DOKUMENTACJA KOŃCOWA PROJEKTU BEMSI

Szymon Spaczyński, Anastasiya Dastanka

Temat projektu

Tematem projektu jest szablon aplikacji webowej wirtualnej przychodni z minimalnym wykorzystaniem cookies.

Opis projektu

W ramach projektu wykonany został prototyp aplikacji mogącej być wykorzystywanej w wirtualnej przychodni. Podzielona jest ona na dwa główne moduły – notatki medyczne pacjenta oraz wizyty pacjentów u lekarza.

Notatki medyczne są wypisywane przez lekarza i przypisywane konkretnemu pacjentowi. Wizyty są zamawiane przez pacjenta, który wybiera przy tym konkretnego lekarza, do którego chce się umówić. Ponadto z kalendarza wybiera datę i godzinę wizyty.

W projekcie zaimplementowano mechanizm logowania użytkowników. Każdy użytkownik może mieć jedną z trzech ról – pacjent, lekarz, personel medyczny. Każda z tych ról ma przypisane określone funkcjonalności, z których może korzystać, tak jak opisano w tabeli poniżej:

Funkcjonalność/Rola	**pacjent**	**lekarz**	**personel medyczny**
Dostęp do notatek medycznych	TAK, dostęp do notatek wystawionych na siebie	TAK, dostęp do notatek wypisanych przez siebie	NIE, brak dostępu do żadnych notatek
Dodanie nowej notatki	NIE	TAK	NIE
Wyświetlenie umówionych wizyt	TAK, dostęp do wizyt umówionych przez siebie	TAK, dostęp do wizyt, które pacjent umówił u danego lekarza	TAK, dostęp do wizyt umówionych przez wszystkich pacjentów u wszystkich lekarzy
Anulowanie wizyty	TAK, możliwość anulowania umówionej przez siebie wizyty	TAK, możliwość anulowania wizyty umówionej u danego lekarza	TAK, możliwość anulowania dowolnej wizyty zarejstrowanej w systemie

Aplikacja składa się z dwóch części – backendu i frontendu. Backend został napisany w Pythonie w języku Django, zaś frontend w języku JavaScript z użyciem frameworka Vue 3. Oba te moduły komunikują się ze sobą za pomocą API wystawionego przez aplikację backendową. Przy tych połączeniach wykorzystywany jest token JWT, o czym nieco więcej będzie wspomniane w dalszej części dokumentu.

Analiza ryzyka i przyjętych zabezpieczeń

Podstawowe zagrożenia w tego typu systemach dotyczą kwestii uwierzytelnienia użytkownika. Poniżej opisano najważniejsze z nich wraz ze sposobem ich rozwiązania.

Zbyt proste hasło

Ustawienie przez użytkownika zbyt krótkiego i prostego hasła sprawia, że atakującemu łatwo będzie je przechwycić poprzez atak *brute force* lub używając listy najczęściej używanych haseł. Jednakże wymaganie od użytkownika podania hasła bardzo złożonego, zawierającego znaki specjalne itp. sprawiłoby, że ustawiane hasła koniec końców byłoby proste dla nich do zapamiętania a tym samym dość proste do odgadnięcia.

W naszym projekcie zaimplementowaliśmy dwie stosunkowo proste reguły walidacyjne – hasło musi zawierać co najmniej 10 znaków i nie może znajdować się na liście 100 000 najbardziej powszechnych haseł i popularnych kombinacji klawiaturowych. Listę tych haseł pobraliśmy z pliku: https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/100k-most-used-passwords-NCSC.txt

Poza tym nie stosujemy żadnego innego wymagania dotyczącego posiadania wielkich i małych liter, cyfr czy znaków specjalnych. 10 znaków to w naszej ocenie rozsądny kompromis między łatwością zapamiętania hasła a trudnością jego złamania. Dłuższe hasła z pewnością byłyby trudniejsze obliczeniowo do złamania przy użyciu ataku *brute force*, jednak użytkownicy prawdopodobnie stosowaliby wówczas prostsze ciągi znaków, aby ich nie zapomnieć.

Wykradzenie bazy danych wraz z hasłem użytkownika

Użytkownicy oraz ich dane uwierzytelniania oczywiście są przechowywane w bazie danych aplikacji. Gdyby ktoś wykradł bazę danych, mógłby przejąć te dane. Dlatego ważne jest, by nie trzymać haseł w formie jawnego tekstu, lecz odpowiednio je szyfrować. Użycie w tym celu zwykłej funkcji skrótu może jednak być niewystarczające, ponieważ każdemu ciągowi znaków odpowiadałby zawsze ten sam skrót. Z tego powodu lepsze jest wykonanie logicznej operacji bitowej między zahashowanym hasłem i pewnym losowo wygenerowanym ziarnem, ewentualnie także dodanie do niego stałego ciągu znaków, tzw. *pepper*. W naszym projekcie wykorzystujemy wspomniane trzy sposoby, wykonując je dodatkowo pewną liczbę razy, co znacznie utrudnia rozszyfrowanie i odgadnięcie haseł. Przyjęty w naszej aplikacji schemat szyfrowania hasła podczas rejestracji użytkownika jest następujący:

1. Do hasła doklej stały ciąg znaków, tzw. *pepper* – „bemsi projekt”

2. Wygeneruj unikalne losowe ziarno, tzw. *salt*, zapisz je w bazie danych w rekordzie z nowo tworzonym użytkownikiem
3. Trzykrotnie zahashuj hasło z pieprzem używając uprzednio wygenerowanego ziarna
4. Otrzymany ciąg znaków zapisz w bazie danych jako hasło

Do generowania ziarna oraz hashowania hasła została wykorzystana biblioteka języka Python o nazwie *bcrypt*. Dzięki temu algorytmowi szanse na odgadnięcie hasła po wykradzeniu bazy danych znacząco zmaleją. Dodatkowo użycie pieprzu oraz wykonanie algorytmu kilkukrotnie sprawia, że posiadanie samego tylko ziarna – które atakujący zobaczy w wykradzionej bazie danych – będzie niewystarczające do odgadnięcia jego hasła.

```
salt = bcrypt.gensalt()
password = passwd + pepper
hashed = password.encode('utf-8')
for i in range(3):
    hashed = bcrypt.hashpw(hashed, salt)
```

Kod źródłowy realizujący hashowanie hasła

Ów algorytm należy powtarzać kolejno przy każdym logowaniu użytkownika aby sprawdzić poprawność jego hasła. Z tego powodu nie może to być zbyt złożona obliczeniowo operacja.

Autoryzacja dostępu do zapytań API

Wszystkie zapytania API, z wyjątkiem logowania i rejestracji konta, wymagają umieszczenia w nagłówku zapytania HTTP tokena JWT. Każdy token zawiera informację o loginie oraz roli użytkownika, a także czas jego ważności. Tokeny te jednak nie są umieszczane w jawnej formie – wówczas atakujący bez problemu mógłby się podszyć pod użytkownika i np. używając polecenia *curl* wykonać w jego imieniu pewną operację w systemie. Każdy token jest podpisywany przez serwer zanim trafi do klienta. Odbywa się to z wykorzystaniem rekomendowanego w tym celu algorytmu RS256 i architektury PKI. Podczas logowania użytkownika wygenerowany obiekt JSON jest podpisywany przy użyciu klucza prywatnego i w takiej formie trafia do aplikacji frontendowej. Gdy zaś aplikacja frontendowa przekazuje później ów token przy zapytaniu HTTP, jest on odszyfrowywany przy użyciu klucza publicznego, a następnie weryfikacji podlegają informacje w nim zawarte, jak login czy rola. Zastosowanie architektury PKI sprawia ponadto, że klucze będą trudniejsze do złamania. Zastosowano przy tym klucze SSH wygenerowane przy użyciu komendy *ssh-keygen -t rsa*. Klucz prywatny dodatkowo zaszyfrowano hasłem, aby jeszcze zwiększyć jego bezpieczeństwo. Taka architektura

sprawia, że każdy – mając klucz publiczny – może odczytać zawartość tokena i zdobyć niezbędne dane o użytkowniku, jednak aby wygenerować taki token potrzebny jest klucz prywatny, utajniony. Oba klucze są przechowywane w plikach na serwerze. Wiąże się z tym oczywiście ryzyko wykradzenia plików wraz ze zdobyciem dostępu do serwera, jednak nie za bardzo można z tym cokolwiek zrobić – gdzieś owe klucze należy trzymać. Dodatkowo, nawet gdyby klucz został skompromitowany, można w krótkim czasie podmienić pliki, co sprawi że w jednym momencie wszystkie wygenerowane uprzednio tokeny stracą ważność.

Do generowania, podpisywania i odszyfrowywania tokenów użyta została biblioteka *PyJWT* w połączeniu z biblioteką *cryptography*.

Przechowywanie tokena JWT, ochrona przed wykradzeniem

W naszym projekcie nie używamy i nie przechowujemy w przeglądarce identyfikatorów sesji. Zamiast tego używamy tokenów JWT. Tokeny te są przekazywane w nagłówkach zapytań HTTP w postaci *Authorization: Bearer <token>*, stąd potrzeba przechowywać je w przeglądarce w *localStorage* lub *sessionStorage*. Trzymanie danych w *sessionStorage* sprawiłoby, że dane zostałyby utracone po zamknięciu okna przeglądarki, dlatego zdecydowaliśmy się przechowywać je w *localStorage* przeglądarki. Jednak w wyniku tego są one narażone na atak XSS, albo na przechwycenie i wykonanie requesta poleceniem *curl*. Z tego powodu każdy token ma swój czas ważności, po którym nie jest już uznawany przez serwer za sposób identyfikacji. W momencie, w którym do serwera zostanie wysłany nagłówek z przedawnionym tokenem, serwer zwraca odpowiedź „JWT Token expired”, wtedy po stronie frontendu generowane jest żądanie o odświeżenie tokenu. My zastosowaliśmy czas ważności tokena równy 1 minuta, aby znacząco utrudnić atak XSS, a jednocześnie zapobiec nadmiernemu obciążeniu serwera ciągłymi zapytaniami o odświeżenie tokena.

Odświeżenie tokena

Wygaśnięcie ważności tokena nie powinno jednak oznaczać konieczności ponownego logowania się na konto, gdyż byłoby to bardzo uciążliwe dla użytkownika. Typowym rozwiązaniem tego problemu jest zastosowanie oprócz zwykłego tokena również tzw. *refresh token*. Jest on wysyłany na serwer podczas zapytania o odświeżenie tokena, i jeśli zostanie on poprawnie zweryfikowany, serwer odpowiada nowym tokenem oraz refresh tokenem który po kolejnej minucie zostanie ponownie użyty do odświeżenia tokena. W naszym projekcie pod względem strukturalnym token refresh różni się tym od zwykłego tokena, że zawiera dodatkowe pole *refresh*, którego wartość jest ustawiona na True.

Bardzo istotna tutaj jest kwestia przechowywania refresh tokena. Token ten powinien mieć znacznie dłuższy czas ważności od zwykłego – w naszym przypadku ma on czas ważności 24 godziny – dlatego powinien być on bardziej odporny na atak XSS, czyli na dostęp z poziomu języka JavaScript. Dlatego w naszym projekcie token ten jest

przekazywany z serwera jako ciasteczko w momencie logowania konta i odświeżania tokena, z ustawionymi flagami Secure i HttpOnly. Dzięki temu token będzie niedostępny nawet przy użyciu instrukcji języka JavaScript „document.cookie”.

Oczywiście, w dalszym ciągu będzie on widoczny w pamięci przeglądarki, bowiem gdzieś przechowywać go należy. Z tego powodu należy wprowadzić dodatkowe zabezpieczenie, jakim jest jednorazowość tokena do odświeżenia. W naszym projekcie zrealizowaliśmy to jako dodatkową tabelę w bazie danych, w której przechowywane są wszystkie zblacklistowane refresh tokeny. W momencie próby odświeżenia tokena system weryfikuje, czy refresh token, który frontend wysłał poprzez cookie, nie jest już unieważniony, a jeśli jest, nie zezwala na przedłużenie ważności tokena. Po udanym odświeżeniu tokena, a także po wylogowaniu się użytkownika, refresh token jest zapisywany na blackliście w bazie danych, i nie będzie już mógł zostać ponownie użyty.

Przechowywanie tokenów po stronie serwera

Oprócz przechowywania tokenów po stronie klienta, można by je także przechowywać na serwerze, np. w bazie danych. Rozwiązanie to miałoby wiele zalet – można by np. w prosty sposób unieważnić wszystkie tokeny wystawione na jednego użytkownika, dzięki czemu, gdyby próbował zalogować się do systemu z wielu urządzeń naraz, można by uniemożliwić korzystanie z systemu gdyby wylogował się choćby w jednym miejscu. Jednakże, trzymanie tokenów w bazie rodzi ryzyko wykradzenia ich w momencie wycieku danych z bazy, ponadto scentralizowana usługa takich tokenów nieco przeczy ich pierwotnemu przeznaczeniu. Koniec końców, zagrożenie związane z zalogowaniem się do systemu z kilku miejsc naraz jest stosunkowo niewielkie, tym bardziej że w naszym systemie użyty jest mechanizm *idle*, który nieco dokładniej zostanie opisany w dalszej części dokumentu. Z tego powodu nie zdecydowaliśmy się przechowywać ani tokenów zwykłych, ani refresh tokenów (poza tymi umieszczonymi na blackliście) po stronie serwera. Gdybyśmy jednak tworzyli system o jeszcze wyższym poziomie krytyczności, np. system bankowy, warto byłoby takie rozwiązanie rozważyć.

Pozostawienie zalogowanego systemu

Może się wydarzyć również sytuacja, w której użytkownik pozostawi komputer z zalogowanym kontem i wyjdzie np. na obiad. W takiej sytuacji, jeśli dostęp do komputera będzie otwarty, ktoś niepowołany będzie w stanie w imieniu tego użytkownika wykonywać pewne akcje w systemie. Aby się przed tym zabezpieczyć, zaimplementowaliśmy w projekcie mechanizm *idle*, który wylogowuje użytkownika, tym samym kasując przechowywane w przeglądarce tokeny, gdy ten przez 15 minut pozostaje nieaktywny w tym sensie, że nie rejestrowany jest ruch myszy. Do sprawdzenia aktywności wykorzystaliśmy bibliotekę *v-idle* przeznaczoną do frameworka Vue. Oczywiście jeżeli w ciągu tych 15 minut ktoś niepowołany uzyska dostęp do komputera, będzie w stanie w jego imieniu korzystać z systemu. Jednak my, jako deweloperzy systemu, nie możemy nic na to poradzić – pozostaje zaufać użytkownikowi. W systemach o wyższej krytyczności

czas ten powinien prawdopodobnie być jeszcze krótszy niż 15 minut, wtedy jednak użytkownik byłby wylogowywany z konta nawet w przypadku krótkotrwałych przerw jak wyjście do toalety.

Autoryzacja użytkownika i weryfikacja uprawnień

Aplikacja od strony frontendu sprawdza, czy użytkownik ma taką rolę jaką powinien, aby móc korzystać z danej funkcjonalności (tak jak opisano w tabeli na stronie 1). Oznacza to, że np. użytkownik z rolą „personel medyczny" nie będzie w stanie wejść na stronę, na której są notatki medyczne – przy próbie wyświetli się komunikat, że nie jest uprawniony do korzystania z tej funkcjonalności. Oczywiście takie zabezpieczenie może być cenne z punktu widzenia UI/UX, ale nie z poziomu bezpieczeństwa. W dalszym ciągu bowiem użytkownik będzie w stanie – jeśli podejrzy adres URL odpowiedniego endpointa – wysłać zapytanie np. poleceniem *curl*, jeśli by nawet miał umieścić w nagłówku token JWT pobrany z *localStorage* przeglądarki. Dlatego w naszym projekcie autoryzacja dostępu jest zaimplementowana nie tylko z poziomu frontendu, ale także backendu – na podstawie odkodowanego tokena JWT pobierana jest informacja o roli użytkownika, a następnie podejmowana jest decyzja, czy udzielić użytkownikowi dostępu do zasobu, czy nie. W przypadku niepowodzenia nie jest zwracany szczegółowy komunikat, jedynie błąd z kodem 401, aby nie udzielać użytkownikowi zbyt wiele szczegółowych informacji o działaniu systemu.

Podsumowanie

Nasz system, tak jak każdy inny, z pewnością nie jest odporny na kryptograficzne ataki, jednak dołożono wszelkich starań, aby zaimplementować niezbędne zabezpieczenia zgodnie z istniejącymi i powszechnie przyjętymi normami, tak aby zminimalizować wszelkie ryzyko.

Instalacja i użycie aplikacji

Do uruchomienia aplikacji niezbędne będą:

- Język Python w wersji najlepiej 3.10
- Django (*pip install django)*
- django-rest-framework (*pip install djangorestframework)*
- django-cors-headers (*pip install django-cors-headers)*
- PyJWT oraz cryptography (*pip install pyjwt[crypto])*
- Bcypt (*pip install bcrytp)*
- Node.js i pakiet npm

Projekt składa się z dwóch głównych katalogów – backend i frontend. Aby aplikacja działa, należy uruchomić osobno dwa terminale – jeden w folderze backend, a drugi frontend. W folderze backend, po zainstalowaniu niezbędnych pakietów, należy uruchomić komendy:

python manage.py makemigrations

python manage.py migrate

python manage.py runserver

Następnie w folderze frontend uruchamiamy komendy:

npm install

npm run dev

Po czym wchodzimy w adres pokazany w terminalu z folderu frontend. Tym sposobem powinien ukazać się ekran logowania do witryny. Można następnie założyć konto, wprowadzając do formularza odpowiednie dane i hasło, które spełnia opisane wcześniej warunki. Następnie będzie się można zalogować na konto, a na ekranie powinna pojawić się taka strona startowa.

Strona startowa aplikacji

Następnie można korzystać z funkcjonalności projektu wchodząc w przyciski „Wizyty” lub „Notatki medyczne”, stosownie do roli użytkownika, tak jak opisano w tabeli na stronie 1.