        if consistent:
            key = [partial_key[i] for i in range(10)]
            result = test_key_and_decrypt(encrypted, key)
            if result:
                candidates.append(result_data)

    return candidates

def test_key_and_decrypt(encrypted, key):
    try:
        decrypted = bytes(encrypted[i] ^ key[i % 10] for i in
range(len(encrypted)))
        decoded = decrypted.decode('ascii', errors='replace')

        # 嚴格條件
        printable_ratio = sum(1 for c in decoded if c.isprintable()) /
len(decoded)

        # 必須 100% 可列印且包含 FLAG 格式
        if printable_ratio == 1 and 'NASA_HW11{' in decoded:
            flag_match = re.search(PATTERN, decoded)
```

```
            if flag_match:
                return flag_match.group(0), decoded, printable_ratio

        return None
    except:
        return None
```

- FLAG3: NASA_HW11{https://youtu.be/1GxwDuV5JMc}
  - 漏洞分析：
    從 `fatcat.py` 中的 `proof_of_work` 和 `do_many_PoW` 函數可以看到：

```
def proof_of_work():
    r = random.randint(0, 2**24 - 1)  # 搜索空間固定為 2^24
    h = hashlib.md5(str(r).encode()).hexdigest()[0:8]  # 只取 MD5 前 8 位
    i = input(f'Give me `i` such that md5(i)[0:8] == "{h}" : ').strip()
    if hashlib.md5(str(i).encode()).hexdigest()[0:8] != h:
        print('Wrong')
        exit()

def do_many_PoW():
    n = 10
    rate = n / timeit(proof_of_work, number=n)  # 計算解題速率
    rate_str = str(rate)
    print(f'Wow! You can solve {rate_str} PoWs per second!')
    if rate > 150:  # 要求速度超過150 PoW/秒
        print(f'Thanks for helping out with the work! Please accept this
flag as thanks: {FLAG3}')
```

    漏洞點：

    1. 搜索空間有限且固定 (2^24 = 16,777,216 種可能)
    2. 只需匹配 MD5 hash 的前 8 位，碰撞空間相對較小
    3. 要求極高的解題速度 (150 PoW/s)，代表需要先計算好而非使用暴力破解
    4. 每次挑戰的目標都在已知的有限集合內

  - 攻擊原理：
    與其每次都進行暴力搜索，不如預先建立 rainbow table。由於搜索空間固定為2^24，我們可以預先計算所有可能值的 MD5 hash 前 8 位，建立從 hash 到原值的映射表。這樣每次 PoW 挑戰都能在 O(1) 時間內完成查找，很容易解題速度可以超過 150 PoW/s。所以，對所有 i ∈ [0, 2^24-1]，計算 `hash_table[md5(str(i))[:8]] = i` 然後給定目標 hash `h`，直接返回 `hash_table[h]`
  - 解題過程：
    1. 連線後選擇選項 4 開始 PoW 挑戰
    2. 在挑戰開始前預先建立完整的 rainbow table
    3. 遍歷 0 到 2^24-1 的所有整數，計算每個數字的 MD5 hash 前 8 位
    4. 建立從 8 位 hash value 到原整數的映射字典
    5. 接收對方發送的 10 個 PoW 挑戰
    6. 對每個挑戰，解析目標 hash value，直接從 rainbow table 中找出對應的原值並立即發送答案給對方

7. 完成所有 10 個挑戰後，系統計算解題速率
8. 因為查找時間接近 0，速率超過 150 PoW/s，最終獲得 FLAG3

- 程式實做關鍵的部份：

```
def solve_part_c(conn: remote):
    # 策略：預先建立完整 rainbow table
    print("Building comprehensive rainbow table...")
    rainbow_table = {}
    for i in range(2**24):  # 遍歷完整搜索空間
        hash_val = hashlib.md5(str(i).encode()).hexdigest()[:8]
        rainbow_table[hash_val] = i  # 建立 hash 到原值的映射

    print(f"Rainbow table built with {len(rainbow_table)} entries")

    conn.recvuntil(b"Your choice: ")
    conn.sendline(b"4")

    # 處理 10 個 PoW 挑戰，瞬間查找
    for pow_num in range(10):
        # 接收挑戰並解析目標 hash
        line = conn.recvuntil(b": ").decode()

        if 'md5(i)[0:8] == "' in line:
            start_idx = line.find('md5(i)[0:8] == "') + len('md5(i)[0:8]
== "')
            end_idx = line.find('"', start_idx)
            target_hash = line[start_idx:end_idx]

            # 只要 O(1) 時間查找，不需要當場計算，直接從 rainbow table 中找出對應
的原值
            solution = rainbow_table.get(target_hash)

            if solution is not None:
                conn.sendline(str(solution).encode())
            else:
                return False  # 理論上不會發生，但還是寫一下
```

- `for i in range(2**24):` 預先遍歷整個搜索空間，建立 rainbow table，一次性計算完來換取後續的高速查找
- `rainbow_table[hash_val] = i` 建立反向映射，從 hash value 快速找到原始數值
- `solution = rainbow_table.get(target_hash)` O(1) 字典查找替代 O(n) 暴力搜索，這是達成 150+ PoW/s 的原因
- rainbow table 大小：2^24 ≈ 1677 萬條記錄，占用約 100-200 MB 記憶體，是可接受的空間成本
- 建表需要大約 3 分鐘，但每次查找只要幾毫秒，10 個挑戰總共花可能不到 1 秒，可以遠遠超過解題速度要求，因此成功拿到 FLAG3

- FLAG4: NASA_HW11{y0u_KN0w_r3F13C710n_4774cK}

- 漏洞分析：

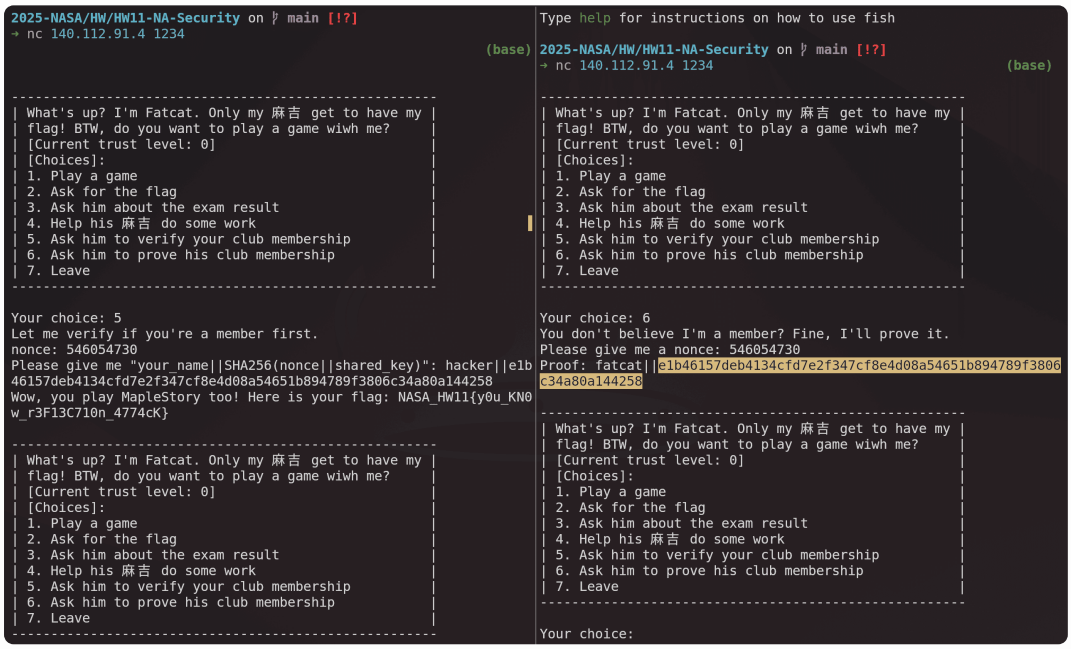
```
def verifier():
    nonce = str(random.randint(0, 2**32-1))
    print(f'nonce: {nonce}')
```

```
    response = input('Please give me
"your_name||SHA256(nonce||shared_key)": ').strip()
    try:
        v = response.split('||')
        name = v[0]
        mac = v[1]
    except IndexError:
        print("\nInvalid\n")
        exit()

    if name == 'fatcat':
        print("You can't be me! You IMPOSTOR!")
        exit()

    if hashlib.sha256(f'{nonce}||
{club_shared_key}'.encode()).hexdigest() != mac:
        print('You are not a member!')
        exit()

def prover():
    try:
        nonce = int(input('Please give me a nonce: ').strip())
    except ValueError:
        print("\nInvalid\n")
        exit()

    name = 'fatcat'
    mac = hashlib.sha256(f'{nonce}||
{club_shared_key}'.encode()).hexdigest()
    response = f'{name}||{mac}'
    print(f'Proof: {response}')
```

在 `fatcat.py` 第 97 到第 128 行內，發現到我要成功被 verified 的方法就是我的 `name` 不能是 `fatcat`，而且後面的算 `SHA256` 也沒有用到 `name`，代表說我的可用除了 `fatcat` 以外的任意名字，另外，我的 `mac` 必須跟他用的 `nonce` 與 `club_shared_key` 算出來的一樣，但是，我只能知道他用的 `nonce`，我沒辦法知道 `club_shared_key`，但是在觀察 `prover()` 就發現到他的 input 是 `nonce`，他幫我們用 `nonce` 與 `club_shared_key` 算 `mac` 給我，所以就可以進行以下攻擊：

- 攻擊原理：正常的 MAC 應該包含所有需要認證的數據，`name`， `nonce`，與 `shared_key` 等等，因為這題 MAC 沒有包含用戶名，我們可以進行 reflection attack
- 解題過程：
    1. 選擇選項 5，讓他執行 `verifier()` 並給我 `nonce`
    2. 再開另外一個連線，選擇選項 6，並給他剛剛拿到的 `nonce`，這樣他就會給我他算的 `mac`
    3. 在切到第一個連線，給他 `{name}||{mac}`，這裡 `name` 除了 `fatcat` 外都可以，`mac` 就是從第二個連線拿到的 `mac`，這就相當於叫 `prover()` 幫我算 `verifier()` 要的答案後給我
    4. 送出後就拿到 FLAG4 了

- 直接用 `nc 140.112.91.4 1234` 在 terminal 解題：



先在左邊輸入 5 得到 `nonce`，然後右邊新的連線輸入 6 並把 `nonce` 餵給他，然後拿到 proof 的 `mac`，最後再貼回到左側，前面加上 `{name}||`，`name` 除了 `fatcat` 之外都可以，好了之後送出就拿到 flag 了，另外也有用 python 來實做